PAC-MAN

Contents

----------Pac-Man----------
A 4am crack 2015-03-10

Name: Pac-Man
Genre: arcade
Year: 1984
Publisher: Datasoft, Inc.
Media: single-sided 5.25-inch floppy
OS: custom
Other versions: Asimov has an uncracked .nib image; all other copies appear to be the Atarisoft version

Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

COPYA
  immediate disk read error

Locksmith Fast Disk Backup
  unable to read any track

EDD 4 bit copy (no sync, no count)
  works

Copy][+ nibble editor
  T00 -> modified address epilogue
    (CC AA EB)
  T01-T0C -> obviously data, but no
    visible structure, no address or
    data fields
  T0D+ -> unformatted

Disk Fixer
  ["O" -> "Input/Output Control"]
    set Address Epilogue to "CC AA EB"
  T00 readable -> custom bootloader
  everything else still a black box

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

Why didn't Locksmith FDB work?
  ditto

The original disk switches to an uninitialized 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
Everyone Boots

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

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

As expected, my AUTOTRACE script didn't get very far, since this disk uses a custom bootloader.

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

*801L

0801-   74          ???
0802-   85 B0       STA   $B0
0804-   58          CLI
0805-   6A          ROR
0806-   EA          NOP
0807-   73          ???
0808-   4B          ???
0809-   CD 53 CA    CMP   $CA53
080C-   C8          INY
080D-   42          ???
080E-   1A          ???
080F-   3D 63 CA    AND   $CA63,X
```

Um, there must be some mistake. This is not executable code. I mean, it's just not. Yet, address $0800 contains $01, so this is the only sector the disk controller ROM reads into memory before passing control.

I have no idea how this disk boots.

.
. [time passes]
.

According to <http://www.ataripreservation.org/websites/freddy.offenga/illopc31.txt>, $74 is an undocumented 6502 opcode that takes a single byte argument and does nothing. Like a double NOP, but with two bytes instead of one.

According to <http://www.6502.org/tutorials/65c02opcodes.html>, $74 is relatively obscure variant of the STZ (STore Zero) instruction, which was introduced in the 65C02. This form of the STZ instruction takes a one byte operand, a zero page memory location.

The disassembler built into the Apple monitor assumes all unknown opcodes are single-byte, so it misrepresents opcode $74 as a single-byte instruction and incorrectly prints a three-byte JMP instruction on the next line. When the 65c02 made some of those opcodes valid instructions, the monitor disassembler was never updated with information on their mnemonics or arguments, so it has the same problem.

Opcode $64 does nothing of consequence on either CPU, but more importantly, it does nothing in 2 bytes instead of 1.

This is the actual code:

```
0801-   74 85        DOP   $85,X ; NOPx2
0803-   B0 58        BCS   $085D
```

The carry bit is always set coming out of the disk controller routine, so this is an unconditional jump.

Everyone boots.

Chapter 2
Too Fast, Too Furious

```
*85DL

085D-   A9 CA        LDA   #$CA
085F-   8D 9D 08     STA   $089D
0862-   B0 31        BCS   $0895
```

Another unconditional jump (since the carry is still set).

```
*895L

; X register has the slot number (x16)
0895-   8A           TXA
0896-   48           PHA

; decrypt part of boot0 into text page
0897-   A0 98        LDY   #$98
0899-   B9 00 08     LDA   $0800,Y
089C-   49 AA        EOR   #$AA
089E-   99 00 07     STA   $0700,Y
08A1-   88           DEY
08A2-   D0 F5        BNE   $0899

; slot number (x16)
08A4-   68           PLA
08A5-   4A           LSR
08A6-   4A           LSR
08A7-   4A           LSR
08A8-   4A           LSR

; Y register is 0 at this point (after
; decryption loop), so this is just
; triggering write access on RAM bank 2
08A9-   99 81 C0     STA   $C081,Y

; munge slot number into $Cx byte
08AC-   09 C0        ORA   #$C0
08AE-   85 3F        STA   $3F
08B0-   8D 94 07     STA   $0794
```

```
; continue with (decrypted) bootloader
08B3-   4C 05 07    JMP   $0705
```

I'll reproduce the decryption loop and save it somewhere other than the text page.

```
*8A0:27
*8A4:60
*897L

0897-   A0 98       LDY   #$98
0899-   B9 00 08    LDA   $0800,Y
089C-   49 AA       EOR   #$AA
089E-   99 00 27    STA   $2700,Y
08A1-   88          DEY
08A2-   D0 F5       BNE   $0899
08A4-   60          RTS

*897G
*2705L

2705-   C0 40       CPY   #$40
2707-   D9 E1 67    CMP   $67E1,Y
270A-   F9 60 62    SBC   $6260,Y
270D-   E8          INX
270E-   B0 97       BCS   $26A7
2710-   C9 60       CMP   #$60
2712-   F9 E1 A0    SBC   $A0E1,Y
2715-   ED 92 63    SBC   $6392
```

Well, that's technically real code, but I get the feeling it's not the code I was looking for.

Let's back up.

.
. [time passes]
.

Oh look, here's the problem:

```
*85DL

085D-   A9 CA       LDA   #$CA
085F-   8D 9D 08    STA   $089D
```

That's changing the decryption loop, specifically the XOR key. And I blew right past it.

```
*89D:CA
*897G
*2705L

2705-   A0 20       LDY   #$20
2707-   B9 81 07    LDA   $0781,Y
270A-   99 00 02    STA   $0200,Y
270D-   88          DEY
270E-   D0 F7       BNE   $2707
```

Look at that. Real code after all.

```
*2781L

2781-   A9 00       LDA   #$00
2783-   AA          TAX
2784-   A0 BC       LDY   #$BC
2786-   9D 00 04    STA   $0400,X
2789-   CA          DEX
278A-   D0 FA       BNE   $2786
278C-   EE 07 02    INC   $0207
278F-   88          DEY
2790-   D0 F4       BNE   $2786
2792-   4C 00 C6    JMP   $C600
```

That would be "The Badlands", a.k.a. the point of no return. It clears most of main memory and reboots slot 6. It gets copied to $0200. Don't end up in The Badlands.

```
*2710L

2710-   A9 00       LDA   #$00

; finish triggering the switch to RAM
; bank 2 (again, Y is 0 by this point)
2712-   99 81 C0    STA   $C081,Y

; set the reset vector in page 3 and
; the lower-level reset vector at $FFFC
; to point to The Badlands at $0200
2715-   8D F2 03    STA   $03F2
2718-   8D FC FF    STA   $FFFC
271B-   A9 02       LDA   #$02
271D-   8D F3 03    STA   $03F3
2720-   8D FD FF    STA   $FFFD
2723-   A9 A7       LDA   #$A7
2725-   8D F4 03    STA   $03F4

; zap encrypted boot0 (at $0800)
2728-   A9 40       LDA   #$40
272A-   A0 00       LDY   #$00
272C-   99 00 08    STA   $0800,Y
272F-   88          DEY
2730-   D0 FA       BNE   $272C

; switch to reading RAM bank 2
2732-   99 80 C0    STA   $C080,Y

; finish setting up vector to $Cx5C (to
; reuse the disk controller ROM routine
; to read more sectors)
2735-   A9 5C       LDA   #$5C
2737-   C8          INY
2738-   85 3E       STA   $3E
```

```
; $01 in $0800
273A-   8C 00 08    STY   $0800

; looks like a multi-sector read loop
273D-   AC 80 07    LDY   $0780
2740-   F0 22        BEQ   $2764
2742-   A9 07        LDA   #$07
2744-   48           PHA
2745-   B9 6F 07    LDA   $076F,Y
2748-   85 3D        STA   $3D
274A-   CE 80 07    DEC   $0780
274D-   AD 7F 07    LDA   $077F
2750-   85 27        STA   $27
2752-   A9 3D        LDA   #$3D
2754-   48           PHA
2755-   CE 7F 07    DEC   $077F
2758-   98           TYA
2759-   48           PHA
275A-   6C 3E 00    JMP   ($003E)
```

OK, this is fascinating. ($3E) points to the start of the "read sector" routine in the disk controller ROM ($C65C if we booted from slot 6). That routine reads the physical sector given in $3D into the page given in $27. When it's done, it jumps to $0801. This is not anything extraordinary; DOS 3.3 works the same way.

What's different here is that $0801 isn't connected to the read loop (as it is in DOS 3.3). We flooded page 8 with $40, which is the rarely used "RTI" instruction. (I had to look that up.)

What does "RTI" do? It pulls one byte off the stack and sets the processor flags (like PLP). Then it pulls an address off the stack and jumps to it. But unlike RTS, the address on the stack is the actual address, not the actual address -1.

The PHA at $0744 puts $07 on the stack.
The PHA at $0754 puts $3D on the stack.
The PHA at $0759 puts $00 on the stack.
Then we jump to ($3E), which is $C65C, which reads a sector and jumps to $801, which is an RTI, which pops the stack and sets all the processor flags to 0, then pops the stack two more times and jumps to $073D.

So it's a loop.

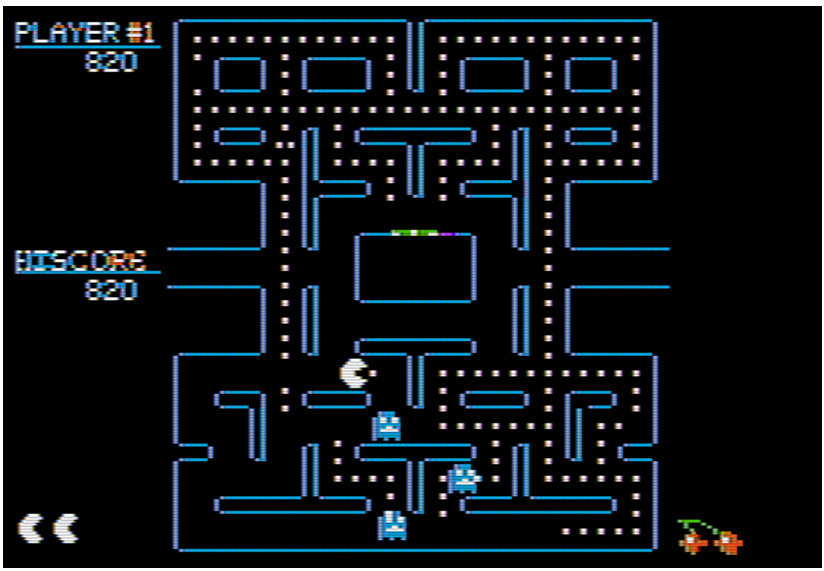
$780 is the sector count. $77F is the highest page (decremented after each sector read).

```
*277F.2780

277F- 05
2780- 02
```

So this reads two sectors into $0400..$05FF and continues at $0764 (by the BEQ at $0740).

```
*2764L

2764-    EE 7F 07     INC    $077F
2767-    EE 7F 07     INC    $077F
276A-    A6 2B        LDX    $2B
276C-    4C 84 04     JMP    $0484
```

The next stage of the boot begins at $0484.



Chapter 3
Who Needs Sectors Anyway?

Let's capture the code that ends up at $0400. I'm betting it's an RWTS of some sort, and that it loads the rest of the disk.

```
*9600<C600.C6FFM

; set up the callback after boot0 is
; decrypted but before we jump to it
96F8-   08          PHP
96F9-   48          PHA
96FA-   A9 09       LDA   #$09
96FC-   8D B4 08    STA   $08B4
96FF-   A9 97       LDA   #$97
9701-   8D B5 08    STA   $08B5
9704-   68          PLA
9705-   28          PLP

; start the boot
9706-   4C 01 08    JMP   $0801

; first callback is here -- set up the
; next callback after boot0 loads the
; RWTS but before we jump to it
9709-   08          PHP
970A-   48          PHA
970B-   A9 1A       LDA   #$1A
970D-   8D 6D 07    STA   $076D
9710-   A9 97       LDA   #$97
9712-   8D 6E 07    STA   $076E
9715-   68          PLA
9716-   28          PLP

; continue the boot
9717-   4C 05 07    JMP   $0705
```

```
; second callback is here -- copy the
; RWTS (and decrypted boot0) 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 the slot 6 drive motor
9730-   AD E8 C0    LDA   $C0E8

; switch to ROM
9733-   AD 82 C0    LDA   $C082

; reboot to my work disk
9736-   4C 00 C5    JMP   $C500

*BSAVE TRACE2,A$9600,L$139
*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOT1 0400-07FF,A$2400,L$400
]CALL -151
*2484L

2484-   A9 0F       LDA   #$0F
2486-   48          PHA             ; hmm
```

```
; switch to hi-res graphics screen 1
; (uninitialized)
2487-   8D 50 C0     STA   $C050
248A-   8D 57 C0     STA   $C057
248D-   A9 0F        LDA   #$0F
248F-   8D 52 C0     STA   $C052
2492-   8D 54 C0     STA   $C054
2495-   85 FE        STA   $FE

; turn on the drive motor
2497-   A6 2B        LDX   $2B
2499-   BD 89 C0     LDA   $C089,X

; set up The Badlands again, for good
; measure
249C-   A0 20        LDY   #$20
249E-   B9 81 07     LDA   $0781,Y
24A1-   99 00 02     STA   $0200,Y
24A4-   88           DEY
24A5-   10 F7        BPL   $249E

; not sure what these are yet
24A7-   A9 B4        LDA   #$B4
24A9-   85 F7        STA   $F7
24AB-   A0 18        LDY   #$18
24AD-   84 FD        STY   $FD
24AF-   86 2B        STX   $2B
24B1-   A9 0C        LDA   #$0C
24B3-   85 F8        STA   $F8
24B5-   8D 81 C0     STA   $C081

; set unfriendly reset vector again
24B8-   A9 02        LDA   #$02
24BA-   8D F3 03     STA   $03F3
24BD-   A9 00        LDA   #$00
24BF-   8D F2 03     STA   $03F2
24C2-   A9 A7        LDA   #$A7
24C4-   8D F4 03     STA   $03F4
24C7-   A9 17        LDA   #$17
24C9-   48           PHA             ; hmm
```

```
; main loop
24CA-   AD 5A 05    LDA   $055A
24CD-   18          CLC
24CE-   69 02       ADC   #$02
24D0-   20 00 04    JSR   $0400
24D3-   20 00 05    JSR   $0500
24D6-   C6 F8       DEC   $F8
24D8-   D0 F0       BNE   $24CA
```

By the looks of it, $0400 moves the drive head to the phase given in the accumulator on the way in. A track is two phases, so this is just reading whole tracks. No tricks, nothing fancy. Zero page $FC is the track counter (initialized with $0C at $04B1, then decremented at $04D6).

$0500 is the actual read routine.

```
*2500L

2500-   A9 0C       LDA   #$0C
2502-   85 FF       STA   $FF
2504-   BD 89 C0    LDA   $C089,X
```

```
; five-nibble prologue "FE 9F 97 EE DA"
2507-   A0 00        LDY   #$00
2509-   BD 8C C0     LDA   $C08C,X
250C-   10 FB        BPL   $2509
250E-   C9 FE        CMP   #$FE
2510-   D0 EE        BNE   $2500
2512-   BD 8C C0     LDA   $C08C,X
2515-   10 FB        BPL   $2512
2517-   49 9F        EOR   #$9F
2519-   D0 E5        BNE   $2500
251B-   BD 8C C0     LDA   $C08C,X
251E-   10 FB        BPL   $251B
2520-   C9 97        CMP   #$97
2522-   D0 DC        BNE   $2500
2524-   BD 8C C0     LDA   $C08C,X
2527-   10 FB        BPL   $2524
2529-   C9 EE        CMP   #$EE
252B-   D0 D3        BNE   $2500
252D-   BD 8C C0     LDA   $C08C,X
2530-   10 FB        BPL   $252D
2532-   C9 DA        CMP   #$DA
2534-   D0 CA        BNE   $2500

; main loop
; decode 4-4 encoded nibbles into ($FD)
2536-   BD 8C C0     LDA   $C08C,X
2539-   10 FB        BPL   $2536
253B-   2A           ROL
253C-   85 F6        STA   $F6
253E-   BD 8C C0     LDA   $C08C,X
2541-   10 FB        BPL   $253E
2543-   25 F6        AND   $F6
2545-   91 FD        STA   ($FD),Y
2547-   C8           INY
2548-   D0 EC        BNE   $2536

; one byte delimiter
254A-   BD 8C C0     LDA   $C08C,X
254D-   10 FB        BPL   $254A
254F-   38           SEC
2550-   E6 FE        INC   $FE
```

```
; decrement page count (initialized at
; $0500 with value $0C)
2552-   C6 FF        DEC   $FF
2554-   D0 E0        BNE   $2536
2556-   60           RTS
```

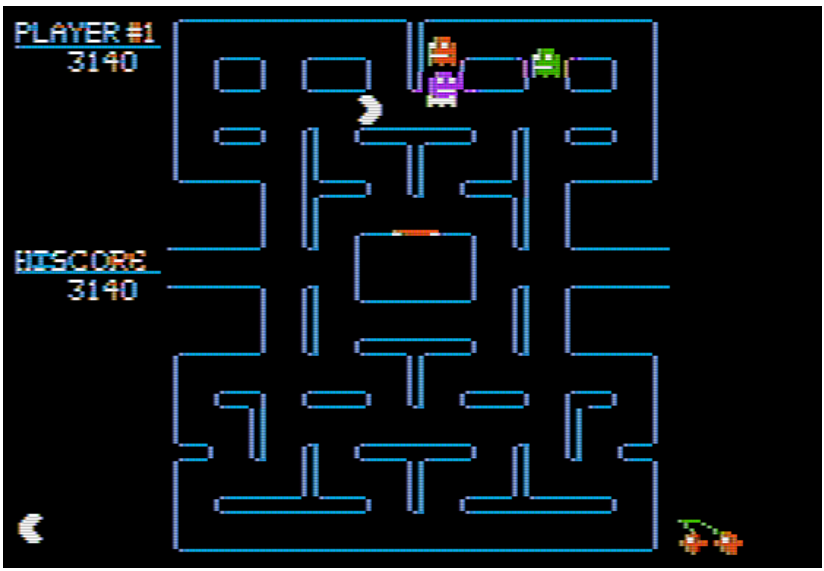
So it reads an entire track at once into ($FD), which starts at $0F18 and increments monotonically. The data itself is 4-4 encoded and starts immediately after a 5-nibble prologue. Each page worth of data is delimited by a single nibble, which is ignored.

(This explains why the original disk seems to take longer to read some tracks than others. It really does take longer, because in the worst case, it needs to wait an entire disk revolution to find the track prologue start reading data.)

There does not appear to be any nibble check or copy protection beyond the custom track structure. All the data is stored on whole tracks, which explains why my EDD bit copy worked.

```
Backtracking, I left off at $24DA...

*24DAL

; turn off drive motor
24DA-   BD 88 C0     LDA   $C088,X
24DD-   8D 81 C0     STA   $C081

; and exit
24E0-   60           RTS
```

But we've pushed bytes to the stack -- $0F (at $0486) and $17 (at $04C9) -- so this "RTS" will pop those off and jump to the start of the game at $0F18.



Chapter 4
In Which We Capture The Flag And Declare Victory

Now I can capture the entire game in memory.

```
*9600<C600.C6FFM

; set up first callback (same as
; previous trace)
96F8-   08          PHP
96F9-   48          PHA
96FA-   A9 09       LDA   #$09
96FC-   8D B4 08    STA   $08B4
96FF-   A9 97       LDA   #$97
9701-   8D B5 08    STA   $08B5
9704-   68          PLA
9705-   28          PLP

; start the boot
9706-   4C 01 08    JMP   $0801

; first callback is here -- set up the
; second callback (again, same as
; previous trace)
9709-   08          PHP
970A-   48          PHA
970B-   A9 1A       LDA   #$1A
970D-   8D 6D 07    STA   $076D
9710-   A9 97       LDA   #$97
9712-   8D 6E 07    STA   $076E
9715-   68          PLA
9716-   28          PLP

; continue the boot
9717-   4C 05 07    JMP   $0705
```

```
; second callback is here -- change the
; bytes pushed to the stack so we break
; unconditionally to the monitor
; instead of starting the game
971A-   A9 FF        LDA   #$FF
971C-   8D 85 04     STA   $0485
971F-   A9 58        LDA   #$58
9721-   8D C8 04     STA   $04C8

; continue the boot and load the game
; into memory
9724-   4C 84 04     JMP   $0484

*BSAVE TRACE3,A$9600,L$127
*9600G
...reboots slot 6...
...loads...
<beep>
```

To test my assumption that the game never touches the original disk once it's in memory, I removed the disk from the drive and did

```
*F18G
```

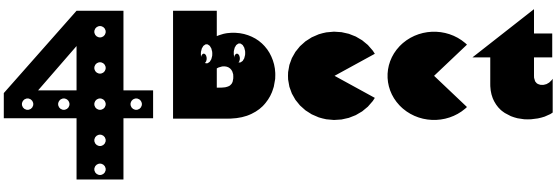
and the game started right up.

After re-running that trace a few times and carefully relocating chunks of memory, and I have the entire game in three files on my work disk:

```
]CATALOG,S5,D1
.
.
.
 B 018 PM.OBJ 0F18-1FFF
 B 066 PM.OBJ 2000-5FFF
 B 065 PM.OBJ 6000-9F17
```

(The disk loads exactly $9000 bytes -- $0C tracks of $0C pages.)

Chapter 5
Introducing 4boot



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.

I've been working on my own bootloader for a while, so this seems like a good excuse to use it.

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

]PR#5
]BLOAD PM.OBJ 0F18-1FFF,A$F18
]BLOAD PM.OBJ 2000-5FFF,A$2000
]BLOAD PM.OBJ 6000-9F17,A$6000
]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

; logical to physical sector mapping
*340.34F

0340- 00 07 0E 06 0D 05 0C 04
0348- 0B 03 0A 02 09 01 08 0F

; RWTS parameter table, pre-initialized
; with slot 6, drive 1, track $01,
; sector $00, address $0F18, and RWTS
; write command ($02)
*388.397

0388- 01 60 01 00 01 00 FB F7
0390- 18 0F 00 00 02 00 00 60
```

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

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

The bootloader (which I've named 4boot) lives on track $00. T00S00 is boot0, which reuses the disk controller ROM routine to load boot1, which lives on sectors $0C-$0E.

Boot0 looks like this:

```
; decrement sector count
0801-   CE 19 08    DEC   $0819

; branch once we've read enough sectors
0804-   30 12       BMI   $0818

; increment physical sector to read
0806-   E6 3D       INC   $3D

; set page to save sector data
0808-   A9 BF       LDA   #$BF
080A-   85 27       STA   $27

; decrement page
080C-   CE 09 08    DEC   $0809

; $0880 is a sparse table of $C1..$C6,
; so this sets up the proper jump to
; the disk controller ROM based on the
; slot number
080F-   BD 80 08    LDA   $0880,X
0812-   8D 17 08    STA   $0817

; read a sector (exits via $0801)
0815-   4C 5C 00    JMP   $005C
```

```
; sector read loop exits to here (from
; $0804) -- note: by the time execution
; reaches here, $0819 is $FF, so this
; just resets the stack
0818-   A2 03       LDX   #$03
081A-   9A          TXS

; set up zero page (used by RWTS) and
; push an array of addresses to the
; stack at the same time
081B-   A2 0F       LDX   #$0F
081D-   BD 80 08    LDA   $0880,X
0820-   95 F0       STA   $F0,X
0822-   48         PHA
0823-   CA          DEX
0824-   D0 F7       BNE   $081D
0826-   60          RTS

*881.88F

0880-    88 FE 92 FE 7B BE FF
0888- BC 17 0F 0F 09 00 18 00
```

These are pushed to the stack in reverse order, starting with $088F. When we hit the "RTS" at $0826, it pops the stack and jumps to $FE89, then $FE93, then $BE7C, then $BD00, then $0F18.

- $FE89 and $FE93 are in ROM (PR#0 and IN#0).
- $BE7C was loaded from sector $0D. It just switches to hi-res graphics page 1.
- $BD00 is the RWTS entry point. It loads T01-T09 into memory,starting at $0F18. (These values are stored in zero page, which we just set.)

[...]

- $0F18 is the game entry point. It never returns, so the other values on the stack are irrelevant.

The RWTS at $BD00 is derived from the ProDOS RWTS. It uses in-place nibble decoding to avoid extra memory copying, and it uses "scatter reads" to read whatever sector is under the drive head when it's ready to load something.

```
*BD00L

; set up some places later in the RWTS
; where we need to read from a slot-
; specific data latch
BD00-   A6 2B       LDX   $2B
BD02-   8A          TXA
BD03-   09 8C       ORA   #$8C
BD05-   8D 96 BD    STA   $BD96
BD08-   8D AD BD    STA   $BDAD
BD0B-   8D C3 BD    STA   $BDC3
BD0E-   8D D7 BD    STA   $BDD7
BD11-   8D EC BD    STA   $BDEC

; advance drive head to next track
BD14-   20 53 BE    JSR   $BE53

; sectors-left-to-read-on-this-track
; counter
BD17-   A0 0F       LDY   #$0F
BD19-   84 F8       STY   $F8
```

```
; Initialize array at $0100 that tracks
; which sectors we've read from the
; current track. The array is in
; physical sector order, thus the RWTS
; assumes data is stored in physical
; sector order on each track. Values
; are the actual pages in memory where
; that sector should go, and they get
; zeroed once the sector is read.
BD1B-   98          TYA
BD1C-   18          CLC
BD1D-   65 FB       ADC   $FB
BD1F-   99 00 01    STA   $0100,Y
BD22-   88          DEY
BD23-   10 F6       BPL   $BD1B

; find the next address prologue and
; store the address field in $2C..$2F,
; like DOS 3.3
BD25-   20 0F BE    JSR   $BE0F

; check if this sector has been read
BD28-   A4 2D       LDY   $2D
BD2A-   B9 00 01    LDA   $0100,Y

; if 0, we've read this sector already,
; so loop back and look for another
BD2D-   F0 F6       BEQ   $BD25
```

```
; if not 0, use the target page and set
; up some STA instructions in the RWTS
; so we write this sector directly to
; its intended page in memory
BD2F-   A8          TAY
BD30-   84 FF       STY   $FF
BD32-   8C EA BD    STY   $BDEA
BD35-   A5 FE       LDA   $FE
BD37-   8D E9 BD    STA   $BDE9
BD3A-   38          SEC
BD3B-   E9 54       SBC   #$54
BD3D-   8D D1 BD    STA   $BDD1
BD40-   B0 02       BCS   $BD44
BD42-   88          DEY
BD43-   38          SEC
BD44-   8C D2 BD    STY   $BDD2
BD47-   E9 57       SBC   #$57
BD49-   8D AA BD    STA   $BDAA
BD4C-   B0 01       BCS   $BD4F
BD4E-   88          DEY
BD4F-   8C AB BD    STY   $BDAB

; read the sector into memory
BD52-   20 6D BD    JSR   $BD6D

; if that failed, just loop back and
; look for another sector
BD55-   B0 CE       BCS   $BD25

; mark this sector as read
BD57-   A4 2D       LDY   $2D
BD59-   A9 00       LDA   #$00
BD5B-   99 00 01    STA   $0100,Y
BD5E-   E6 FB       INC   $FB

; decrement sectors-left-to-read-on-
; this-track counter
BD60-   C6 F8       DEC   $F8
```

```
; loop until we've read all the sectors
; on this track
BD62-   10 C1        BPL   $BD25

; decrement tracks-left-to-read counter
; (set in boot0)
BD64-   C6 FC        DEC   $FC

; loop until we've read all the tracks
BD66-   D0 AC        BNE   $BD14

; turn off drive motor and exit
BD68-   BD 88 C0     LDA   $C088,X
BD6B-   38           SEC
BD6C-   60           RTS
```

Quod erat liberandum.

A 4am crack No. 257

EOF