Pinball Construction Set

2017-03-01

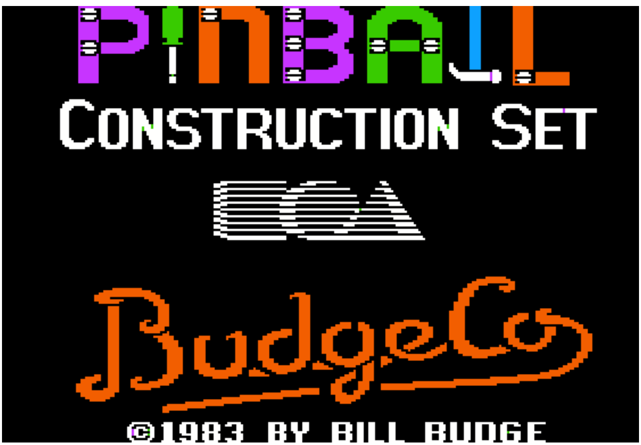
4am

to deprotect and preserve

Contents

```
--------Pinball Construction Set-------
A 4am and san inc crack     2017-03-01
---------------------------------------
```

Name: Pinball Construction Set
Genre: arcade
Year: 1983
Authors: Bill Budge
Publisher: Electronic Arts
Platform: Apple][+ or later
Media: single-sided 5.25-inch floppy
OS: custom



Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

COPYA
read error after a few tracks

Locksmith Fast Disk Backup
reads everything except track $06

EDD 4 bit copy (no sync, no count)
no errors, but copy boots to title
screen then displays graphical
"Insert PCS disk" message

```
Copy ][+ nibble editor
   track $06 seems reasonably structured
   until you look very, very closely

                 --v--

TRACK: 06  START: 1800  LENGTH: 3DFF

19A8: FF FF FF FF FF FF FF FF   VIEW
19B0: FF FF FF FF FF FF FF FF
19B8: FF FF FF FF FF FF FF FF
19C0: FF FF FF FF FF FF FF FF
19C8: FF D5 AA 96 FF FE AA AF  <-19C9
         ^^^^^^^^ ^^^^^ ^^^^^
         prologue V=255 T=$05

19D0: AA AA FF FB DE AA EB FF
      ^^^^^ ^^^^^ ^^^^^^^^
      S=$00 chksm epilogue

19D8: FF FF FF FF FF D5 AA AD
19E0: B5 B5 B5 B5 B5 B5 B5 B5
19E8: B5 B5 B5 B5 B5 B5 B5 B5

                 --^--
```

Did you see it? Track 6 is presenting
itself as track 5! The address field
is lying to us! Bad disk! No lying!

Disk Fixer
  no way to ignore the track number,
  so no way to read track 6

Copy][+ (8.4) sector editor
  Since this ignores the track number
  by default, it can read track 6
  without issue. It appears to contain
  the same data as track 5.

Why didn't COPYA work?
  intentionally corrupted prologue on
  track 6

Why didn't Locksmith FDB work?
  ditto

Why didn't my EDD copy work?
  I don't know. A bit copy preserves
  the address prologue, even if it's
  corrupted. So there must be something
  else. It's definitely executing some
  kind of run-time protection check,
  given the prettiness of the error. No
  disk grinding, no crashes -- just a
  graphical error message on top of the
  title screen.

Next steps:

  1. Trace the boot
  2. Find and disable the runtime check
  3. Declare victory (*)

(*) go to the gym

Chapter 1
In Which We Brag About Our Humble Beginnings

I have two floppy drives, one in slot 6 and the other in slot 5. My "work disk" (in slot 5) runs Diversi-DOS 64K, which is compatible with Apple DOS 3.3 but relocates most of DOS to the language card on boot. This frees up most of main memory (only using a single page at $BF00..$BFFF), which is useful for loading large files or examining code that lives in areas typically reserved for DOS.

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

The floppy drive firmware code at $C600 is responsible for aligning the drive head and reading sector 0 of track 0 into main memory at $0800. Because the drive can be connected to any slot, the firmware code can't assume it's loaded at $C600. If the floppy drive card were removed from slot 6 and reinstalled in slot 5, the firmware code would load at $C500 instead.

To accommodate this, the firmware does some fancy stack manipulation to detect where it is in memory (which is a neat trick, since the 6502 program counter is not generally accessible). However, due to space constraints, the detection code only cares about the lower 4 bits of the high byte of its own address.

$C600 (or $C500, or anywhere in $Cx00) is read-only memory. I can't change it, which means I can't stop it from transferring control to the boot sector of the disk once it's in memory. BUT! The disk firmware code works unmodified at any address. Any address that ends with $x600 will boot slot 6, including $B600, $A600, $9600, &c.

```
; copy drive firmware to $9600
*9600<C600.C6FFM

; and execute it
*9600G
...reboots slot 6...
```

Now then:

```
]PR#5
...
]CALL -151

*9600<C600.C6FFM

*96F8L

96F8-   4C 01 08    JMP   $0801
```

That's where the disk controller ROM code ends and the on-disk code begins. But $9600 is part of read/write memory. I can change it at will. So I can interrupt the boot process after the drive firmware loads the boot sector from the disk but before it transfers control to the disk's bootloader.

```
; instead of jumping to on-disk code,
; copy boot sector to higher memory so
; it survives a reboot
96F8-   A0 00       LDY   #$00
96FA-   B9 00 08    LDA   $0800,Y
96FD-   99 00 28    STA   $2800,Y
9700-   C8          INY
9701-   D0 F7       BNE   $96FA

; turn off slot 6 drive motor
9703-   AD E8 C0    LDA   $C0E8

; reboot to my work disk in slot 5
9706-   4C 00 C5    JMP   $C500

*BSAVE TRACE0,A$9600,L$109
*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE OBJ.0800-08FF,A$2800,L$100
```

Now let's see how this disk boots.

Chapter 2
In Which Every Exit Is An Entrance Somewhere Else

```
]CALL -151

; move bootloader back to its original
; location
*800<2800.28FFM

*801L

0801-   A5 27       LDA   $27
0803-   C9 09       CMP   #$09
0805-   D0 18       BNE   $081F
0807-   A5 2B       LDA   $2B
0809-   4A          LSR
080A-   4A          LSR
080B-   4A          LSR
080C-   4A          LSR
080D-   09 C0       ORA   #$C0
080F-   85 3F       STA   $3F
0811-   A9 5C       LDA   #$5C
0813-   85 3E       STA   $3E
0815-   18          CLC
0816-   AD FE 08    LDA   $08FE
0819-   6D FF 08    ADC   $08FF
081C-   8D FE 08    STA   $08FE
081F-   AE FF 08    LDX   $08FF
0822-   30 15       BMI   $0839
0824-   BD 4D 08    LDA   $084D,X
0827-   85 3D       STA   $3D
0829-   CE FF 08    DEC   $08FF
082C-   AD FE 08    LDA   $08FE
082F-   85 27       STA   $27
0831-   CE FE 08    DEC   $08FE
0834-   A6 2B       LDX   $2B
0836-   6C 3E 00    JMP   ($003E)
0839-   EE FE 08    INC   $08FE
083C-   EE FE 08    INC   $08FE
083F-   20 89 FE    JSR   $FE89
0842-   20 93 FE    JSR   $FE93
0845-   20 2F FB    JSR   $FB2F
0848-   A6 2B       LDX   $2B
084A-   6C FD 08    JMP   ($08FD)
```

This is identical to a standard DOS 3.3 disk. In fact, it's identical to the DOS 3.3 System Master. It reuses the drive firmware, specifically an entry point at $Cn5C (n = boot slot), to read more sectors from track 0. The lowest address is at $08FE, and the number of sectors is at $08FF (off by 1, due to how the loop is constructed).

```
*08FE.08FF

08FE- B6 09
```

OK, still identical to DOS 3.3. It will load $0A sectors into $B600..$BFFF and jump to $B700 via the indirect jump to ($08FD). I can interrupt the boot at $084A before that happens, then examine the next stage of the bootloader.

```
; copy the drive firmware again
*9600<C600.C6FFM

; set up a callback at $084A, so it
; jumps back to code under my control
; instead of continuing to the next
; stage of the bootloader
96F8-    A9 4C       LDA   #$4C
96FA-    8D 4A 08    STA   $084A
96FD-    A9 0A       LDA   #$0A
96FF-    8D 4B 08    STA   $084B
9702-    A9 97       LDA   #$97
9704-    8D 4C 08    STA   $084C

; start the boot
9707-    4C 01 08    JMP   $0801
```

```
; (callback is here)
; copy the sectors we read at $B600+
; to lower memory so they survive a
; reboot
970A-   A2 0A       LDX   #$0A
970C-   A0 00       LDY   #$00
970E-   B9 00 B6    LDA   $B600,Y
9711-   99 00 26    STA   $2600,Y
9714-   C8          INY
9715-   D0 F7       BNE   $970E
9717-   EE 10 97    INC   $9710
971A-   EE 13 97    INC   $9713
971D-   CA          DEX
971E-   D0 EE       BNE   $970E

; turn off the slot 6 drive motor and
; reboot to my work disk
9720-   AD E8 C0    LDA   $C0E8
9723-   4C 00 C5    JMP   $C500

*BSAVE TRACE2,A$9600,L$126

*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE OBJ.B600-BFFF,A$2600,L$A00
]CALL -151

*FE89G FE93G      ; disconnect DOS
*B600<2600.2FFFM ; move bootloader
```

Now we can see the next stage of the bootloader, which starts at $B700.

```
*B700L

; setting up a standard RWTS parameter
; table
B700-   8E E9 B7    STX   $B7E9
B703-   8E F7 B7    STX   $B7F7
B706-   8E C3 B7    STX   $B7C3
B709-   A9 01       LDA   #$01
B70B-   8D EA B7    STA   $B7EA
B70E-   8D F8 B7    STA   $B7F8
B711-   8A          TXA
B712-   4A          LSR
B713-   4A          LSR
B714-   4A          LSR
B715-   4A          LSR
B716-   AA          TAX

; not sure why we're clearing out some
; slot-indexed locations in the input
; buffer ($0200-$02FF)
B717-   A9 00       LDA   #$00
B719-   9D D8 02    STA   $02D8,X
B71C-   9D D0 02    STA   $02D0,X

B71F-   A2 03       LDX   #$03
B721-   86 07       STX   $07
B723-   20 72 B7    JSR   $B772
```

```
*B772L

B772-   BD 62 B7    LDA   $B762,X
B775-   8D C2 B7    STA   $B7C2
B778-   BC 6E B7    LDY   $B76E,X
B77B-   BD 66 B7    LDA   $B766,X
B77E-   48          PHA
B77F-   BD 6A B7    LDA   $B76A,X
B782-   AA          TAX
B783-   68          PLA
B784-   8D EC B7    STA   $B7EC
B787-   8E ED B7    STX   $B7ED
B78A-   8C F1 B7    STY   $B7F1
B78D-   A9 01       LDA   #$01
B78F-   8D F4 B7    STA   $B7F4
B792-   A9 B7       LDA   #$B7
B794-   A0 E8       LDY   #$E8
B796-   20 B5 B7    JSR   $B7B5
B799-   B0 F7       BCS   $B792
B79B-   AC ED B7    LDY   $B7ED
B79E-   88          DEY
B79F-   10 05       BPL   $B7A6
B7A1-   A0 0F       LDY   #$0F
B7A3-   CE EC B7    DEC   $B7EC
B7A6-   8C ED B7    STY   $B7ED
B7A9-   CE F1 B7    DEC   $B7F1
B7AC-   CE C2 B7    DEC   $B7C2
B7AF-   D0 E1       BNE   $B792
B7B1-   60          RTS
```

OK, that's a multi-sector read routine.

- $B762,X ends up in $B7C2, which is decremented (at $B7AC) and eventually ends the loop. So that's the number of sectors to read.

- $B76E,X ends up in $B7F1. That's the target address (high byte).

- $B766,X ends up in $B7EC. That's the starting track. Tracks are read in decreasing order (decremented at $B7A3).

- $B76A,X ends up in $B7ED. That's the starting sector. Sectors are read in decreasing order (decremented at $B79E).

- $B7B5 is the standard high-level entry point to read a single sector with a DOS 3.3-shaped RWTS. I checked the code; it's identical to an unprotected DOS 3.3 disk.

So what are we reading? Well, X was 3 on entry (set at $B71F), so...

```
*B765

B765- 0D

*B771

B771- B5

*B769

B769- 01

*B76D

B76D- 0C
```

So we're reading #$0D sectors, starting at T01,S0C into $B500 and decrementing. All told, T01,S00-S0C end up in $A900..$B5FF.

Continuing from $B726...

```
B726-   A6 07        LDX   $07
B728-   CA           DEX
B729-   10 F6        BPL   $B721
```

Ah! So this is a loop, and we'll end up accessing each of the items in each of those arrays. (Remember that the track, sector, and address all decrement.)

Looking through the rest of the arrays at $B762, $B76E, $B766, and $B76A, I constructed this chart of what is being read and where it ends up. (Remember, everything decrements, so that address is the highest page of each range.)

X	count	address	track	sector
3	$0D	$B500	$01	$0C
2	$01	$4000	$03	$08
1	$18	$1F00	$03	$07
0	$20	$3F00	$22	$0F

That last read (X=0) will fill hi-res page 1 with what I can only assume is the title screen.

Continuing from $B72B...

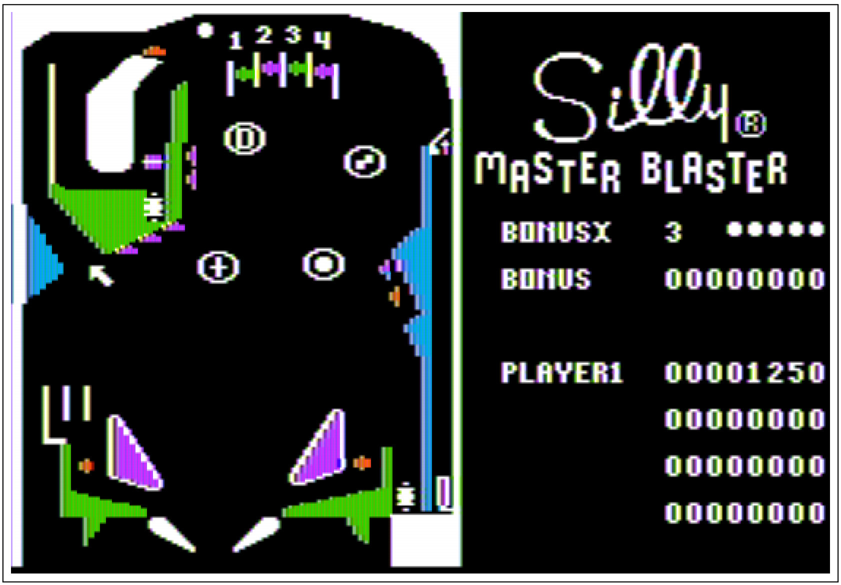
```
; immediately display hi-res page 1
; (must be the title screen)
B72B-   8D 50 C0    STA   $C050
B72E-   8D 52 C0    STA   $C052
B731-   8D 54 C0    STA   $C054
B734-   8D 57 C0    STA   $C057

; wait for a while, or until the user
; hits a key or presses either joystick
; button
B737-   8D 10 C0    STA   $C010
B73A-   A9 14       LDA   #$14
B73C-   85 07       STA   $07
B73E-   AD 00 C0    LDA   $C000
B741-   0D 61 C0    ORA   $C061
B744-   0D 62 C0    ORA   $C062
B747-   30 09       BMI   $B752
B749-   A9 00       LDA   #$00
B74B-   20 A8 FC    JSR   $FCA8
B74E-   C6 07       DEC   $07
B750-   10 EC       BPL   $B73E

; reset stack pointer
B752-   A2 00       LDX   #$00
B754-   9A          TXS

; check machine ID
B755-   AD C0 FB    LDA   $FBC0
B758-   F0 6A       BEQ   $B7C4
B75A-   4C DF B7    JMP   $B7DF
```

I will ignore the machine ID check for the moment. (It checks whether you're running on an Apple //c or later, which I'm not.)

Continuing from $B7DF...

```
*B7DFL

B7DF-   20 FC 1E     JSR   $1EFC
B7E2-   4C 00 1E     JMP   $1E00
```

That's part of the code we loaded in one of the calls to the multi-sector read routine (when X=1). Now I get to interrupt the boot once again and see what ends up at $1EFC and $1E00.



Chapter 3
Two Holes Are Better Than One; Any Mouse Will Tell You That

```
; reboot because I destroyed DOS
*C500G
...

]CALL -151

*9600<C600.C6FFM

; set up callback #1 (same as previous
; trace)
96F8-   A9 4C       LDA   #$4C
96FA-   8D 4A 08    STA   $084A
96FD-   A9 0A       LDA   #$0A
96FF-   8D 4B 08    STA   $084B
9702-   A9 97       LDA   #$97
9704-   8D 4C 08    STA   $084C

; start the boot
9707-   4C 01 08    JMP   $0801

; (callback #1 is here)
; break to the monitor after all the
; sector reads are complete
970A-   A9 4C       LDA   #$4C
970C-   8D DF B7    STA   $B7DF
970F-   A9 59       LDA   #$59
9711-   8D E0 B7    STA   $B7E0
9714-   A9 FF       LDA   #$FF
9716-   8D E1 B7    STA   $B7E1

; continue the boot
9719-   4C 00 B7    JMP   $B700
```

```
*BSAVE TRACE3,A$9600,L$11C

*9600G
...reboots slot 6...
...read read read...
...briefly displays title screen...
<beep>

*1EFCL

1EFC-   AD FE 1F      LDA   $1FFE
1EFF-   D0 6C         BNE   $1F6D
1F01-   85 07         STA   $07
1F03-   A9 C8         LDA   #$C8
1F05-   85 08         STA   $08
1F07-   A2 07         LDX   #$07
1F09-   C6 08         DEC   $08
1F0B-   CA            DEX
1F0C-   30 5F         BMI   $1F6D
1F0E-   A0 0C         LDY   #$0C
1F10-   B1 07         LDA   ($07),Y
1F12-   C9 20         CMP   #$20
1F14-   D0 F3         BNE   $1F09
1F16-   A0 FB         LDY   #$FB
1F18-   B1 07         LDA   ($07),Y
1F1A-   C9 D6         CMP   #$D6
1F1C-   D0 EB         BNE   $1F09
.
.
.
```

I won't list the rest of this routine, but I'll tell you what it's doing: it's scanning the peripheral ROM space, checking for the presence of a mouse card. The magic identification bytes are at offset $0C and offset $FB, as documented in an obscure technote from decades ago:

"""
The AppleMouse II card is identified by a value of $20 at $Cn0C ("X-Y Pointing device, type zero") and a value of $D6 at $CnFB, where n is the slot number.
"""
      -Mouse Technical Note #5, 11/1990

So, not copy protection-related. But it does explain why the machine ID check would skip this routine on later machines, which have a native interface for the mouse and don't need a separate peripheral card.

The next jump is to $1E00:

```
*1E00L

1E00-   A2 00       LDX   #$00
1E02-   20 36 1E    JSR   $1E36

*1E36L

1E36-   86 08       STX   $08
1E38-   AD C3 B7    LDA   $B7C3
```

```
; more RWTS parameters, possibly
; setting up another disk read?
1E3B-   8D E9 B7     STA   $B7E9
1E3E-   A2 01        LDX   #$01
1E40-   8E EA B7     STX   $B7EA
1E43-   CA           DEX
1E44-   8E F0 B7     STX   $B7F0
1E47-   8E EB B7     STX   $B7EB
1E4A-   20 7D 1E     JSR   $1E7D

*1E7DL

1E7D-   A2 05        LDX   #$05
1E7F-   20 4F 1E     JSR   $1E4F

*1E4FL

1E4F-   BD 65 1E     LDA   $1E65,X
1E52-   8D C2 B7     STA   $B7C2
1E55-   BD 71 1E     LDA   $1E71,X
1E58-   85 07        STA   $07
1E5A-   BD 6B 1E     LDA   $1E6B,X
1E5D-   BC 77 1E     LDY   $1E77,X
1E60-   A6 07        LDX   $07
1E62-   4C 84 B7     JMP   $B784
```

More disk reading, but taking values from local arrays (at $1E65+, $1E71+, $1E6B+, and $1E77+) and jumping into the middle of the routine we examined above. X=5 on entry (set at $1E7D), so we end up reading 3 sectors into $0B00 (decrementing). I'll keep an eye on whether we jump into that range, but for now, let's continue.

```
Continuing from $1E82...

1E82-   20 A6 1A    JSR   $1AA6

*1AA6L

1AA6-   18          CLC
1AA7-   A2 A6       LDX   #$A6
1AA9-   A0 00       LDY   #$00
1AAB-   20 E9 1D    JSR   $1DE9
1AAE-   20 ED 1D    JSR   $1DED
1AB1-   A0 DF       LDY   #$DF
1AB3-   20 EB 1D    JSR   $1DEB
1AB6-   A9 1E       LDA   #$1E
1AB8-   A0 00       LDY   #$00
1ABA-   20 E9 1D    JSR   $1DE9

*1DE9L

1DE9-   86 01       STX   $01
1DEB-   84 00       STY   $00
1DED-   71 00       ADC   ($00),Y
1DEF-   88          DEY
1DF0-   D0 FB       BNE   $1DED
1DF2-   E6 01       INC   $01
1DF4-   60          RTS
```

Nice. This is a very compact checksum routine. It adds all the values in several pages of memory -- starting at $A600, as set by X and Y at $1AA7, and also the page at $1E00..$1EFF. Note that $1E00 is part of the code path that led us to this point.

EXCEPT!

That's not actually what it does. Note the instruction at $1AB6 -- an "LDA", not an "LDX". I'm almost certain this was a mistake. If that were an "LDX", it would extend the anti-tamper check to $1E00..$1EFF. But instead, it overwrites the checksum value and ends up checksumming $A600..$A6FF (again, but with a starting value of #$1E).

```
; compare the final checksum
1ABD-   C9 9A         CMP   #$9A

; branch on success
1ABF-   F0 05         BEQ   $1AC6

; on failure, pop the stack so we
; forget the fact that we were called
; as a subroutine, then continue to
; $1E92
1AC1-   68            PLA
1AC2-   68            PLA
1AC3-   4C 92 1E      JMP   $1E92
```

Hmm. I wonder...

```
*C050 C052 C054 C057 N 1E92G
```

...displays "Insert PCS disk" message over the title screen...

Someone went out of their way to ensure that no one tampers with this code.

Who would do such a thing?

Chapter 4
In Which We Do Such A Thing

Retracing my steps and continuing from $1E85...

```
*1E85L

; more disk reading, this time just 1
; sector (T05,S00) into $A500
1E85-   A2 04       LDX   #$04
1E87-   20 4F 1E    JSR   $1E4F

; and call $A600, which -- perhaps not
; coincidentally -- was part of the
; anti-tamper check at $1AA6
1E8A-   20 00 A6    JSR   $A600
```

At this point, it would be wise to save this code in memory -- especially since someone has gone to so much trouble to prevent us from doing exactly that.

```
*C500G
...

]CALL -151

*9600<C600.C6FFM

; set up callback #1
96F8-   A9 4C       LDA   #$4C
96FA-   8D 4A 08    STA   $084A
96FD-   A9 0A       LDA   #$0A
96FF-   8D 4B 08    STA   $084B
9702-   A9 97       LDA   #$97
9704-   8D 4C 08    STA   $084C

; start the boot
9707-   4C 01 08    JMP   $0801
```

```
; (callback #1 is here)
; set up callback #2
970A-   A9 4C       LDA   #$4C
970C-   8D E2 B7    STA   $B7E2
970F-   A9 1C       LDA   #$1C
9711-   8D E3 B7    STA   $B7E3
9714-   A9 97       LDA   #$97
9716-   8D E4 B7    STA   $B7E4

; continue the boot
9719-   4C 00 B7    JMP   $B700

; (callback #2 is here)
; set up unconditional break at $1E8A
; instead of calling $A600
971C-   A9 4C       LDA   #$4C
971E-   8D 8A 1E    STA   $1E8A
9721-   A9 59       LDA   #$59
9723-   8D 8B 1E    STA   $1E8B
9726-   A9 FF       LDA   #$FF
9728-   8D 8C 1E    STA   $1E8C

; continue the boot
972B-   4C 00 1E    JMP   $1E00

*BSAVE TRACE4,A$9600,L$12E

*9600G
...reboots slot 6...
...read read read...
...displays title screen...
...waits...
...reads some more...
<beep>

Success!
```

```
*2500<A500.B5FFM
*C500G
...

]BSAVE OBJ.A500-B5FF,A$2500,L$1100
]CALL -151

*FE89G FE93G      ; disconnect DOS again
*A500<2500.35FFM ; move code into place
```

Now let's see what's at $A600.

```
*A600L

A600-   4C 69 A6    JMP   $A669

*A669L

A669-   8D E4 A8    STA   $A8E4
A66C-   20 E6 A7    JSR   $A7E6
A66F-   20 B9 A7    JSR   $A7B9
A672-   4C 69 A6    JMP   $A669
```

Hmm, a loop. $A672 jumps back to $A669.

```
*A7E6L

; swap part of zero page (starting at
; location $50) with nearby storage at
; $A8E5
A7E6-   A2 0F        LDX   #$0F
A7E8-   B5 50        LDA   $50,X
A7EA-   48           PHA
A7EB-   BD E5 A8     LDA   $A8E5,X
A7EE-   95 50        STA   $50,X
A7F0-   68           PLA
A7F1-   9D E5 A8     STA   $A8E5,X
A7F4-   CA           DEX
A7F5-   10 F1        BPL   $A7E8
A7F7-   AD E2 A8     LDA   $A8E2
A7FA-   60           RTS
```

Continuing from $A66F, which JSRs to $A7B9...

```
*A7B9L

; save X and Y on the stack
A7B9-   8A           TXA
A7BA-   48           PHA
A7BB-   98           TYA
A7BC-   48           PHA

A7BD-   20 38 A8     JSR   $A838

*A838L

; pull return address (-1) off the
; stack and put it in zero page $54/$55
A838-   68           PLA
A839-   85 54        STA   $54
A83B-   68           PLA
A83C-   85 55        STA   $55
```

```
; pull two more bytes off the stack --
; these are the Y and X registers that
; we pushed at $A7B9
A83E-   68          PLA
A83F-   85 52       STA   $52
A841-   68          PLA
A842-   85 53       STA   $53

A844-   E6 52       INC   $52

; increment $54/$55 as a 16-bit value
A846-   E6 54       INC   $54
A848-   D0 02       BNE   $A84C
A84A-   E6 55       INC   $55

; and jump there
A84C-   6C 54 00    JMP   ($0054)
```

So we end up jumping to... the next instruction after the JSR at $A7BD?!?

```
*A7C0L

; More stack manipulation! Pop the
; next two bytes off the stack and put
; them in $52/$53 (overwriting what we
; just put there at $A83E, but OK)
A7C0-   68          PLA
A7C1-   85 52       STA   $52
A7C3-   68          PLA
A7C4-   85 53       STA   $53

; take the byte at *that* address,
; offset by +4...
A7C6-   A0 04       LDY   #$04
A7C8-   B1 52       LDA   ($52),Y
A7CA-   C8          INY
A7CB-   D0 02       BNE   $A7CF
A7CD-   E6 53       INC   $53
```

```
; ...and use it as an index into an
; array at $A7D9...
A7CF-   AA              TAX
A7D0-   BD D9 A7        LDA   $A7D9,X

; ...which self-modifies a later
; instruction...
A7D3-   8D E0 A8        STA   $A8E0 --+
                                      |
; and jumps there                     |
A7D6-   4C DF A8        JMP   $A8DF   |
                                      |
*A8DFL                                |
                                      |
A8DF-   4C 00 A8        JMP   $A800 <-+
```

Here is the table that controls the lower byte of that "JMP" instruction at $A8DF:

```
*A7D9.

A7D9- .. 00 27 55 5F 6D 8A AD
A7E0- B9 A2 4F CE 26 79
```

(The array ends there. $A7E6 is real code; it's the start of the routine that swaps zero page with $A8E5 that we saw earlier.)

So this is a dispatch table. It's looking at the byte pointed to by ($52),Y -- which works out to $A675, but then location $52 is incremented (at $A7CD) -- and treating it as a kind of command. Different entry points in the $A6xx range are then dispatched to handle different commands.

Oh my God. This is an interpreter. The data starting at $A675 is bytecode -- the "compiled" form of an interpreted language. They invented their own programming language, then they wrote their copy protection in that language. And I get to reverse engineer it without source code or documentation.

Chapter 5
Mamas Don't Let Your Babies Grow Up To Be Bytecodes

My current working theory is that this is some kind of interpreted programming language. Since I have no documentation on the langauge, I'm just going to do the best I can. Slowly.

The main command dispatch is at $A7C8, which takes the byte at ($52),Y and looks up the low byte of an address in an array that starts at $A7D9. Given that $A7E6 is the start of real code, I posit there are 13 commands in this language, numbered $00 through $0C.

Based on the array at $A7D9, command $00 is dispatched to $A800:

```
*A800L

A800-   20 10 A8    JSR   $A810

*A810L

; Get the next byte from the bytecode
; buffer -- the same way we got the
; original command, but location $52
; was incremented earlier, so this will
; get the next byte after the command
; that sent us here.
A810-   B1 52       LDA   ($52),Y

; Munge that byte. (Don't know why.)
A812-   49 03       EOR   #$03

; Increment the pointer into the
; bytecode buffer.
A814-   C8          INY
A815-   D0 02       BNE   $A819
A817-   E6 53       INC   $53
```

```
; Store the (slightly munged) byte we
; just read.
A819-   85 54          STA   $54

; Get a second byte from the bytecode
; buffer.
A81B-   B1 52          LDA   ($52),Y

; Again, increment the pointer into the
; buffer.
A81D-   C8             INY
A81E-   D0 02          BNE   $A822
A820-   E6 53          INC   $53

; Munge this byte too, but differently.
; (Again, don't know why.)
A822-   49 D9          EOR   #$D9

; Store that munged byte into an
; adjacent location in zero page.
A824-   85 55          STA   $55
A826-   60             RTS
```

If ($52),Y always points to the next byte in the bytecode buffer, then this subroutine at $A810 swallows two bytes, decrypts them, and stores them in zero page $54 and $55. Like a function that takes two bytes as parameters.

Continuing from $A803...

```
; Take those two bytes we decrypted in
; $A810 and overwrite the pointer in
; $52/$53
A803-   A5 54          LDA   $54
A805-   85 52          STA   $52
A807-   A5 55          LDA   $55
A809-   85 53          STA   $53
```

```
; Reset the offset in the Y register.
A80B-   A0 00       LDY   #$00

; Jump to the main command dispatch.
A80D-   4C C8 A7    JMP   $A7C8
```

So command $00 is an absolute jump. Like the native "JMP" instruction, but to a new address inside the bytecode buffer instead. The new address is given in the two bytes after the command, in an encrypted form just because f--- you.

(Remember, unlike most languages, this one was purposely designed to be difficult to understand.)

-o-

Command $01 is dispatched to $A827:

```
*A827L

; Get the next two bytes from the
; bytecode buffer (same as command $00)
A827-   20 10 A8    JSR   $A810

; Save Y on the stack.
A82A-   98          TYA
A82B-   48          PHA

; Load location $56. (I don't know how
; this is set. Maybe one of the other
; commands?)
A82C-   A5 56       LDA   $56

; [see below]
A82E-   20 52 A8    JSR   $A852

; Store the return value (in the
; accumulator) back into location $56.
A831-   85 56       STA   $56

; Restore Y from the stack.
A833-   68          PLA
A834-   A8          TAY

; Jump to the main command dispatch.
A835-   4C C8 A7    JMP   $A7C8
```

So what's at $A852? Something so simple it's brilliant.

```
*A852L

A852-   6C 54 00    JMP   ($0054)
```

Command $01 is exactly like a native "JSR" instruction. It takes 2 bytes of command parameters, treats that as the address of native 6502 code, and JSRs to it. It loads the native accumulator with the value of zero page $56, and stores the native accumulator back into that same zero page address afterwards. So that's a way for subroutines to both take a value and return a value. (Now I'm positive that there are other bytecode commands to get and set zero page $56 directly.)

-o-

Command $02 is dispatched to $A855:

```
*A855L

; Get the next two bytes from the
; bytecode buffer and put them in $54/
; $55 (same as the other commands).
A855-   20 10 A8     JSR   $A810

; Look at that mysterious location $56.
A858-   A5 56        LDA   $56

; If it's not 0, branch to the middle
; of the dispatcher for command $00.
; This skips over the JSR at $A800 and
; continues with the virtual jump --
; in this case, to the address that we
; just got out of the bytecode buffer
; after this command (at $A855, via
; $A810).
A85A-   D0 A7        BNE   $A803

; Otherwise, fall through and jump to
; the main command dispatch.
A85C-   4C C8 A7     JMP   $A7C8
```

So command $02 functions like a "BNE" instruction, but it uses a bytecode address instead of a relative branch. If the value of zero page $56 is non-zero, it jumps to the bytecode address given in the 2-byte command parameters.

Location $56 is important, yo.

-o-

Command $03 is dispatched to $A85F:

```
*A85FL

; Get the next byte from the bytecode
; buffer.
A85F-   B1 52       LDA   ($52),Y

; Advance the bytecode buffer pointer.
A861-   C8          INY
A862-   D0 02       BNE   $A866
A864-   E6 53       INC   $53

; Decrypt this byte with yet another
; key.
A866-   49 4C       EOR   #$4C

; Store it in... $56!
A868-   85 56       STA   $56

; Jump to the main command dispatch.
A86A-   4C C8 A7    JMP   $A7C8
```

And now we see how to set location $56. This is like an "LDA" instruction with an immediate value. So location $56 functions as a kind of register, like the accumulator or X register in native 6502 assembly. Command $03 loads this register with a byte stored in the bytecode buffer (encrypted, of course, because f--- you).

-o-

Command $04 is dispatched to $A86D:

```
*A86DL

; Get the next two bytes from the
; bytecode buffer and put them in $54/
; $55 (same as the other commands).
A86D-   20 10 A8    JSR   $A810

; Load the value from that address.
A870-   A2 00       LDX   #$00
A872-   A1 54       LDA   ($54,X)

; Store it in the "register" at $56.
A874-   85 56       STA   $56

; Jump to the main command dispatch.
A876-   4C C8 A7    JMP   $A7C8
```

This is like an "LDA" instruction with an address. It can load from anywhere on the Apple // (not limited to the bytecode buffer) and stores it in the "register" at location $56.

Note the weird addressing mode that sets X=0 then immediately uses it as an index. There is no 6502 addressing mode to indirectly reference a zero page address. What we really want to do here is "LDA ($54)", but that doesn't exist. "LDA ($0054)" doesn't exist either. We're already using Y as an index into the bytecode buffer, so we clobber the X register and use "LDA ($54,X)" instead.

-o-

Command $05 is dispatched to $A88A:

```
*A88AL

; Get the next two bytes from the
; bytecode buffer.
A88A-   20 10 A8    JSR   $A810

; Collapse ($52),Y down to ($52).
; (That is, add Y to the address at
; location $52/$53.)
A88D-   98          TYA
A88E-   18          CLC
A88F-   65 52       ADC   $52
A891-   85 52       STA   $52
A893-   90 02       BCC   $A897
A895-   E6 53       INC   $53

; Push that address to the stack.
A897-   A5 52       LDA   $52
A899-   48          PHA
A89A-   A5 53       LDA   $53
A89C-   48          PHA

; Reset Y and branch to the middle of
; the dispatcher for command $00, like
; the virtual "BNE" command above.
A89D-   A0 00       LDY   #$00
A89F-   4C 03 A8    JMP   $A803
```

This is like a "JSR", but it's calling a bytecode address instead of a native address. The bytecode address of the caller is pushed to the stack. I assume there's some other bytecode command that simulates an "RTS" to pull that bytecode address off the stack and continue interpreting from there.

-o-

Command $06 is dispatched to $A8AD:

```
*A8ADL

; Get the next two bytes from the
; bytecode buffer.
A8AD-   20 10 A8    JSR   $A810

; Take the value of the $56 register.
A8B0-   A5 56       LDA   $56

; Store it in the (native) address
; that was given in the two bytes after
; the command.
A8B2-   A2 00       LDX   #$00
A8B4-   81 54       STA   ($54,X)

; Jump to the main command dispatch.
A8B6-   4C C8 A7    JMP   $A7C8
```

This is a "STA <native-address>" instruction. It takes the value of the register at location $56 and can store it anywhere in the Apple II memory (not just within the bytecode buffer).

Again with the weird addressing mode. We really want to say "STA ($54)", but that doesn't exist. So we set X=0 and use "STA ($54,X)" instead.

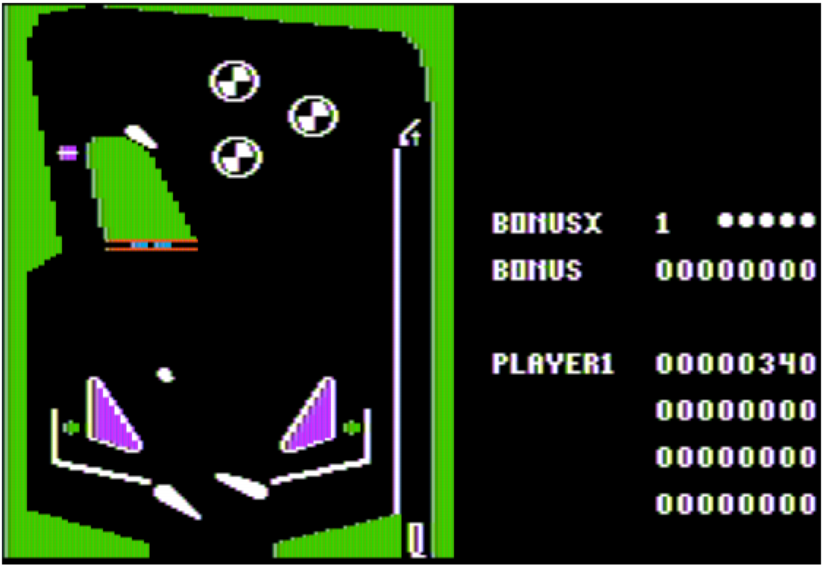
-o-

Command $07 is dispatched to $A8B9:

```
*A8B9L

; Get the next byte from the bytecode
; buffer.
A8B9-   B1 52       LDA   ($52),Y

; Advance the bytecode buffer pointer.
A8BB-   C8          INY
A8BC-   D0 02       BNE   $A8C0
A8BE-   E6 53       INC   $53

; Decrypt this byte with yet another
; key (actually the same key as command
; $03 used).
A8C0-   49 4C       EOR   #$4C

; Store the decrypted byte.
A8C2-   85 54       STA   $54

; Subtract the decrypted byte from the
; "register" at location $56.
A8C4-   A5 56       LDA   $56
A8C6-   38          SEC
A8C7-   E5 54       SBC   $54

; Store that value back to the register
; at $56.
A8C9-   85 56       STA   $56

; Jump to the main command dispatch.
A8CB-   4C C8 A7    JMP   $A7C8
```

This is a subtraction command. It takes a single byte from the bytecode buffer, decrypts, it, then subtracts it from the $56 register and stores the result back into the $56 register. Note how it abstracts away the carry flag (always important to set before using the 6502 native "SBC" instruction).

Let's call this "SUB <immediate>".



-o-

Command $08 is dispatched to $A8A2:

```
*A8A2L

; Pop two bytes off the stack and store
; them in $52/$53.
A8A2-   68          PLA
A8A3-   85 53       STA   $53
A8A5-   68          PLA
A8A6-   85 52       STA   $52

; Jump to the main command dispatch.
A8A8-   A0 00       LDY   #$00
A8AA-   4C C8 A7    JMP   $A7C8
```

Aha! This is the "RTS" that I posited when I examined command $05 (JSR to a bytecode address). That command pushed a bytecode address to the stack, and this command pops it off and continues the interpreter from that bytecode address.

I love it when a theory comes together.

-o-

Command $09 is dispatched to $A84F:

```
*A84FL

; Get the next two bytes from the
; bytecode buffer.
A84F-   20 10 A8    JSR   $A810

; Treat those bytes as a native address
; and jump there.
A852-   6C 54 00    JMP   ($0054)
```

This is a "JMP <native address>" command. I guess this is how you escape the interpreted machine. On second thought, the native code could easily jump back to $AC78 -- assuming nothing had clobbered Y or $52/$53 -- and the interpreter would pick up right where it left off.

Freely switching between interpreted and native code breaks my brain.

-o-

Command $0A is dispatched to $A8CE:

```
*A8CEL

; Get the next two bytes from the
; bytecode buffer.
A8CE-   20 10 A8    JSR   $A810

; Get the value of the register at
; location $56.
A8D1-   A2 00       LDX   #$00
A8D3-   A1 54       LDA   ($54,X)

; Add 1.
A8D5-   18          CLC
A8D6-   69 01       ADC   #$01

; Store that back into the register.
A8D8-   81 54       STA   ($54,X)
A8DA-   85 56       STA   $56

; Jump to the main command dispatch.
A8DC-   4C C8 A7    JMP   $A7C8
```

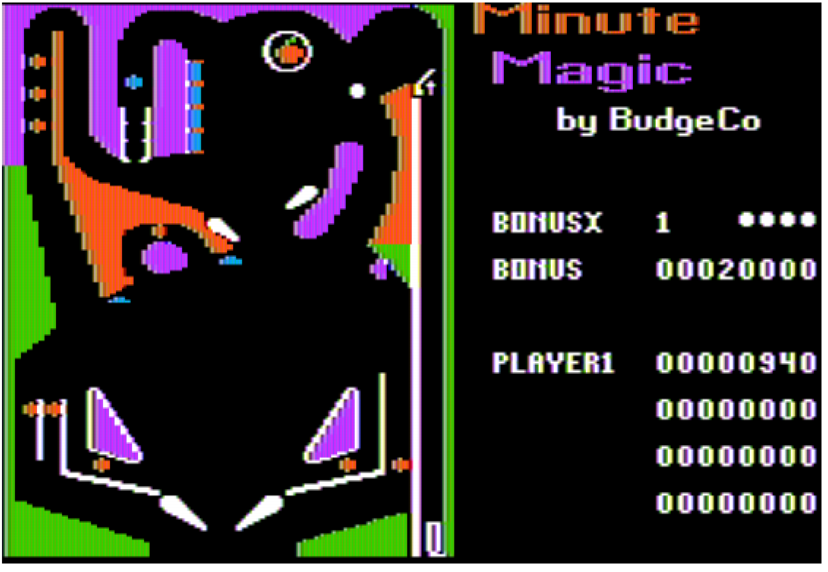
Cool, so we've incremented an address by 1, and set the register to the new value. It's an "INC" command.

-o-

Command $0B is dispatched to $A826:

```
*A826L

A826-   60          RTS
```

Hmm, it doesn't even jump back to the main command dispatch, so it will crash when it "returns" to a non-natively-executable address. Maybe this command was some sort of debugging entry point during development, then removed?



-o-

Command $0C is dispatched to $A879:

```
*A879L

; Get the next two bytes from the
; bytecode buffer.
A879-   20 10 A8    JSR   $A810

; Take the value of the "register" at
; location $56.
A87C-   A5 56       LDA   $56

; Add it to the two-byte value we just
; decrypted from the bytecode buffer.
A87E-   18          CLC
A87F-   65 54       ADC   $54
A881-   85 54       STA   $54
A883-   90 02       BCC   $A887
A885-   E6 55       INC   $55

; Jump to the middle of the handler for
; command $04.
A887-   4C 70 A8    JMP   $A870
```

As we saw above, this is the code at $A870:

```
; Load the value from that address.
A870-   A2 00       LDX   #$00
A872-   A1 54       LDA   ($54,X)

; Store it in the "register" at $56.
A874-   85 56       STA   $56
```

```
; Jump to the main command dispatch.
A876-   4C C8 A7     JMP   $A7C8
```

OK, this is like an indexed load command. It takes the value of the register and treats it as an index added to an arbitrary address as a base. The contents of the resulting address (which could be anywhere in Apple II memory) are stored back in the register at location $56.

-o-

In summary, this is an assembly-like language stored in memory as bytecode. There are 12 commands, numbered $00 to $0C ($0B is unused). Each command is followed by a single parameter, and each parameter is either an immediate value (one byte) or an address (two bytes). Each command always takes the same type of parameter, i.e. command $03 always takes a byte, command $04 always takes an address, &c. There is one register, stored in zero page $56, which can be referenced or modified by different commands.

There is no attempt at sandboxing. In fact, there are specific commands to access, increment, and even overwrite arbitrary memory locations. We can call native functions, passing the contents of our virtual register and storing the return value back into that register before continuing to the next command.

And everything is encrypted because f--- you.

\o/

R56 = 8-bit register (stored in zp$56)
 \# = byte length of command parameter

cmd	name	#	comment
$00	JMP	2	jump to bytecode addr
$01	JSRA	2	JSR to native addr
$02	BNE	2	BNE to bytecode addr
$03	LDI	1	R56 = <value>
$04	LDA	2	R56 = (native addr)
$05	JSR	2	JSR to bytecode addr
$06	STA	2	(native addr) = R56
$07	SUB	1	R56 -= <value>
$08	RTS	2	return from subroutine
$09	JMPA	2	jump to native addr
$0A	INC	2	R56 = ++(native addr)
$0B	DBUG	0	unused [crashes]
$0C	ILDA	2	R56=(native-addr),R56

So what do we do with this information, now that we have it?

Chapter 6
In Which We Write A Disassembler

If you recall, the entry point into the interpreter was at $A669:

```
*A669L

A669-   8D E4 A8      STA   $A8E4
A66C-   20 E6 A7      JSR   $A7E6
A66F-   20 B9 A7      JSR   $A7B9   <-- !
A672-   4C 69 A6      JMP   $A669
```

The routine at $A7B9 popped the stack, added 4, and started interpreting. So So the bytecode buffer starts at $A675.

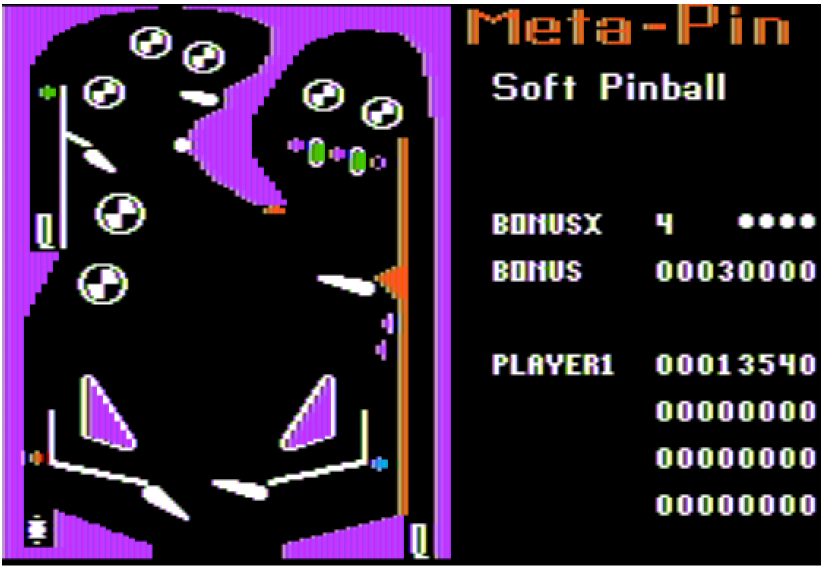
```
*A675.

A675- .. .. .. .. .. 04 E7 71
A678- 06 A3 7E 04 63 7F 07 74
A680- 02 A2 7E 04 62 7F 07 2C
A688- 02 A2 7E 04 64 7F 07 74
```

Even looking at it raw, we can make a bit of sense of it. The first byte is $04, which is the "LDA (native addr)" command. The next two bytes are the address we're loading, but of course they're XOR'd with different values so it's not immediately obvious what the address is. But the next command is at $A678, and it's command $06, which is "STA". Then another "LDA" at $A67B, a "SUB" at $A67E, &c.

Surely there's a better way.

The Apple II monitor (i.e. where we've been poking around with disassembly listings, not a physical monitor) has a little-used extension point called the <Ctrl-Y> vector. You can install your own callback on page 3 (like the reset vector) which will be called when you enter a command in the monitor and type <Ctrl-Y><Return>. The command you typed will be in the text input buffer at $0200, and you can do anything you like with it.

Let's write a disassembler.



--v--

```
;pcsdecoder.a
;Copyright (c) 2017 qkumba && 4am
;assemble with ACME
;<http://sourceforge.net/projects/
;acme-crossass/>

*=$7fd

   jmp  setup

GlobalHandler
   ldx  #$FD
   ldy  #0
-  jsr  asc2hex
   ora  $40
   sta  $40, x
   txa
   eor  #1
   tax
   lsr
   bcc  -
   iny
   inx
   inx
   bmi  -

   ldy  #0

print
; print the bytecode address
   lda  $3D
   jsr  $FDDA
   lda  $3C
   jsr  $FDDA
```

```
; and a space
  jsr  prspace

; get the command byte
  lda  ($3C), y
  cmp  #$0D
  bcs  alldone
  tax
  asl
  asl
  pha

; look up how many parameter bytes to
; expect after this command
  lda  parms, x
  sta  $41

; print the raw bytes (command byte
; followed by however many parameter
; bytes -- no attempt at decryption
; or deobfuscation yet)
  tax
- lda  ($3C), y
  jsr  $FDDA
  iny
  dex
  bpl  -
```

```
; print enough spaces so everything
; lines up (since not all commands have
; the same number of parameters)
  lda  $41
  sec
  sbc  #3
  asl
  tax
- jsr  prspace
  inx
  bne  -
  pla

; look up the command name and print it
  tax
  ldy  #4
- lda  names, x
  jsr  $FDED
  inx
  dey
  bne  -

; and another space
  jsr  prspace

  jsr  $FCBA
  ldy  $41
  beq  checkend
  dey
  php

; print either "#$" or "$", depending
; on the command
  bne  +
  lda  #$A3
  jsr  $FDED
+ lda  #$A4
  jsr  $FDED
```

```
; decrypt the parameter bytes and print
; them
   lda  #$4C
   plp
   beq  +
   lda  #$D9
   eor  ($3C), y
   jsr  $FDDA
   dey
   lda  #3
+  eor  ($3C), y
   jsr  $FDDA
   ldy  $41
-  jsr  $FCBA
   dey
   bne  -

checkend
; all done with this line -- print a
; carriage return and loop back if
; there are any more commands to
; disassemble
   php
   jsr  $FD8E
   plp
   bcc  print

alldone
; jump back to the monitor when we're
; done
    jmp   $ff69
```

```
; install the <Ctrl-Y> vector
setup
    lda   #$4c
    sta   CTRLY
    lda   #<GlobalHandler
    sta   CTRLY+1
    lda   #>GlobalHandler+1
    sta   CTRLY+2

; print a welcome message
    ldx   #0
    beq   +
-   jsr   $FDED
    inx
+   lda   welcome, x
    bne   -

    beq   alldone

; array of command names
names
   !text    "JMP " ;0
   !text    "JSRA" ;1
   !text    "BNE " ;2
   !text    "LDI " ;3
   !text    "LDA " ;4
   !text    "JSR " ;5
   !text    "STA " ;6
   !text    "SUB " ;7
   !text    "RTS " ;8
   !text    "JMPA" ;9
   !text    "INC " ;a
   !text    "    " ;b
   !text    "ILDA" ;c
```

```
; array of byte length of parameters
; after each command
parms
   !byte   2 ;0
   !byte   2 ;1
   !byte   2 ;2
   !byte   1 ;3
   !byte   2 ;4
   !byte   2 ;5
   !byte   2 ;6
   !byte   1 ;7
   !byte   0 ;8
   !byte   2 ;9
   !byte   2 ;a
   !byte   0 ;b
   !byte   2 ;c

; utility functions to convert between
; the ASCII values in the input buffer
; and the hex values they represent
asc2hex
   jsr  byt2hex
   asl
   asl
   asl
   asl
   sta  $40

byt2hex
   lda  $200, y
   iny
   sec
   sbc  #$B0
   cmp  #$0A
   bcc  +
   sbc  #7
+ rts
```

```
prspace
   lda  #$A0
   jmp  $FDED

welcome
   !text $8D,"EA BYTECODE DECODER "
   !text "INSTALLED.",$8D,$00
```

--^--

I've included a copy of the PCSDECODER binary on my work disk.

```
*BRUN PCSDECODER
EA BYTECODE DECODER INSTALLED.

*A675.A692<Ctrl-Y>

A675 04E771  LDA  $A8E4
A678 06A37E  STA  $A7A0
A67B 04637F  LDA  $A660
A67E 0774    SUB  #$38
A680 02A27E  BNE  $A7A1
A683 04627F  LDA  $A661
A686 072C    SUB  #$60
A688 02A27E  BNE  $A7A1
A68B 04647F  LDA  $A667
A68E 0774    SUB  #$38
A690 02A27E  BNE  $A7A1
```

Mirabile visu!

Chapter 7
Wax On, Wax Off
Phase On, Phase Off

Let's take this line by line.

```
; Copy one byte from $A8E4 to $A7A0.
; Note: $A8E4 was set at $A669 to the
; value of the accumulator on entry.
A675 04E771  LDA  $A8E4
A678 06A37E  STA  $A7A0

; If address $A660 is not #$38, branch
; to another bytecode address.
A67B 04637F  LDA  $A660
A67E 0774    SUB  #$38
A680 02A27E  BNE  $A7A1

; If address $A661 is not #$60, branch
; to that same bytecode address as
; above.
A683 04627F  LDA  $A661
A686 072C    SUB  #$60
A688 02A27E  BNE  $A7A1

; If address $A667 is not #$38, branch
; once again to that same bytecode
; address.
A68B 04647F  LDA  $A667
A68E 0774    SUB  #$38
A690 02A27E  BNE  $A7A1
```

I'm beginning to think this is some sort of anti-tamper check, and $A7A1 is the bytecode address of the fatal error handler.

```
*A660L

A660-   38          SEC
A661-   60          RTS
A662-   18          CLC
A663-   60          RTS
A664-   8D F6 A8    STA   $A8F6
A667-   38          SEC
A668-   60          RTS
```

OK, without knowing exactly what's going on yet, that definitely could be the success and failure paths of a protection check. By convention (taken from DOS 3.3), many checks will clear the carry on success or set it on failure. One common technique to defeat such checks is to change the "SEC" to a "CLC" so that the caller thinks that the failure was actually a success. Another technique, if the code is set up to support it, is to change the "RTS" after the "SEC" so it falls through to the "CLC" before returning.

So... and I'm just spitballing here... this could be another layer of anti-tamper checking, where the interpreted code ensures the nearby native code has not been altered on its way back to the caller.

Continuing the bytecode disassembly at $A693:

```
; call a bytecode subroutine
A693 05B87F   JSR  $A6BB
```

I can disassemble that subroutine too.

```
*A6BB.A6FF<Ctrl-Y>

; turn on slot 6 drive motor
A6BB 04EA19   LDA  $C0E9
```

Hey, it hadn't occurred to me, but of course the ability to "load" arbitrary native addresses means we have complete freedom to hit all the soft switches on the Apple // that control the disk, the graphics modes, the memory banks, the keyboard, and a bunch of other stuff.

Oh joy.

```
; Wait for the drive to spin up by
; loading our bytecode register with
; #$FF and calling out to the native
; WAIT routine at $FCA8. Twice. The
; "JSRA" bytecode command loads the
; native accumulator with the value of
; the bytecode register, so this works
; exactly as you would expect it to
; work.
A6BE 03B3     LDI  #$FF
A6C0 01AB25   JSRA $FCA8
A6C3 03B3     LDI  #$FF
A6C5 01AB25   JSRA $FCA8
```

```
; Reset the slot 6 data latch. (Like
; many protection schemes that *aren't*
; written in an obfuscated interpreted
; language like a madman, this will
; only work if you boot from slot 6.)
A6C8 04ED19   LDA   $C0EE

; Initialize a counter, maybe?
A6CB 034C     LDI   #$00
A6CD 06E171   STA   $A8E2

; Will examine this is a moment.
A6D0 05E07F   JSR   $A6E3
A6D3 05E07F   JSR   $A6E3
A6D6 05E07F   JSR   $A6E3
A6D9 05E07F   JSR   $A6E3

; Turn off slot 6 drive motor.
A6DC 04EB19   LDA   $C0E8

; Load the value of that counter on the
; way out.
A6DF 04E171   LDA   $A8E2

; Return to the (bytecode) caller
A6E2 08       RTS
```

OK, so no details yet, but this is definitely low-level disk-related code. We're turning on the floppy drive motor manually, resetting the data latch, doing something (bytecode at $A6E3), then turning off the drive motor on the way out.

So what's at $A6E3?

```
*A6E3.A711<Ctrl-Y>

; Initialize a counter, maybe?
A6E3 034F      LDI   #$03
A6E5 069C7E    STA   $A79F

; Call native subroutine (will examine
; this in a moment).
A6E8 01637E    JSRA $A760

; Stepper motor phase 3 ON
A6EB 04E419    LDA   $C0E7

; Call the same native subroutine again
; (twice).
A6EE 01637E    JSRA $A760
A6F1 01637E    JSRA $A760

; Stepper motor phase 0 ON
A6F4 04E219    LDA   $C0E1

; Call a second native subroutine (will
; examine shortly).
A6F7 057E7E    JSR   $A77D

; Re-initialize a counter, maybe?
A6FA 034F      LDI   #$03
A6FC 069C7E    STA   $A79F

; Call the first native subroutine
; again.
A6FF 01637E    JSRA $A760

; Stepper motor phase 3 ON (again)
A702 04E419    LDA   $C0E7
```

```
; Call the first native subroutine
; (twice).
A705 01637E  JSRA $A760
A708 01637E  JSRA $A760

; Stepper motor phase 2 ON
A70B 04E619  LDA  $C0E5

; Call the second native subroutine
A70E 057E7E  JSR  $A77D

; Return to the (bytecode) caller
A711 08      RTS
```

Whatever is happening at $A760, it happens WHILE THE DRIVE HEAD IS MOVING.

```
*A760L

A760-   20 12 A7    JSR   $A712

*A712L

; look for a $D5 nibble
A712-   A0 FF       LDY   #$FF
A714-   AE 9F A7    LDX   $A79F
A717-   AD EC C0    LDA   $C0EC
A71A-   10 FB       BPL   $A717
A71C-   C9 D5       CMP   #$D5
```

```
; branch forward once we find it
A71E-   F0 07       BEQ   $A727  ---+
A720-   88          DEY             |
A721-   D0 F4       BNE   $A717     |
A723-   CA          DEX             |
A724-   D0 F1       BNE   $A717     |
A726-   60          RTS             |
                                    |
; next nibble needs to be $AA       |
A727-   AD EC C0    LDA   $C0EC  <--+
A72A-   10 FB       BPL   $A727
A72C-   C9 AA       CMP   #$AA

; branch forward if it is
A72E-   F0 05       BEQ   $A735  ---+
                                    |
; otherwise start over              |
A730-   88          DEY             |
A731-   D0 E4       BNE   $A717     |
                                    |
; Y register acts as a Death        |
; Counter. If it hits 0, we set     |
; the carry flag and return to      |
; the caller.                       |
A733-   38          SEC             |
A734-   60          RTS             |
                                    |
; next nibble needs to be $96       |
A735-   AD EC C0    LDA   $C0EC  <--+
A738-   10 FB       BPL   $A735
                                [...]
```

```
A73A-   C9 96          CMP   #$96
A73C-   F0 05          BEQ   $A743   ---+
A73E-   88             DEY              |
A73F-   D0 D6          BNE   $A717      |
                                        |
; otherwise fail (as above)             |
A741-   38             SEC              |
A742-   60             RTS              |
                                        |
; parse through address field           |
; (4-and-4 encoded values)              |
A743-   A0 02          LDY   #$02    <--+
A745-   AD EC C0       LDA   $C0EC
A748-   10 FB          BPL   $A745
A74A-   2A             ROL
A74B-   85 50          STA   $50
A74D-   AD EC C0       LDA   $C0EC
A750-   10 FB          BPL   $A74D
A752-   25 50          AND   $50
A754-   85 50          STA   $50
A756-   88             DEY
A757-   8E 9F A7       STX   $A79F
A75A-   10 E9          BPL   $A745

; clear carry and return to caller
A75C-   18             CLC
A75D-   A2 01          LDX   #$01
A75F-   60             RTS
```

OK, so we're looking for the standard
"D5 AA 96" address prologue, then
parsing (but mostly ignoring) the
address field.

*** While the drive head is moving. ***

Continuing from $A763...

```
*A763L

; if anything went wrong looking for
; the address field, skip ahead (with
; the carry bit still set)
A763-   B0 03       BCS   $A768
A765-   20 03 A6    JSR   $A603

*A603L

; find the standard data field prologue
; "D5 AA AD"
A603-   A0 20       LDY   #$20
A605-   88          DEY
A606-   F0 58       BEQ   $A660
A608-   AD EC C0    LDA   $C0EC
A60B-   10 FB       BPL   $A608
A60D-   49 D5       EOR   #$D5    <--
A60F-   D0 F4       BNE   $A605
A611-   AD EC C0    LDA   $C0EC
A614-   10 FB       BPL   $A611
A616-   C9 AA       CMP   #$AA    <--
A618-   D0 F3       BNE   $A60D
A61A-   AD EC C0    LDA   $C0EC
A61D-   10 FB       BPL   $A61A
A61F-   C9 AD       CMP   #$AD    <--
A621-   D0 EA       BNE   $A60D
```

```
; look at raw nibbles (no actual
; decoding of 6-and-2 data, despite
; the fact that we just verified the
; standard address and data prologue)
A623-   48          PHA
A624-   68          PLA

; look at first $56 nibbles
A625-   A0 56       LDY   #$56
A627-   AD EC C0    LDA   $C0EC
A62A-   10 FB       BPL   $A627
A62C-   2C 00 C0    BIT   $C000

; every nibble must be $B5
A62F-   C9 B5       CMP   #$B5

; otherwise branch immediately to the
; failure path
A631-   D0 31       BNE   $A664
A633-   88          DEY
A634-   D0 F1       BNE   $A627

; do the same for the next $100 nibbles
A636-   A0 00       LDY   #$00
A638-   AD EC C0    LDA   $C0EC
A63B-   10 FB       BPL   $A638
A63D-   2C 00 C0    BIT   $C000

; again, every nibble must be $B5
A640-   C9 B5       CMP   #$B5

; otherwise fail immediately
A642-   D0 20       BNE   $A664
A644-   88          DEY
A645-   D0 F1       BNE   $A638
```

```
; read checksum nibble into Y register
A647-    AC EC C0      LDY    $C0EC
A64A-    10 FB         BPL    $A647
A64C-    48            PHA
A64D-    68            PLA

; verify "DE AA" data field epilogue
A64E-    AD EC C0      LDA    $C0EC
A651-    10 FB         BPL    $A64E
A653-    C9 DE         CMP    #$DE
A655-    D0 09         BNE    $A660
A657-    AD EC C0      LDA    $C0EC
A65A-    10 FB         BPL    $A657
A65C-    C9 AA         CMP    #$AA
A65E-    F0 02         BEQ    $A662

; Hey look! These are the addresses
; that the bytecode was checking to
; ensure they hadn't been modified.
A660-    38            SEC
A661-    60            RTS
A662-    18            CLC
A663-    60            RTS
A664-    8D F6 A8      STA    $A8F6
A667-    38            SEC
A668-    60            RTS
```

Continuing from $A768...

```
; Clever (ab)use of the carry bit here:
; if the carry is clear (because all
; that disk stuff succeeded), the value
; of $A8E2 will remain unchanged. Note:
; this is the counter we initialized in
; bytecode (at $A6CD).
A768-   A9 00       LDA   #$00
A76A-   6D E2 A8    ADC   $A8E2
A76D-   8D E2 A8    STA   $A8E2

; stepper motor phase 0 OFF
A770-   AD E0 C0    LDA   $C0E0

; stepper motor phase 1 OFF
A773-   AD E2 C0    LDA   $C0E2

; stepper motor phase 2 OFF
A776-   AD E4 C0    LDA   $C0E4

; stepper motor phase 3 OFF
A779-   AD E6 C0    LDA   $C0E6
A77C-   60          RTS
```

Disengaging all 4 stepper motors stops the drive head. The drive *motor* is still on; the disk is still spinning. But the head itself is no longer moving between tracks. It's at rest now. May it rest in peace.

Which brings us to $A77D. Literally, that's the next instruction we haven't examined, and it's also the entry point to a bytecode subroutine that was called earlier.

```
*A77D.A793<CTRL-Y>

; not shown, but this is a wait routine
A77D 033C     LDI  #$70
A77F 01977E   JSRA $A794

; disengage all 4 stepper motors (same
; as the native code at $A770)
A782 04E319   LDA  $C0E0
A785 04E119   LDA  $C0E2
A788 04E719   LDA  $C0E4
A78B 04E519   LDA  $C0E6

; wait again
A78E 0364     LDI  #$28
A790 01977E   JSRA $A794

; return to (bytecode) caller
A793 08       RTS
```

Chapter 8
What The Hell Is Going On

Returning to the original disk with the
Copy II Plus nibble editor, I can see
what's going on by examining track $05
and stepping by quarter tracks:

--v--

```
TRACK: 05  START: 1800  LENGTH: 3DFF

1EB0: FF FF FF FF FF FF FF FF    VIEW
1EB8: FF FF FF FF FF FF FF FF
1EC0: FF FF FF FF FF FF FF FF
1EC8: FF FF FF FF FF FF FF FF
1ED0: D5 AA 96 FF FE AA AF AA  <-1ED5
1ED8: AA FF FB DE AA EB FF FF
1EE0: FF FF FF FF D5 AA AD B5
1EE8: B5 B5 B5 B5 B5 B5 B5 B5  FIND:
1EF0: B5 B5 B5 B5 B5 B5 B5 B5  AA AF AA

.
.
.

TRACK: 05.25  START: 1800  LENGTH: 3DFF

2038: FF FF FF FF FF FF FF FF    VIEW
2040: FF FF FF FF FF FF FF FF
2048: FF FF FF FF FF FF FF FF
2050: FF FF FF FF D5 AA 96 FF
2058: FE AA AF AA AA FF FB DE  <-2059
2060: AA EB FF FF FF FF FF FF
2068: D5 AA AD B5 B5 B5 B5 B5
2070: B5 B5 B5 B5 B5 B5 B5 B5  FIND:
2078: B5 B5 B5 B5 B5 B5 B5 B5  AA AF AA

.
.
.
```

```
TRACK: 05.50  START: 1800  LENGTH: 3DFF

2080: FF FF FF FF FF FF FF FF    VIEW
2088: FF FF FF FF FF FF FF FF
2090: FF FF FF FF FF FF FF FF
2098: FF FF FF FF FF FF D5 AA
20A0: 96 FF FE AA AF AA AA FF  <-20A3
20A8: FB DE AA EB FF FF FF FF
20B0: FF FF D5 AA AD B5 B5 B5
20B8: B5 B5 B5 B5 B5 B5 B5 B5  FIND:
20C0: B5 B5 B5 B5 B5 B5 B5 B5  AA AF AA

.
.
.

TRACK: 05.75  START: 1800  LENGTH: 3DFF

2240: FF FF FF FF FF FF FF FF    VIEW
2248: FF FF FF FF FF FF FF FF
2250: FF FF FF FF FF FF FF FF
2258: FF FF FF FF FF D5 AA 96
2260: FF FE AA AF AA AA FF FB  <-2262
2268: DE AA EB FF FF FF FF FF
2270: FF D5 AA AD B5 B5 B5 B5
2278: B5 B5 B5 B5 B5 B5 B5 B5  FIND:
2280: B5 B5 B5 B5 B5 B5 B5 B5  AA AF AA

.
.
.
```

```
TRACK: 06  START: 1800  LENGTH: 3DFF

22A0: FF FF FF FF FF FF FF FF   VIEW
22A8: FF FF FF FF FF FF FF FF
22B0: FF FF FF FF FF FF FF FF
22B8: FF FF FF FF FF FF FF D5
22C0: AA 96 FF FE AA AF AA AA  <-22C4
22C8: FF FB DE AA EB FF FF FF
22D0: FF FF FF D5 AA AD B5 B5
22D8: B5 B5 B5 B5 B5 B5 B5 B5  FIND:
22E0: B5 B5 B5 B5 B5 B5 B5 B5  AA AF AA

                 --^--
```

Here's the thing: starting on track 5, there are FIVE CONSECUTIVE IDENTICAL SYNCHRONIZED QUARTER TRACKS! Tracks 5, 5.25, 5.5, 5.75, and 6 contain the same stream of bits, perfectly synchronized to the point that you could read a sector from disk while the drive head was moving between tracks.

Which is precisely what this protection check is doing. It seeks to track $05 by reading T05,S00 (at $1E85). Then it calls the bytecode routine at $A6E3. Let's revisit that routine; it might make a modicum of sense now. YMMV(*)

```
*A6E3.A711<Ctrl-Y>

A6E3 034F     LDI  #$03
A6E5 069C7E   STA  $A79F

; read sector
A6E8 01637E   JSRA $A760
```

(*) Your modicum may vary

```
; engage drive stepper motor 3
A6EB 04E419  LDA  $C0E7

; read sector (twice), as above, and
; verify that all nibbles are read as
; expected even though the drive head
; is now moving
A6EE 01637E  JSRA $A760
A6F1 01637E  JSRA $A760

; engage drive stepper motor 1
A6F4 04E219  LDA  $C0E1
; wait, then disengage all stepper
; motors
A6F7 057E7E  JSR  $A77D
```

I believe, although I'm not positive, that we are now on track $06 (which claims to be track $05, but never mind that).

Now we do the same test, but moving in the opposite direction:

```
A6FA 034F    LDI  #$03
A6FC 069C7E  STA  $A79F

; read sector while drive head is
; stationary
A6FF 01637E  JSRA $A760

; Stepper motor phase 3 ON (again)
A702 04E419  LDA  $C0E7

; read sector while drive head is
; moving back towards track 5
A705 01637E  JSRA $A760
A708 01637E  JSRA $A760
```

```
; Stepper motor phase 2 ON
A70B 04E619  LDA  $C0E5

; wait, then disengage all stepper
; motors
A70E 057E7E  JSR  $A77D
A711 08      RTS
```

The important part about $A760 is not that it reads a sector worth of nibbles and verifies them. It's that it does it WHILE THE DRIVE HEAD IS MOVING. Then it does it again, but with the head moving in the opposite direction.

That is, in a word, impossible. (To duplicate. Also to master. How the hell did they write out these disks in the first place?) Which is why they assumed that people would try to edit out the protection. Which is why they went to such great lengths to ensure that key elements of the code weren't modified.

Remember when I thought that track $06 was lying by claiming to be track $05? It's so much worse than that! Not only are the two tracks identical, so is LITERALLY EVERYTHING IN BETWEEEN.

Chapter 9
What The Hell Can We Do About It

If you recall (it's OK if you don't), we were in the middle of disassembling the bytecode routine at $A675. We'd gotten as far as $A693, which was a JSR to the (bytecode) routine at $A6BB.

Continuing from $A696, which is once again bytecode...

```
; On exit of the bytecode routine at
; $A6BB, the virtual register contains
; the value of the counter at $A8E2. If
; it's still 0, everything went well.
; If it's not 0, branch ahead.
A696 029F7F   BNE  $A69C

; all is well; continue on success path
A699 00AE7F   JMP  $A6AD

; Execution continues here (from $A696)
; We're apparently willing to give it
; all one more try before giving up.
A69C 05B87F   JSR  $A6BB
A69F 02A67F   BNE  $A6A5
A6A2 00AE7F   JMP  $A6AD

; If the counter coming out of the disk
; routine is 2, jump back to $A67B to
; start the entire copy protection
; routine over again. Otherwise branch
; forward to $A7A1.
A6A5 074E     SUB  #$02
A6A7 02A27E   BNE  $A7A1
A6AA 00787F   JMP  $A67B
```

```
*A7A1.A7A5<CTRL-Y>

A7A1 036C     LDI  ##$20
A7A3 09AD7E   JMPA $A7AE

*A7AEL

; Restore zero page locations we saved
; earlier. (Note that this subroutine
; loads the accumulator with $A8E2 on
; the way out.)
A7AE-   20 E6 A7     JSR   $A7E6

; Set zp$48 to $A8E2, then set the
; carry and return to the (native)
; caller.
A7B1-   8D 48 00     STA   $0048
A7B4-   38           SEC
A7B5-   60           RTS

Meanwhile, back in bytecode-land:

; Success path continues here (from
; $A6A2). This address was set at the
; very beginning of the bytecode
; routine (at $A678).
A6AD 04A37E   LDA  $A7A0

; Indexed read of $A67B + ($A7A0)
A6B0 0C787F   ILDA $A67B

; Subtract ##$B5
A6B3 07F9     SUB  ##$B5

; Store that back in $A7A0
A6B5 06A37E   STA  $A7A0
```

```
; Jump out of interpreter and continue
; execution in native code
A6B8 09A57E  JMPA $A7A6

*A7A6L

; Restore zero page locations we saved
; earlier. (Note that this subroutine
; loads the accumulator with $A8E2 on
; the way out.)
A7A6-   20 E6 A7     JSR   $A7E6

; Set zp$48 to $A8E2, then clear the
; carry and return to the (native)
; caller.
A7A9-   8D 48 00     STA   $0048
A7AC-   18           CLC
A7AD-   60           RTS
```

And now we're out of the bytecode interpreter and back to... $1E8D, after the "JSR $A600" at $1E8A.

```
*1E8DL

; check return code stored in location
; $48 (0 = success)
1E8D-   A5 48        LDA   $48

; if protection check was unsuccessful,
; branch forward
1E8F-   D0 01        BNE   $1E92

; otherwise return to the caller
1E91-   60           RTS
```

```
; Ah! We've already examined this code.
; It was the failure path if the tamper
; check at $1AA6 failed. It displays
; a graphical "Insert PCS disk" message
; over the title screen.
1E92-   A0 01       LDY   #$01
1E94-   20 CF 17    JSR   $17CF
1E97-   20 D9 1E    JSR   $1ED9
1E9A-   A0 03       LDY   #$03
1E9C-   20 CF 17    JSR   $17CF
1E9F-   20 D9 1E    JSR   $1ED9
1EA2-   A0 02       LDY   #$02
1EA4-   20 CF 17    JSR   $17CF
1EA7-   A9 F6       LDA   #$F6
1EA9-   A2 1E       LDX   #$1E
1EAB-   20 52 19    JSR   $1952
...
```

So that's it. Our descent into bytecode is complete, and we have returned to native code more or less unscathed.

Now, what the hell do we do about it?

The solution is made somewhat simpler by a bug we discovered earlier: this code at $1Exx is supposed to be tamper-checked by the subroutine at $1AA6, but it isn't. (Honestly, even if the tamper check worked properly, all I would need to do is remove the JSR at $1E82 so the tamper check is never called. This bug saves me exactly one byte.)

Since the call to $A600 will always return, I can simply change the "BNE" instruction after it returns so it always branches to the "RTS" at $1E91 instead of the error routine at $1E92 that displays "Insert PCS disk."

According to my calculations earlier, the code at $1E00 was read from T03,S06 by the multi-sector read routine (when X=1). And lo, there it is, ready for a one-bit patch:

```
T03,S06,$90: 01 -> 00
```

Quod erat liberandum.

Acknowledgments

Many thanks to qkumba for writing the initial version of PCSDECODER and reviewing drafts of this write-up. Oh, and he also expanded this crack into a near-universal EA patcher and added it to Passport. So now anyone can automate the cracking of about two dozen disks that use the same protection. Booyah.

A 4am and san inc crack   No. 1033
------------------EOF------------------