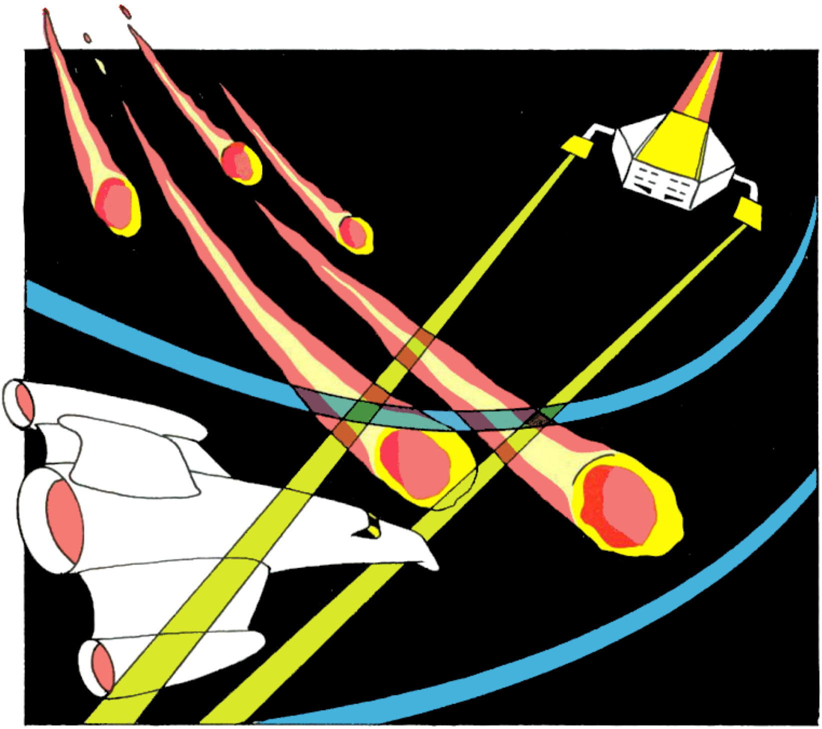
STARBLASTER



2015-10-13

4am
to deprotect
and preserve

Contents

0 In Which Various Automated Tools Fail In Interesting Ways 4

1 In Which We Reap The Benefits Of Automation Before The Robots Rise Up And Kill Us All 7

2 In Which Things Get A Little Weird 14

3 In Which Things Get Really Weird 21

4 In Which We Snatch Defeat From The Jaws of Victory, And Vice-Versa 32

5 0boot 41

6 6 + 2 48

7 Back to 0boot 53

8 0boot boot1 62

9 Compatibility is Tough, Let's Go Shopping 76

A Acknowledgements 80

B Changelog 80

```
-----------------Starblaster------------------
A 4am crack                        2015-10-12
-----------------------. updated 2015-10-13
                       |___________________________
```

```
Name: Starblaster
Genre: arcade
Year: 1981
Authors: G. Engelstein & G. Kriegsman
Publisher: Piccadilly Software, Inc.
Media: single-sided 5.25-inch floppy
OS: custom
Previous cracks: Mr. Xerox
Similar cracks:
  #073 Ribbit
  #299 Falcons
```

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)
  tons of read errors; copy hangs with
  the drive motor on

Copy ][+ nibble editor
  T00 has a modified address prologue
    (D5 AA B5) and modified epilogues
  T01+ appears to be 4-4 encoded data
    with custom prologue/delimiter
    (neither 13 nor 16 sectors)

Disk Fixer
  not much help

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. A nibble check during
  boot?
```

The original disk switches to a hi-res graphics screen immediately on boot, then gradually displays the title screen as it loads from disk. The game is a single-load; it doesn't access the disk once it starts. I can probably extract the entire thing from memory and save it.

Next steps:

1. Trace the boot until the entire game is in memory
2. Save it (in chunks, if necessary)
3. Write it out to a standard disk with a fastloader to reproduce the original disk's boot experience

Chapter 1
In Which We Reap The Benefits Of Automation Before The Robots Rise Up And Kill Us All

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

]PR#5
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0
/!\ BOOT0 RELOCATES TO $0200
CAPTURING BOOT0 STAGE 2
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0 STAGE 2
```

Some recent upgrades to my AUTOTRACE program come in handy here. T00,S00 relocates itself to low memory, then re-uses the disk controller ROM routine to read a few sectors from track $00 (probably containing an RWTS of sorts). The pattern is common enough that I automated detection and tracing of it.

```
]BLOAD BOOT0,A$800
]CALL -151
```

```
*801L

; immediately move this code to the
; input buffer at $0200
0801-   A2 00       LDX   #$00
0803-   BD 00 08    LDA   $0800,X
0806-   9D 00 02    STA   $0200,X
0809-   E8          INX
080A-   D0 F7       BNE   $0803
080C-   4C 0F 02    JMP   $020F

*20F<80F.8FFM

; set up a nibble translate table
; at $0800
020F-   A0 AB       LDY   #$AB
0211-   98          TYA
0212-   85 3C       STA   $3C
0214-   4A          LSR
0215-   05 3C       ORA   $3C
0217-   C9 FF       CMP   #$FF
0219-   D0 09       BNE   $0224
021B-   C0 D5       CPY   #$D5
021D-   F0 05       BEQ   $0224
021F-   8A          TXA
0220-   99 00 08    STA   $0800,Y
0223-   E8          INX
0224-   C8          INY
0225-   D0 EA       BNE   $0211
0227-   84 3D       STY   $3D

; $00 into zero page $26 and $03 into
; $27 means we're probably going to be
; loading data into $0300..$03FF later.
0229-   84 26       STY   $26
022B-   A9 03       LDA   #$03
022D-   85 27       STA   $27
022F-   A6 2B       LDX   $2B
0231-   20 5D 02    JSR   $025D
```

```
*25DL

; read a sector from track $00 (this is
; actually derived from the code in the
; disk controller ROM routine at $C65C,
; but looking for an address prologue
; of "D5 AA B5" instead of "D5 AA 96")
; and using the nibble translate table
; we set up earlier at $0800
025D-   18          CLC
025E-   08          PHP
025F-   BD 8C C0    LDA   $C08C,X
0262-   10 FB       BPL   $025F
0264-   49 D5       EOR   #$D5
0266-   D0 F7       BNE   $025F
0268-   BD 8C C0    LDA   $C08C,X
026B-   10 FB       BPL   $0268
026D-   C9 AA       CMP   #$AA
026F-   D0 F3       BNE   $0264
0271-   EA          NOP
0272-   BD 8C C0    LDA   $C08C,X
0275-   10 FB       BPL   $0272
0277-   C9 B5       CMP   #$B5
0279-   F0 09       BEQ   $0284
027B-   28          PLP
027C-   90 DF       BCC   $025D
027E-   49 AD       EOR   #$AD
0280-   F0 1F       BEQ   $02A1
0282-   D0 D9       BNE   $025D
0284-   A0 03       LDY   #$03
0286-   84 2A       STY   $2A
0288-   BD 8C C0    LDA   $C08C,X
028B-   10 FB       BPL   $0288
028D-   2A          ROL
028E-   85 3C       STA   $3C
                                  [...]
```

```
0290-   BD 8C C0    LDA   $C08C,X
0293-   10 FB       BPL   $0290
0295-   25 3C       AND   $3C
0297-   88          DEY
0298-   D0 EE       BNE   $0288
029A-   28          PLP
029B-   C5 3D       CMP   $3D
029D-   D0 BE       BNE   $025D
029F-   B0 BD       BCS   $025E
02A1-   A0 9A       LDY   #$9A
02A3-   84 3C       STY   $3C
02A5-   BC 8C C0    LDY   $C08C,X
02A8-   10 FB       BPL   $02A5

; use the nibble translate table we set
; up earlier
02AA-   59 00 08    EOR   $0800,Y
02AD-   A4 3C       LDY   $3C
02AF-   88          DEY
02B0-   99 00 08    STA   $0800,Y
02B3-   D0 EE       BNE   $02A3
02B5-   84 3C       STY   $3C
02B7-   BC 8C C0    LDY   $C08C,X
02BA-   10 FB       BPL   $02B7
02BC-   59 00 08    EOR   $0800,Y
02BF-   A4 3C       LDY   $3C

; store in $0300
02C1-   91 26       STA   ($26),Y
02C3-   C8          INY
02C4-   D0 EF       BNE   $02B5
02C6-   BC 8C C0    LDY   $C08C,X
02C9-   10 FB       BPL   $02C6
02CB-   59 00 08    EOR   $0800,Y
02CE-   D0 8D       BNE   $025D
02D0-   60          RTS
```

Continuing from $0237...

```
*237L

0237-   A9 A9       LDA   #$A9
0239-   8D 0F 03    STA   $030F
023C-   A9 02       LDA   #$02
023E-   8D 10 03    STA   $0310
0241-   4C 01 03    JMP   $0301
```

This is where I need to interrupt the boot, before it jumps to $0301. Here is the relevant portion of my AUTOTRACE program that does that automatically:

```
*97BFL

; replicate the memory move
97BF-   A2 00       LDX   #$00
97C1-   BD 00 08    LDA   $0800,X
97C4-   9D 00 02    STA   $0200,X
97C7-   E8          INX
97C8-   D0 F7       BNE   $97C1

; now set up a callback to a routine
; under my control
97CA-   A9 D7       LDA   #$D7
97CC-   8D 42 02    STA   $0242
97CF-   A9 97       LDA   #$97
97D1-   8D 43 02    STA   $0243

; start the boot
97D4-   4C 0F 02    JMP   $020F
```

```
; callback is here -- move the code
; from $0300 to the graphics page so it
; can survive a reboot
97D7-    A0 00        LDY   #$00
97D9-    B9 00 03     LDA   $0300,Y
97DC-    99 00 23     STA   $2300,Y
97DF-    C8           INY
97E0-    D0 F7        BNE   $97D9

; set up markers to tell AUTOTRACE what
; to do after we reboot
97E2-    A9 83        LDA   #$83
97E4-    8D 00 01     STA   $0100
97E7-    49 A5        EOR   #$A5
97E9-    8D 01 01     STA   $0101

; turn off the slot 6 drive motor
97EC-    AD E8 C0     LDA   $C0E8

; reboot to my work disk (this will run
; AUTOTRACE again and complete the
; process)
97EF-    4C 00 C5     JMP   $C500
```

Once this reboots, AUTOTRACE saves the captured code from $0300 at $2300 into a file. Let's see what that looks like.

Chapter 2
In Which Things Get A Little Weird

```
]BLOAD BOOT0 0300-03FF,A$300
]CALL -151

*301L

0301-   78          SEI
0302-   D8          CLD

; I have no idea what this is doing
0303-   B9 00 08    LDA   $0800,Y
0306-   0A          ASL
0307-   0A          ASL
0308-   0A          ASL
0309-   99 00 08    STA   $0800,Y
030C-   C8          INY
030D-   D0 F4       BNE   $0303

; This is even weirder -- it literally
; does nothing, repeatedly. It's not a
; wait loop. Maybe a simplified version
; of a more complicated boot routine?
030F-   A9 02       LDA   #$02
0311-   A0 1E       LDY   #$1E
0313-   EA          NOP
0314-   EA          NOP
0315-   EA          NOP
0316-   EA          NOP
0317-   EA          NOP
0318-   EA          NOP
0319-   EA          NOP
031A-   EA          NOP
031B-   C8          INY
031C-   D0 F5       BNE   $0313
```

```
; write-enable RAM bank 1
031E-   AD 89 C0    LDA   $C089
0321-   AD 89 C0    LDA   $C089

; wipe all memory (including RAM bank)
0324-   20 7B 03    JSR   $037B

; write-enable RAM bank 2
0327-   AD 81 C0    LDA   $C081
032A-   AD 81 C0    LDA   $C081

; wipe all memory again (including RAM
; bank)
032D-   20 7B 03    JSR   $037B

; now *this* looks like a normal sector
; read loop (much like DOS 3.3 uses in
; its boot0 code)
0330-   A9 09       LDA   #$09
0332-   85 27       STA   $27
0334-   4A          LSR
0335-   85 39       STA   $39
0337-   85 3F       STA   $3F
0339-   84 38       STY   $38
033B-   84 3E       STY   $3E
033D-   AD 0F 03    LDA   $030F
0340-   8D 50 03    STA   $0350
0343-   AD 10 03    LDA   $0310
0346-   8D 51 03    STA   $0351
```

```
; set up entry point to disk controller
; ROM routine, based on the slot number
; we booted from (still in zero page
; $2B at this point)
0349-    A6 2B          LDX   $2B
034B-    8A             TXA
034C-    4A             LSR
034D-    4A             LSR
034E-    4A             LSR
034F-    4A             LSR
0350-    09 C0          ORA   #$C0
0352-    85 37          STA   $37
0354-    A9 5D          LDA   #$5D
0356-    85 36          STA   $36
0358-    E6 3D          INC   $3D

; turn on hi-res graphics page
035A-    AD 54 C0       LDA   $C054
035D-    AD 57 C0       LDA   $C057
0360-    AD 52 C0       LDA   $C052
0363-    AD 50 C0       LDA   $C050

; read sector via ($0036), a.k.a. $C65D
0366-    20 78 03       JSR   $0378
0369-    20 9E 03       JSR   $039E

; after 4 sectors, break out of read
; loop and continue execution at $039A
036C-    A5 3D          LDA   $3D
036E-    49 03          EOR   #$03
0370-    F0 28          BEQ   $039A
0372-    E6 39          INC   $39
0374-    E6 3D          INC   $3D
0376-    D0 EE          BNE   $0366
...
```

```
; execution continues here
039A-   A0 18       LDY   #$18
039C-   D0 62       BNE   $0400
```

So this is loading 4 sectors into the text page at $0400..$07FF. I'm guessing this is the RWTS that will read the rest of the disk. Whatever it is, I need to capture it. Luckily, there's more than enough space at $039A to put a JMP to a routine under my control. Unluckily, this code wiped memory (twice!) before reading the RWTS, so I will need to disable that before I can set up a callback.

AUTOTRACE didn't automate this part, so I'll have to write a boot tracer by hand like some kind of 20th century peasant.

```
*9600<C600.C6FFM

; first part is the same as TRACE1 --
; replicate the memory move and set up
; a callback before jumping to $0301
96F8-   A2 00       LDX   #$00
96FA-   BD 00 08    LDA   $0800,X
96FD-   9D 00 02    STA   $0200,X
9700-   E8          INX
9701-   D0 F7       BNE   $96FA

; set up callback #1
9703-   A9 10       LDA   #$10
9705-   8D 42 02    STA   $0242
9708-   A9 97       LDA   #$97
970A-   8D 43 02    STA   $0243
```

```
; start the boot
970D-   4C 0F 02     JMP   $020F

; callback #1 is here --
; disable memory wipe subroutine at
; $037B
9710-   A9 60        LDA   #$60
9712-   8D 7B 03     STA   $037B

; set up callback #2 before branch to
; $0400
9715-   A9 4C        LDA   #$4C
9717-   8D 9A 03     STA   $039A
971A-   A9 27        LDA   #$27
971C-   8D 9B 03     STA   $039B
971F-   A9 97        LDA   #$97
9721-   8D 9C 03     STA   $039C

; continue the boot
9724-   4C 01 03     JMP   $0301

; callback #2 is here --
; copy code from text page to graphics
; page so it can survive a reboot
9727-   A2 04        LDX   #$04
9729-   A0 00        LDY   #$00
972B-   B9 00 04     LDA   $0400,Y
972E-   99 00 24     STA   $2400,Y
9731-   C8           INY
9732-   D0 F7        BNE   $972B
9734-   EE 2D 97     INC   $972D
9737-   EE 30 97     INC   $9730
973A-   CA           DEX
973B-   D0 EE        BNE   $972B
```

```
; turn off slot 6 drive motor
973D-    AD E8 C0     LDA   $C0E8

; reboot to my work disk
9740-    4C 00 C5     JMP   $C500

*BSAVE TRACE,A$9600,L$143
*9600G
...reboots slot 6...
...reboots slot 5...

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

Chapter 3
In Which Things Get Really Weird

```
]CALL -151

The code at $0400..$07FF is loaded in
$2400..$27FF, so everything is off by
$2000. Relative branches will look
correct, but absolute addresses will
be off by $2000.

*2400L

; set up The Badlands in zero page
; (wipes memory and displays "REBOOT")
2400-   A2 2A       LDX   #$2A
2402-   BD 15 06    LDA   $0615,X
2405-   95 00       STA   $00,X
2407-   CA          DEX
2408-   D0 F8       BNE   $2402

; write language card RAM bank 2
240A-   2C 81 C0    BIT   $C081
240D-   2C 81 C0    BIT   $C081

; copy ROM to RAM (defense against the
; dark arts of boot tracers and hacker-
; friendly modified hardware)
2410-   A2 30       LDX   #$30
2412-   A0 00       LDY   #$00
2414-   A9 D0       LDA   #$D0
2416-   85 2D       STA   $2D
2418-   A9 00       LDA   #$00
241A-   85 2C       STA   $2C
241C-   B1 2C       LDA   ($2C),Y
241E-   91 2C       STA   ($2C),Y
2420-   C8          INY
2421-   D0 F9       BNE   $241C
2423-   E6 2D       INC   $2D
2425-   CA          DEX
2426-   D0 F4       BNE   $241C
```

```
; read/write LC RAM bank 2
2428-    AD 83 C0     LDA   $C083
242B-    AD 83 C0     LDA   $C083

; set reset vector to $0002 (including
; the low level reset vector at $FFFC)
242E-    A9 02        LDA   #$02
2430-    8D F2 03     STA   $03F2
2433-    8D FC FF     STA   $FFFC
2436-    A9 00        LDA   #$00
2438-    8D F3 03     STA   $03F3
243B-    8D FD FF     STA   $FFFD
243E-    49 A5        EOR   #$A5
2440-    8D F4 03     STA   $03F4

; don't know why (it's $A9)
2443-    AD 0F 03     LDA   $030F
2446-    85 47        STA   $47

; display hi-res screen (uninitialized)
2448-    2C 50 C0     BIT   $C050
244B-    2C 52 C0     BIT   $C052
244E-    2C 54 C0     BIT   $C054
2451-    2C 57 C0     BIT   $C057
2454-    A2 20        LDX   #$20
2456-    A0 00        LDY   #$00

; manually pushing a byte to the stack
2458-    A9 AA        LDA   #$AA
245A-    48           PHA

; checking whether the value we stored
; earlier is still $A9
245B-    A5 47        LDA   $47
245D-    C9 A9        CMP   #$A9

; if not, branch
245F-    D0 04        BNE   $2465
```

```
; pull that value off the stack and put
; another one on
2461-   68            PLA
2462-   A9 2A         LDA   #$2A
2464-   48            PHA

; execution continues here (from $045F
; or by falling through)
2465-   68            PLA
```

OK, I just figured this out. The old Apple II monitor used $45..$49 to save and restore registers and flags during Step and Trace commands. So if the value of this zero page address isn't $A9, it either means that

- it was never $A9 to begin with (someone "booted" the disk with their own boot0 that didn't set $030F like the real boot0 did), or

- it changed in the last few instructions (someone is trying to trace the boot one instruction at a time, and the monitor overwrote $47 while doing so)

The program responds to this alarming turn of events by leaving the wrong value in the accumulator, for purposes unknown.

```
; fill hi-res graphics screen (note: if
; the previous check detected that
; someone was debugging, this will be a
; different color)
2466-   84 37        STY   $37
2468-   86 38        STX   $38
246A-   91 37        STA   ($37),Y
246C-   C8           INY
246D-   D0 FB        BNE   $246A
246F-   E6 38        INC   $38
2471-   CA           DEX
2472-   D0 F6        BNE   $246A

; continue elsewhere
2474-   4C 8D 05     JMP   $058D

*258DL

; initialization of... things
258D-   A9 00        LDA   #$00
258F-   85 35        STA   $35
2591-   85 42        STA   $42
2593-   85 40        STA   $40
2595-   A9 04        LDA   #$04
2597-   85 30        STA   $30
2599-   A9 0A        LDA   #$0A
259B-   85 44        STA   $44
259D-   E6 42        INC   $42
259F-   A9 04        LDA   #$04
25A1-   85 30        STA   $30

25A3-   A5 42        LDA   $42
25A5-   A6 2B        LDX   $2B
25A7-   20 EB 04     JSR   $04EB
```

```
*24EBL

; this appears to be the routine that
; moves the drive arm to a specified
; slot
24EB-   86 2B       STX   $2B
24ED-   AA          TAX
24EE-   18          CLC

; get desired track from a table
24EF-   BD 76 05    LDA   $0576,X
24F2-   A6 2B       LDX   $2B
24F4-   85 3E       STA   $3E

; if we're already on that track, do
; nothing
24F6-   C5 40       CMP   $40
24F8-   F0 4F       BEQ   $2549
```

This is the track array at $0576:

```
0576- 00 41
0578- 3B 3E 38 32 35 2F 29 2C
0580- 23 26 20 1A 1D 17 11 14
0588- 0E 08 0B 05 02
```

Ah, those are actually phases, not tracks. I can tell because the highest one is $41, which would be track $20.5. It's a weird list; it skips around instead of loading from consecutive tracks. It reads from (decimal) tracks 32.5, 29.5, 31, 28, 25, 26.5, 23.5, 20.5, 22, 17.5, 19, 16, 13, 14.5, 11.5, 8.5, 10, 7, 4, 5.5, and 2 -- in that order. Presumably this was to make it difficult to figure out how to make a successful bit copy. Also, several of them are half tracks, which explains why my EDD bit copy failed.

Continuing from $05AA...

```
25AA-   A5 44       LDA   $44
25AC-   85 36       STA   $36
25AE-   A6 2B       LDX   $2B
25B0-   20 79 04    JSR   $0479

*2479L

; this is the main track read routine
2479-   A0 50       LDY   #$50
247B-   84 30       STY   $30
247D-   88          DEY
247E-   D0 04       BNE   $2484
2480-   C6 30       DEC   $30

; sets carry and exits (not shown)
2482-   F0 F3       BEQ   $2477
2484-   BD 8C C0    LDA   $C08C,X
2487-   10 FB       BPL   $2484

; look for custom prologue "DF AD DE"
2489-   C9 DF       CMP   #$DF
248B-   D0 F0       BNE   $247D
248D-   BD 8C C0    LDA   $C08C,X
2490-   10 FB       BPL   $248D
2492-   C9 AD       CMP   #$AD
2494-   D0 F3       BNE   $2489
2496-   BD 8C C0    LDA   $C08C,X
2499-   10 FB       BPL   $2496
249B-   C9 DE       CMP   #$DE
249D-   D0 EA       BNE   $2489
249F-   A0 00       LDY   #$00
24A1-   98          TYA
24A2-   85 31       STA   $31
```

```
; $F5 is epilogue marker
24A4-   BD 8C C0    LDA   $C08C,X
24A7-   10 FB       BPL   $24A4
24A9-   C9 F5       CMP   #$F5
24AB-   F0 24       BEQ   $24D1

; data is 4-4 encoded with a rolling
; checksum
24AD-   38          SEC
24AE-   85 32       STA   $32
24B0-   BD 8C C0    LDA   $C08C,X
24B3-   10 FB       BPL   $24B0
24B5-   2A          ROL
24B6-   25 32       AND   $32

; final data is stored in ($35), which
; was initialized = $0A00
24B8-   91 35       STA   ($35),Y
24BA-   45 31       EOR   $31
24BC-   C8          INY
24BD-   D0 E3       BNE   $24A2

; increment target page
24BF-   E6 36       INC   $36
24C1-   85 31       STA   $31
```

```
; look for between-sector delimiter
; (also $F5)
24C3-   BD 8C C0     LDA   $C08C,X
24C6-   10 FB        BPL   $24C3
24C8-   C9 F5        CMP   #$F5
24CA-   F0 05        BEQ   $24D1
24CC-   38           SEC
24CD-   85 32        STA   $32
24CF-   B0 DF        BCS   $24B0
24D1-   BD 8C C0     LDA   $C08C,X
24D4-   10 FB        BPL   $24D1
24D6-   C9 F5        CMP   #$F5
24D8-   F0 F7        BEQ   $24D1
24DA-   38           SEC
24DB-   85 32        STA   $32
24DD-   BD 8C C0     LDA   $C08C,X
24E0-   10 FB        BPL   $24DD
24E2-   2A           ROL
24E3-   25 32        AND   $32
24E5-   C5 31        CMP   $31
24E7-   D0 8E        BNE   $2477

; clear carry if all data was read and
; the final checksum matched
24E9-   18           CLC
24EA-   60           RTS
```

Continuing from $05B3...

```
*25B3L

; if track read failed, branch
25B3-   B0 48        BCS   $25FD
```

```
*25FDL

; failure path (from $05B3 when the
; track read subroutine sets the carry)
; decrement death counter, eventually
; give up
25FD-   C6 30       DEC   $30
25FF-   D0 A2       BNE   $25A3

; give up -- turn off drive motor and
; jump to The Badlands
2601-   BD 88 C0    LDA   $C088,X
2604-   4C 02 00    JMP   $0002
```

Continuing from $05B5...

```
*25B5L

; increment starting page for the next
; track read (only 8 sectors per track)
25B5-   18          CLC
25B6-   A9 08       LDA   #$08
25B8-   65 44       ADC   $44
25BA-   85 44       STA   $44

; done yet?
25BC-   C9 9A       CMP   #$9A

; nope, branch to read another track
25BE-   90 DD       BCC   $259D

; entire game is in memory -- turn off
; the drive motor
25C0-   BD 88 C0    LDA   $C088,X
```

```
; set some... stuff (no apparent rhyme
; or reason, but I bet it's important)
25C3-    A9 FE       LDA   #$FE
25C5-    8D 00 02    STA   $0200
25C8-    A9 00       LDA   #$00
25CA-    85 B1       STA   $B1
25CC-    A9 AA       LDA   #$AA
25CE-    85 36       STA   $36
25D0-    A9 81       LDA   #$81
25D2-    85 FD       STA   $FD

; copy The Badlands again
25D4-    A2 30       LDX   #$30
25D6-    BD 08 06    LDA   $0608,X
25D9-    9D 05 93    STA   $9305,X
25DC-    CA          DEX
25DD-    D0 F7       BNE   $25D6

; read/write LC RAM bank 2 (again)
25DF-    AD 83 C0    LDA   $C083
25E2-    AD 83 C0    LDA   $C083

; set new reset vector
25E5-    A9 93       LDA   #$93
25E7-    8D FD FF    STA   $FFFD
25EA-    8D F3 03    STA   $03F3
25ED-    A9 00       LDA   #$00
25EF-    8D FC FF    STA   $FFFC
25F2-    8D F2 03    STA   $03F2
25F5-    A9 36       LDA   #$36
25F7-    8D F4 03    STA   $03F4

; start the game
25FA-    4C 00 60    JMP   $6000
```

Chapter 4
In Which We Snatch Defeat From The Jaws of Victory, And Vice-Versa

I can interrupt the boot at $05FA and capture the entire game in memory.

```
*9600<C600.C6FFM

; same as previous trace
96F8-   A2 00       LDX   #$00
96FA-   BD 00 08    LDA   $0800,X
96FD-   9D 00 02    STA   $0200,X
9700-   E8          INX
9701-   D0 F7       BNE   $96FA
9703-   A9 10       LDA   #$10
9705-   8D 42 02    STA   $0242
9708-   A9 97       LDA   #$97
970A-   8D 43 02    STA   $0243
970D-   4C 0F 02    JMP   $020F
9710-   A9 60       LDA   #$60
9712-   8D 7B 03    STA   $037B
9715-   A9 4C       LDA   #$4C
9717-   8D 9A 03    STA   $039A
971A-   A9 27       LDA   #$27
971C-   8D 9B 03    STA   $039B
971F-   A9 97       LDA   #$97
9721-   8D 9C 03    STA   $039C
9724-   4C 01 03    JMP   $0301

; final callback is here --
; set up an unconditional break to the
; monitor instead of starting the game
9727-   A9 59       LDA   #$59
9729-   8D FB 05    STA   $05FB
972C-   A9 FF       LDA   #$FF
972E-   8D FC 05    STA   $05FC

; continue the boot
9731-   4C 00 04    JMP   $0400
```

```
*BSAVE TRACE2,A$9600,L$134
*9600G
...reboots slot 6...
...read read read...
<beep>

*6000G
...crashes at $9307...
```

Curses! Foiled in my moment of triumph!

If I had to guess, I'd say there is a routine early on in the game code that checks $0200 (set to $FE at $05C5). There were also several other zero page locations initialized around the same time, after all the disk activity but before jumping to the game code. Those are getting lost when I break into the monitor. (For example, $0036 is part of the output vector, which gets reset to default values by $FF59.)

```
]PR#5
]BRUN TRACE2
...reboots slot 6...
...read read read...
<beep>
```

Now to recreate the initialization from $05C3..$05D3, and immediately start the game:

```
*200:FE N B1:0 N 36:AA N FD:81 N 6000G
...works...
```

Excellent. I'll be sure to initialize those properly when I recreate the bootloader. Now, I need to actually save the game code to my work disk.

```
]PR#5
]BRUN TRACE2
...reboots slot 6...
...read read read...
<beep>
```

Taking advantage of my whopping 128K, I can type in a short routine that copies main memory to aux memory. All of aux memory will remain undisturbed while I reboot to my work disk.

```
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   $3E
0310-   A9 BE       LDA   #$BE
0312-   85 3F       STA   $3F
0314-   38          SEC
0315-   4C 11 C3    JMP   $C311
```

```
$C311 is the AUXMOVE routine, which is
available on every Apple II with 128K.
It takes four 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

*300G     ; copy $0800..$BEFF to auxmem
*C500G    ; reboot to my work disk
...
]CALL -151

Now a very similar routine to copy it
all back from auxmem:

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   $3E
0310-   A9 BE        LDA   #$BE
0312-   85 3F        STA   $3F
0314-   18           CLC
0315-   4C 11 C3     JMP   $C311

*300G     ; copy $0800..$BEFF aux->main
```

My work disk boot to Diversi-DOS 64K, which relocates DOS to the language card and only requires one page of main memory (at $BF00). Diversi-DOS also has no problem saving files larger than $8000 bytes, which is one of those limits that you never think about until you hit it. (Actually, this is true of most limits.)

```
*BSAVE OBJ 0A00-99FF,A$A00,L$9000
*CATALOG

C1983 DSR^C#254
307 FREE

 A 019 HELLO
 B 005 AUTOTRACE
 B 003 BOOT0
 B 003 BOOT0 0300-03FF
 B 003 TRACE
 B 006 BOOT1 0400-07FF
 B 003 TRACE2
 B 147 OBJ 0A00-99FF
```

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]

; 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 $0A00, and RWTS
; write command ($02)
0388- 01 60 01 00 01 00 FB F7
0390- 00 0A 00 00 02 00 00 60

*BSAVE MAKE,A$300,L$98

*300G          ; write game to disk
```

Now I have the entire game on tracks $01-$09 of a standard 16-sector disk.

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.

It's 2015. Let's write a bootloader. Oh wait, I already wrote one and called it 4boot. It's fast and it's 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 5
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 /
  $FE88 (IN#0, pushed at $0831)
  $FE92 (PR#0, pushed at $0831)
  $00D4 (turn off drive, clean up, jump
         to game entry point)
--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 6
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 7
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 and filled it with a repeated byte. Then you got to watch the title page progressively load over it.

```
; this loop reproduces that effect
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-   A2 20        LDX   #$20
089E-   A0 00        LDY   #$00
08A0-   A9 2A        LDA   #$2A
08A2-   99 00 20     STA   $2000,Y
08A5-   C8           INY
08A6-   D0 FA        BNE   $08A2
08A8-   EE A4 08     INC   $08A4
08AB-   CA           DEX
08AC-   D0 F4        BNE   $08A2
```

[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.]

```
; And that's all she wrote.
08AE-   60           RTS
```

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 8
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_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      /!\

; game-specific initialization
; (copied from $05C3)
00D8-    A9 FE        LDA   #$FE
00DA-    8D 00 02     STA   $0200
00DD-    A9 00        LDA   #$00
00DF-    85 B1        STA   $B1
00E1-    A9 AA        LDA   #$AA
00E3-    85 36        STA   $36
00E5-    A9 81        LDA   #$81
00E7-    85 FD        STA   $FD

; jump to game entry point
00E9-    4C 00 60     JMP   $6000
```

And still four bytes to spare (before we hit the sector array at $00F0).

Chapter 9
Compatibility is Tough, Let's Go Shopping

The original Starblaster game did not work on the Apple IIgs. After several seconds of furious debugging, qkumba figured out why.

```
]PR#5
...
]BLOAD OBJ 0A00-99FF,A$A00
]CALL -151

*6000L

; start of game
6000-    A9 00         LDA    #$00
6002-    8D 1E 8A      STA    $8A1E
6005-    8D 1F 8A      STA    $8A1F
6008-    8D 20 8A      STA    $8A20
600B-    85 09         STA    $09
600D-    85 08         STA    $08
600F-    85 07         STA    $07
6011-    8D 00 03      STA    $0300
6014-    A9 30         LDA    #$30
6016-    8D 02 03      STA    $0302
6019-    A0 00         LDY    #$00
601B-    98            TYA

; a loop to check that the game code is
; unmodified
601C-    A2 0A         LDX    #$0A
601E-    8E 23 60      STX    $6023
6021-    59 00 00      EOR    $0000,Y
6024-    88            DEY
6025-    D0 FA         BNE    $6021
6027-    E8            INX
6028-    E0 93         CPX    #$93
602A-    D0 F2         BNE    $601E
```

```
; proper checksum is stored in $00FD
; (this explains why it was set at the
; end of the bootloader)
602C-   C5 FD       CMP   $FD
602E-   F0 07       BEQ   $6037
...
6037-   4C 8A 87    JMP   $878A

*878AL

878A-   A9 FF       LDA   #$FF
878C-   8D 89 87    STA   $8789
878F-   AD 50 C0    LDA   $C050
8792-   AD 57 C0    LDA   $C057
8795-   AD 52 C0    LDA   $C052
8798-   AD 54 C0    LDA   $C054
879B-   A9 00       LDA   #$00
879D-   AA          TAX
879E-   20 D3 69    JSR   $69D3
87A1-   A9 02       LDA   #$02
87A3-   85 0A       STA   $0A
87A5-   A9 00       LDA   #$00
87A7-   20 BC 62    JSR   $62BC

*62BCL

62BC-   20 4F 63    JSR   $634F

*634FL

6342-   A0 B6       LDY   #$B6
6344-   B9 80 FF    LDA   $FF80,Y <-- !
6347-   C9 AA       CMP   #$AA
6349-   F0 03       BEQ   $634E
634B-   4C 05 93    JMP   $9305
634E-   60          RTS
```

The indexed fetch at $6344 is trying to get the value of $0036, which was set to $AA late in the original bootloader. But on the Apple IIgs, $FF80 + $B6 doesn't wrap around to $0036; it reads $10036 instead. That address was never initialized, so the check fails and the game exits via $9305.

Solution: instead of initializing $0036 in the post-read routine at $00D5, initialize $FF80 + $B6 instead. This will hit the exact address that the game checks later, on every machine. It also has the advantage of not requiring any changes to the actual game code, which means I don't need to worry about disabling the integrity check or recomputing the checksum.

```
00D5-   AD E8 C0    LDA   $C0E8
00D8-   A9 FE       LDA   #$FE
00DA-   8D 00 02    STA   $0200
00DD-   A9 00       LDA   #$00
00DF-   85 B1       STA   $B1
00E1-   A0 B6       LDY   #$B6      <-- 1
00E3-   A9 AA       LDA   #$AA      <-- 2
00E5-   99 80 FF    STA   $FF80,Y   <-- 3
00E8-   A9 81       LDA   #$81
00EA-   85 FD       STA   $FD
00EC-   4C 00 60    JMP   $6000
```

And still one byte to spare.

Quod erat liberandum.

Acknowledgements

Many thanks to qkumba for writing 0boot and allowing me to present it here, to John Brooks for testing on a real IIgs, and to numerous others for testing and general encouragement.

~

Changelog

2015-10-13

- typos [thanks Martin Haye]

2015-10-12

- initial release