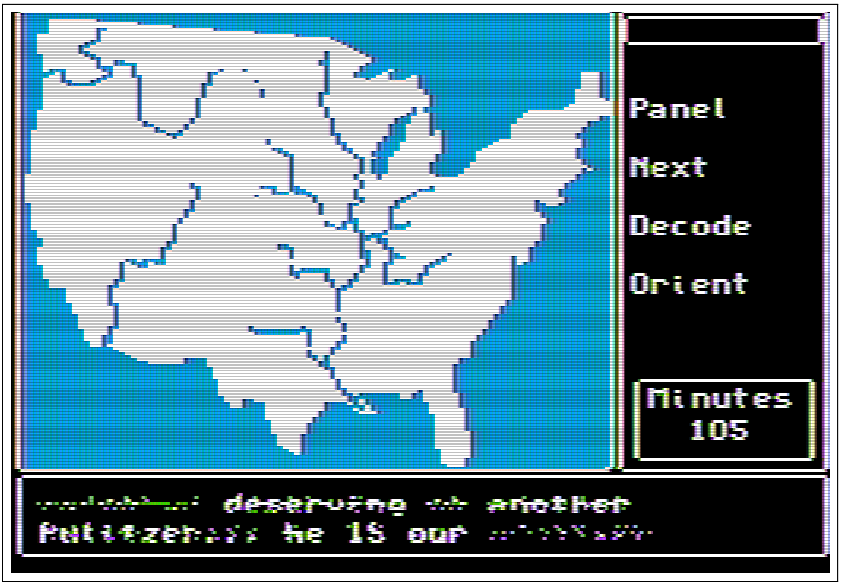
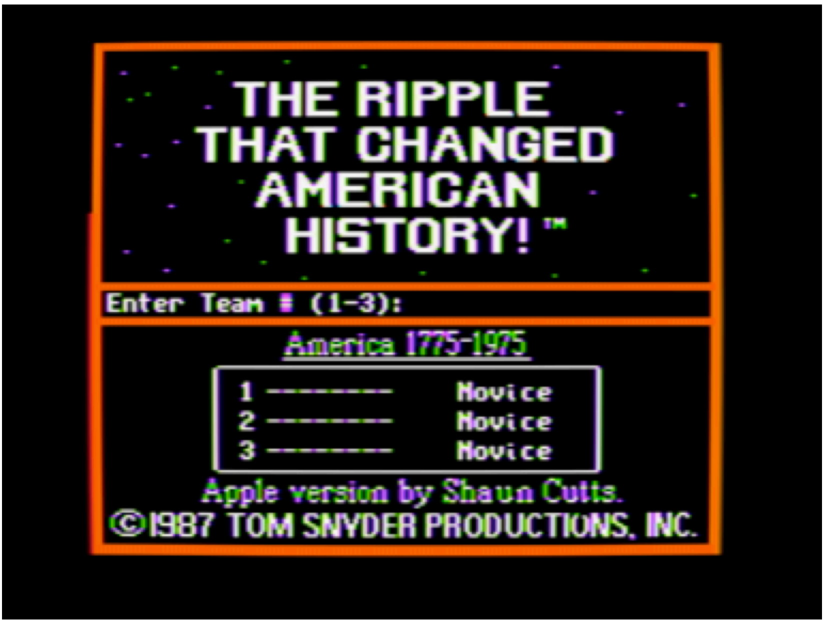
The Ripple That Changed American History!



2016-11-13

4am
to deprotect
and preserve

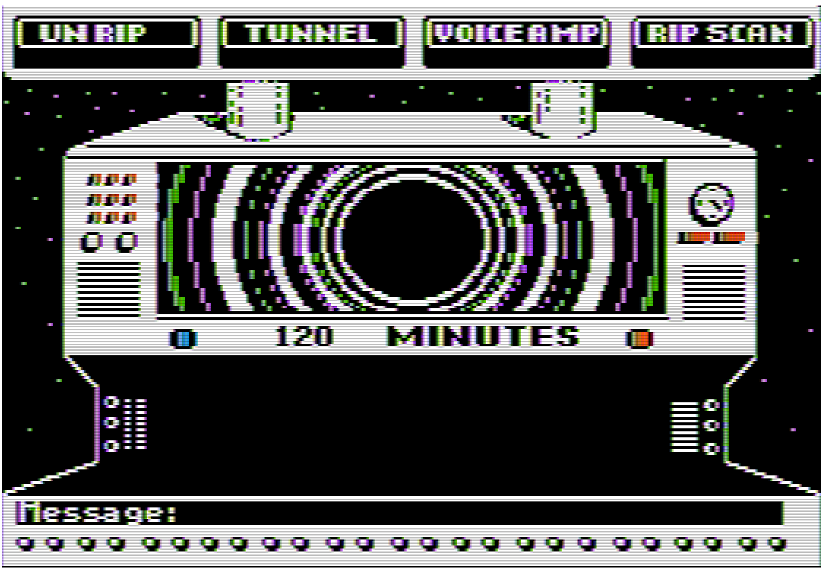
Contents



THE RIPPLE THAT CHANGED AMERICAN HISTORY!™
Enter Team # (1-3):
America 1775-1975
1 ———— Novice
2 ———— Novice
3 ———— Novice
Apple version by Shaun Cutts.
©1987 TOM SNYDER PRODUCTIONS, INC.

--------The Ripple That Changed--------
------------American History-----------
A 4am crack   2016-11-13

Name: The Ripple That Changed American
  History
Genre: educational
Year: 1987
Authors: Shaun Cutts
Publisher: Tom Snyder Productions
Media: single-sided 5.25-inch floppy
OS: custom
Previous cracks: none



Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

```
COPYA
  fails on second pass

Locksmith Fast Disk Backup
  copies everything except track $0E;
  copy boots but hangs just before the
  title screen

EDD 4 bit copy (no sync, no count)
  no errors, but copy boots then clears
  the screen and says "PLEASE REBOOT"
  just before the title screen

Copy ][+ nibble editor
  track $0E is all $FF (sync bytes)
  with just a handful of non-sync bytes

Disk Fixer
  T00,S00 -> custom bootloader
  no sign of DOS 3.3 or a standard RWTS
  no sign of any OS at all

Why didn't any of my copies work?
  Probably a runtime protection check
  that looks at the funky track $0E.

Next steps:

  1. Trace the boot
  2. ???
```

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.

Stay with me, this is all about to come together and go boom.

$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, loads game...
```

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

]BSAVE BOOT0,A$2800,L$100
```

Now let's see how this disk boots.

Chapter 2
Boot Trace and Chill

```
]CALL -151

; move boot0 back into place
*800<2800.28FFM

*801L

; the disk controller ROM always exits
; via $0801, so set that to an RTS so
; we can JSR and not have to set up a
; loop
0801-   A9 60       LDA   #$60
0803-   8D 01 08    STA   $0801

; munge reset vector
0806-   8D F4 03    STA   $03F4
0809-   78          SEI
080A-   A2 00       LDX   #$00
080C-   8E 78 04    STX   $0478
080F-   A2 09       LDX   #$09
0811-   86 27       STX   $27
0813-   A9 00       LDA   #$00
0815-   A2 01       LDX   #$01
0817-   A0 05       LDY   #$05

; multi-sector read loop that takes
; zp$27 = start address ($0900)
;     A = track ($00)
;     X = first logical sector ($01)
;     Y = last logical sector ($05)
0819-   20 1F 08    JSR   $081F
```

So we're reading sectors 1-5 of track 0 into $0900.

```
; execution continues elsewhere (in the
; code we just read)
081C-   4C DB 0C    JMP   $0CDB
```

I can use the same technique to trace the boot and interrupt the process, adding a callback to patch the JMP instruction at $081C so I can capture the code at $0900+.

```
*9600<C600.C6FFM

; set up callback after boot0, before
; jumping to $0CDB
96F8-   A9 4C       LDA   #$4C
96FA-   8D 1C 08    STA   $081C
96FD-   A9 0A       LDA   #$0A
96FF-   8D 1D 08    STA   $081D
9702-   A9 97       LDA   #$97
9704-   8D 1E 08    STA   $081E

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

; (callback is here)
; turn off slot 6 drive motor
970A-   AD E8 C0    LDA   $C0E8

; reboot to my work disk
970D-   4C 00 C5    JMP   $C500

*BSAVE TRACE2,A$9600,L$110

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

]BSAVE BOOT1 0900-0DFF,A$900,L$500
```

Chapter 3
Boot Trace and Boot Trace and Boot Trace and Chill

```
]CALL -151

*CDBL

0CDB-    A9 00          LDA   #$00
0CDD-    8D 51 C0       STA   $C051
0CE0-    8D 00 C0       STA   $C000

; clears hi-res screen (not shown)
0CE3-    20 25 0D       JSR   $0D25
0CE6-    A9 00          LDA   #$00
0CE8-    85 F6          STA   $F6
0CEA-    A9 5B          LDA   #$5B
0CEC-    85 14          STA   $14
0CEE-    A9 00          LDA   #$00
0CF0-    85 F0          STA   $F0
0CF2-    A9 06          LDA   #$06
0CF4-    85 F1          STA   $F1
0CF6-    A9 60          LDA   #$60
0CF8-    85 F7          STA   $F7
0CFA-    A9 01          LDA   #$01
0CFC-    85 F2          STA   $F2
0CFE-    A9 00          LDA   #$00
0D00-    85 F6          STA   $F6

; another multi-sector read routine
; zp$14 = sector count ($5B)
; zp$F0 = start track ($00)
; zp$F1 = start sector ($06)
; zp$F6 = full starting address ($6000)
0D02-    20 0B 0D       JSR   $0D0B
```

So we're reading $5B sectors into
$6000..$BAFF.

```
; turn off drive motor
0D05-    BD 88 C0       LDA   $C088,X
```

```
; and continue elsewhere (in the code
; we just read)
0D08-    4C DB 67     JMP   $67DB
```

Let's capture it.

```
*9600<C600.C6FFM

; set up callback after boot0
96F8-    A9 4C        LDA   #$4C
96FA-    8D 1C 08     STA   $081C
96FD-    A9 0A        LDA   #$0A
96FF-    8D 1D 08     STA   $081D
9702-    A9 97        LDA   #$97
9704-    8D 1E 08     STA   $081E

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

; (callback is here)
; break into monitor instead of jumping
; to $67DB
970A-    A9 4C        LDA   #$4C
970C-    8D 08 0D     STA   $0D08
970F-    A9 59        LDA   #$59
9711-    8D 09 0D     STA   $0D09
9714-    A9 FF        LDA   #$FF
9716-    8D 0A 0D     STA   $0D0A

; continue the boot
9719-    4C DB 0C     JMP   $0CDB

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

*9600G
...reboots slot 6...
<beep>

*2000<6000.BFFFM
```

```
*C500G
...
]BSAVE BOOT2 6000-BAFF,A$2000,L$5B00
]BLOAD BOOT2 6000-BAFF,A$6000
]CALL -151

*67DBL

67DB-   A9 01       LDA   #$01
67DD-   85 04       STA   $04
67DF-   A9 50       LDA   #$50
67E1-   85 06       STA   $06
67E3-   A9 04       LDA   #$04
67E5-   85 05       STA   $05
67E7-   A9 06       LDA   #$06
67E9-   85 02       STA   $02
67EB-   A9 02       LDA   #$02
67ED-   85 03       STA   $03

; this is yet another multi-sector read
; routine
; zp$02 = start track ($06)
; zp$03 = start sector ($02)
; zp$05 = sector count ($04)
; zp$06 = start address ($5000)
67EF-   20 93 92    JSR   $9293
```

Se we're loading 4 sectors into $5000.

```
67F2-   A9 01       LDA   #$01
67F4-   85 04       STA   $04
67F6-   A9 20       LDA   #$20
67F8-   85 06       STA   $06
67FA-   A9 20       LDA   #$20
67FC-   85 05       STA   $05
67FE-   A9 06       LDA   #$06
6800-   85 02       STA   $02
6802-   A9 06       LDA   #$06
6804-   85 03       STA   $03
6806-   20 93 92    JSR   $9293
```

We're loading $20 sectors into $2000.

```
6809-   20 27 A4    JSR   $A427
```

```
*A427L

A427-   D8          CLD
A428-   8D 00 C0    STA   $C000
A42B-   A9 C0       LDA   #$C0
A42D-   8D F6 B8    STA   $B8F6
A430-   A9 01       LDA   #$01
A432-   85 04       STA   $04
A434-   A9 BE       LDA   #$BE
A436-   85 06       STA   $06
A438-   A9 02       LDA   #$02
A43A-   85 05       STA   $05
A43C-   A9 0F       LDA   #$0F
A43E-   85 02       STA   $02
A440-   A9 00       LDA   #$00
A442-   85 03       STA   $03
A444-   20 93 92    JSR   $9293
```

We're loading 2 sectors into $BE00.

```
A447-   20 AF BE    JSR   $BEAF
```

And that's in just-loaded code, but I can set up another boot trace to capture it. However, I'll need to move my trace program somewhere else, since $9600 gets clobbered during boot and I need to have callbacks running after that part is loaded from disk.

Luckily, I can put my boot tracer
anywhere in memory, as long as it's at
$x600. Looking at all the disk reads,
it appears that $1600 is untouched
during boot.

```
*1600<C600.C6FFM

; set up callback #1
16F8-    A9 4C          LDA   #$4C
16FA-    8D 1C 08       STA   $081C
16FD-    A9 0A          LDA   #$0A
16FF-    8D 1D 08       STA   $081D
1702-    A9 17          LDA   #$17
1704-    8D 1E 08       STA   $081E

; start the boot
1707-    4C 01 08       JMP   $0801

; (callback #1 is here)
; set up callback #2
170A-    A9 4C          LDA   #$4C
170C-    8D 08 0D       STA   $0D08
170F-    A9 1C          LDA   #$1C
1711-    8D 09 0D       STA   $0D09
1714-    A9 17          LDA   #$17
1716-    8D 0A 0D       STA   $0D0A

; continue the boot
1719-    4C DB 0C       JMP   $0CDB
```

```
; (callback #2 is here)
; patch the routine at $A427 to break
; unconditionally to the monitor after
; loading sectors into $BE00+
171C-   A9 4C         LDA   #$4C
171E-   8D 47 A4      STA   $A447
1721-   A9 59         LDA   #$59
1723-   8D 48 A4      STA   $A448
1726-   A9 FF         LDA   #$FF
1728-   8D 49 A4      STA   $A449

; continue the boot
172B-   4C DB 67      JMP   $67DB

*BSAVE TRACE4,A$1600,L$12E
*1600G
...reboots slot 6...
...read read read...
<beep>

Success!
```

Chapter 4
Self-Modifying Code Is Best Code!

```
*BEAFL

BEAF-   4E B2 BE    LSR   $BEB2
BEB2-   DD B5 BE    CMP   $BEB5,X
BEB5-   44          ???
BEB6-   00          BRK
BEB7-   6E BA BE    ROR   $BEBA
```

Ooh, self-modifying code. This looks like garbage, but it's not (quite). The first instruction at $BEAF changes the next instruction at $BEB2. The disassembly listing beyond that is misleading, because the code will be different by the time it's run.

I'll step through this one line at a time.

Let's save it first.

```
*2E00<BE00.BFFFM

*C500G
...
]BSAVE OBJ.BE00-BFFF,A$2E00,L$200
]CALL -151

*FE89G FE93G
*BE00<2E00.2FFFM

*BEAFL

BEAF-   4E B2 BE    LSR   $BEB2
BEB2-   DD B5 BE    CMP   $BEB5,X
BEB5-   44          ???
BEB6-   00          BRK
BEB7-   6E BA BE    ROR   $BEBA
```

OK, game plan:

- $2E00..$2FFF is a "pristine" copy of the obfuscated code. If I need to reset, I can copy from there.
- I need to reproduce the self-modifying instructions at $BEAF+, but somewhere else, so that I can stop execution and examine it before it runs the modified code.
- I have no interest in doing bit math by hand, so I'm going to let the computer do it for me. They're good at that. :-)

Thus, a short program at $2000:

```
; reset the code to a pristine state
2000-   A0 00       LDY   #$00
2002-   B9 00 2E    LDA   $2E00,Y
2005-   99 00 BE    STA   $BE00,Y
2008-   B9 00 2F    LDA   $2F00,Y
200B-   99 00 BF    STA   $BF00,Y
200E-   C8          INY
200F-   D0 F1       BNE   $2002

; reproduce the self-modification
2011-   4E B2 BE    LSR   $BEB2
2014-   60          RTS

*2000G
```

```
*BEB2L

BEB2-   6E B5 BE    ROR   $BEB5
```

As I suspected, there is more self-modifying code hidden behind the first self-modified code. But I can extend my deobfuscation program at $2000 to reproduce it.

```
*2014:6E B5 BE 60

; will reset to a pristine state, then
; execute both layers of self-
; modification, then stop
*2000G

*BEB5L

BEB5-   A2 00       LDX   #$00
BEB7-   6E BA BE    ROR   $BEBA
```

Oh God, there's more.

```
*2017:A2 00 6E BA BE 60

*2000G

*BEAFL

BEAF-   4E B2 BE    LSR   $BEB2
BEB2-   6E B5 BE    ROR   $BEB5
BEB5-   A2 00       LDX   #$00
BEB7-   6E BA BE    ROR   $BEBA
BEBA-   6E C9 BE    ROR   $BEC9
BEBD-   6E C0 BE    ROR   $BEC0
```

And again.

```
*201C:6E C9 BE 6E C0 BE 60

*2000G

*BEC0L

BEC0-   6E CD BE    ROR   $BECD
BEC3-   6E D0 BE    ROR   $BED0
BEC6-   38          SEC
BEC7-   08          PHP
BEC8-   28          PLP
BEC9-   7E D3 BE    ROR   $BED3,X
BECC-   08          PHP
```

*BEC0 alters $BECD. $BEC3 alters $BED0.

```
*2022:6E CD BE 6E D0 BE 60

*2000G

*BEC6L

BEC6-   38          SEC
BEC7-   08          PHP
BEC8-   28          PLP
BEC9-   7E D3 BE    ROR   $BED3,X
BECC-   08          PHP
BECD-   E8          INX
BECE-   E0 E6       CPX   #$E6
BED0-   90 F6       BCC   $BEC8
```

Ah, this is probably the last layer of self-modification. It's a loop that alters an entire range of memory starting at $BED3.

```
*2028:38 08 28 7E D3 BE 08 E8 E0 E6 90
 F6 60

*2000G

*BED3L

BED3-    A5 F0          LDA    $F0
BED5-    48             PHA
BED6-    A5 F2          LDA    $F2
BED8-    48             PHA
BED9-    A9 0E          LDA    #$0E
BEDB-    85 F0          STA    $F0
BEDD-    A9 00          LDA    #$00
BEDF-    85 F2          STA    $F2
BEE1-    20 00 67       JSR    $6700
```

Aha! Real code at last!

Chapter 5
One Byte To Rule Them All

```
; save some zero page locations on the
; stack
BED3-   A5 F0         LDA   $F0
BED5-   48            PHA
BED6-   A5 F2         LDA   $F2
BED8-   48            PHA

; move drive head to track $0E (the
; unreadable track!)
BED9-   A9 0E         LDA   #$0E
BEDB-   85 F0         STA   $F0
BEDD-   A9 00         LDA   #$00
BEDF-   85 F2         STA   $F2
BEE1-   20 00 67      JSR   $6700

; hard-coded slot 6 (very common in
; protection checks -- try booting your
; favorite original disk from slot 5
; sometime, then watch as it switches
; to read from slot 6 for no reason!)
BEE4-   A2 60         LDX   #$60

; turn on drive motor
BEE6-   BD 89 C0      LDA   $C089,X

; restore zero page
BEE9-   68            PLA
BEEA-   85 F2         STA   $F2
BEEC-   68            PLA
BEED-   85 F0         STA   $F0

; find a nibble
BEEF-   BD 8C C0      LDA   $C08C,X
BEF2-   10 FB         BPL   $BEEF
```

```
; But not just any nibble -- one with
; a timing bit after it. The PHA/PLA
; burns enough cycles that the data
; latch will only retain its value that
; long if the nibble is followed by a
; timing bit.
BEF4-   48              PHA
BEF5-   68              PLA

; Specifically, this nibble needs to be
; $D5
BEF6-   C9 D5           CMP   #$D5

; otherwise we loop forever
BEF8-   D0 F5           BNE   $BEEF
```

This explains why the copy I made with Locksmith Fast Disk Backup would just hang forever. On a standard formatted track, there are no $D5 nibbles with timing bits after them. So this loop will never end.

The EDD bit copy (which actually tried to preserve track $0E) did successfully copy the $D5 nibble + timing bit. So it must fail later on.

Onward...

```
BEFA-   A0 00           LDY   #$00
BEFC-   8C 6D BF        STY   $BF6D
```

```
; look for nibbles that are not $D5 or
; $F7
BEFF-   BD 8C C0    LDA   $C08C,X
BF02-   10 FB       BPL   $BEFF
BF04-   C9 D5       CMP   #$D5
BF06-   F0 0F       BEQ   $BF17
BF08-   C9 F7       CMP   #$F7
BF0A-   D0 01       BNE   $BF0D

; increment two separate counters, one
; in the Y register and one in $BF6D
BF0C-   C8          INY
BF0D-   18          CLC
BF0E-   6D 6D BF    ADC   $BF6D
BF11-   8D 6D BF    STA   $BF6D
BF14-   4C FF BE    JMP   $BEFF

; executionn continues here when we hit
; an $F7 nibble -- check the Y register
; counter and start over if it's still
; zero
BF17-   98          TYA
BF18-   F0 E0       BEQ   $BEFA

; look for sync byte ($FF) with timing
; bit after it
BF1A-   BD 8C C0    LDA   $C08C,X
BF1D-   10 FB       BPL   $BF1A
BF1F-   24 00       BIT   $00
BF21-   C9 FF       CMP   #$FF
BF23-   D0 05       BNE   $BF2A
BF25-   C8          INY
BF26-   D0 F2       BNE   $BF1A
BF28-   F0 10       BEQ   $BF3A

; if first non-$FF nibble is $D5, fail
BF2A-   C9 D5       CMP   #$D5
BF2C-   F0 31       BEQ   $BF5F
```

```
; skip over 6 nibbles
BF2E-   A0 05       LDY   #$05
BF30-   BD 8C C0    LDA   $C08C,X
BF33-   10 FB       BPL   $BF30
BF35-   48          PHA
BF36-   68          PLA
BF37-   88          DEY
BF38-   D0 F6       BNE   $BF30

; more sync bytes
BF3A-   BD 8C C0    LDA   $C08C,X
BF3D-   10 FB       BPL   $BF3A
BF3F-   48          PHA
BF40-   68          PLA
BF41-   C9 FF       CMP   #$FF
BF43-   F0 F5       BEQ   $BF3A

; $D5 at this point = fail
BF45-   C9 D5       CMP   #$D5
BF47-   D0 16       BNE   $BF5F

; non-$FF at this point = fail
BF49-   BD 8C C0    LDA   $C08C,X
BF4C-   10 FB       BPL   $BF49
BF4E-   C9 FF       CMP   #$FF
BF50-   D0 0D       BNE   $BF5F

; check the count that was accumulating
; in $BF6D
BF52-   AD 6D BF    LDA   $BF6D
BF55-   C9 10       CMP   #$10
BF57-   D0 06       BNE   $BF5F
```

```
; success path falls through to here --
; turn off drive motor and continue
; elsewhere
BF59-   BD 88 C0    LDA   $C088,X
BF5C-   4C AF BF    JMP   $BFAF

*BFAFL

; wipe the entire protection check from
; memory
BFAF-   A0 FF       LDY   #$FF
BFB1-   98          TYA
BFB2-   99 AF BE    STA   $BEAF,Y
BFB5-   88          DEY
BFB6-   D0 FA       BNE   $BFB2

; and return gracefully (all the way
; back to the next instruction after
; the JSR at $A447)
BFB8-   60          RTS
```

Note: this also clobbers the counter at $BF6D, which means that there are no side effects to this protection check. The caller has no way to verify that the check succeeded beyond "execution returned to the caller so it must have worked." That's good for us.

Meanwhile, on the failure path...

```
*BF5FL

; copy a bit of code to lower memory
BF5F-   A0 40       LDY   #$40
BF61-   B9 6E BF    LDA   $BF6E,Y
BF64-   99 00 03    STA   $0300,Y
BF67-   88          DEY
BF68-   10 F7       BPL   $BF61
BF6A-   4C 00 03    JMP   $0300
```

```
; this will be executed from $0300 --
; clear main memory, $0800..$BFFF
BF6E-   BD 88 C0    LDA   $C088,X
BF71-   A0 00       LDY   #$00
BF73-   A2 B8       LDX   #$B8
BF75-   99 00 BF    STA   $BF00,Y
BF78-   88          DEY
BF79-   D0 FA       BNE   $BF75
BF7B-   CE 09 03    DEC   $0309
BF7E-   CA          DEX
BF7F-   8A          TXA
BF80-   D0 F3       BNE   $BF75

; switch back to text mode
BF82-   AD 54 C0    LDA   $C054
BF85-   AD 51 C0    LDA   $C051
BF88-   AD 81 C0    LDA   $C081
BF8B-   20 93 FE    JSR   $FE93
BF8E-   20 89 FE    JSR   $FE89
BF91-   20 58 FC    JSR   $FC58

; display error message
BF94-   A0 0C       LDY   #$0C
BF96-   B9 34 03    LDA   $0334,Y
BF99-   99 08 07    STA   $0708,Y
BF9C-   88          DEY
BF9D-   10 F7       BPL   $BF96

; and exit to BASIC prompt
BF9F-   4C 00 E0    JMP   $E000
BFA2-  ["PLEASE REBOOT"]
```

...which is exactly the behavior I saw from my failed EDD bit copy. This crazy code -- and the crazy bit sequence on track $0E that it's verifying -- were designed to fool the best bit copiers of the day.

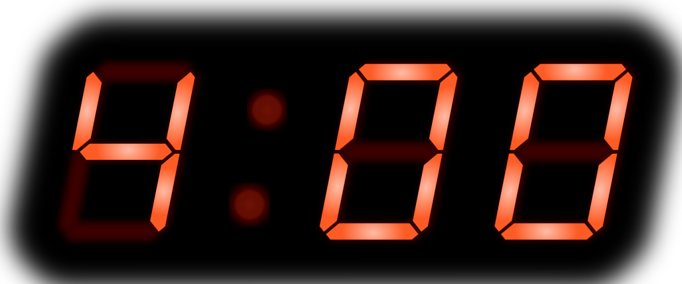
However, the protection check is almost completely separate from the calling code. There's no integration at all; the subroutine at $BEAF either returns gracefully, fails catastrophically, or loops infinitely. That was a lot of adverbs, but the point is that I don't need to patch the protection check at all. I just need to make sure we never call it.

Turning to my trusty Disk Fixer sector editor, I can search for "20 AF BE" (the "JSR $BEAF" code) and change the JSR instruction to BIT -- essentially a no-op -- to bypass the protection.

```
T04,S0A,$47: 20 -> 2C

]PR#6
...works...
```

Quod erat liberandum.



A 4am crack No. 898
------------------EOF------------------