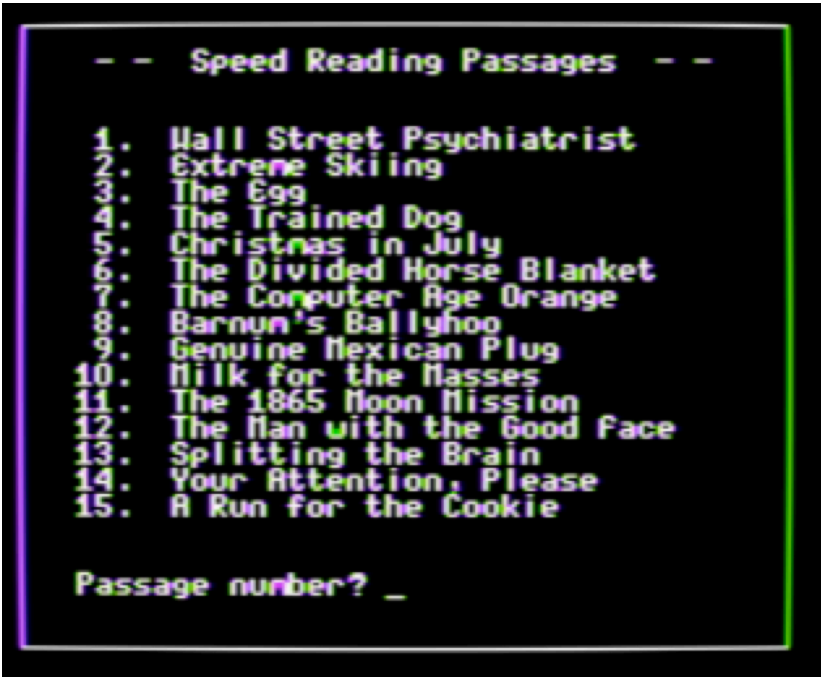
Speed Reader II

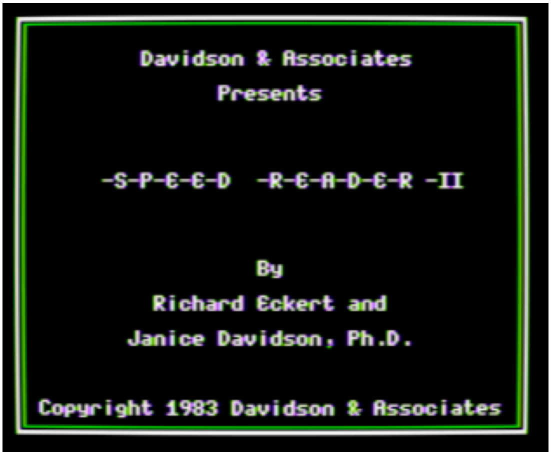


```
 - -  Speed Reading Passages  - -

   1.  Wall Street Psychiatrist
   2.  Extreme Skiing
   3.  The Egg
   4.  The Trained Dog
   5.  Christmas in July
   6.  The Divided Horse Blanket
   7.  The Computer Age Orange
   8.  Barnum's Ballyhoo
   9.  Genuine Mexican Plug
  10.  Milk for the Masses
  11.  The 1865 Moon Mission
  12.  The Man with the Good Face
  13.  Splitting the Brain
  14.  Your Attention, Please
  15.  A Run for the Cookie


  Passage number? _
```

2016-03-04

4am
to deprotect
and preserve

Contents



Davidson & Associates
Presents

-S-P-E-E-D -R-E-A-D-E-R -II

By
Richard Eckert and
Janice Davidson, Ph.D.

Copyright 1983 Davidson & Associates

```
------------Speed Reader II------------
A 4am crack                 2016-03-04
-------------------. updated 2016-10-07
                   |___________________
```

```
Name: Speed Reader II
Version: 09.12.86
Genre: educational
Year: 1986 according to disk label
      1983 according to title screen
Authors: Richard Eckert and Janice
  Davidson, Ph.D.
Publisher: Davidson & Associates, Inc.
Media: single-sided 5.25-inch floppy(*)
OS: DOS 3.3 variant ("Protected.DOS")
Previous cracks: none of this version
Similar cracks:
  #539 Find the Rhyme
  #413 The Spelling Machine
  #141 Word Attack
  #138 Spell It
  #131 Classmate
  #123 Outdoor Safety
  #122 Home Alone
  #121 Math Blaster
```

(*) There is a separate data disk that comes with the program disk, but mine was too badly damaged to be recovered. The program is usable without the data disk. I've cracked a previous version, Speed Reader II 08.05.84, which includes a compatible data disk, but I don't know if there were any updates to the data itself.

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)
  no errors, but the copy grinds and
  crashes

Copy ][+ nibble editor
  modified address epilogues (AF FF FF)
  modified address prologues, seem to
  rotate through a sequence
    T01 -> "D5 AA 97"
    T02 -> "D7 AA 96"
    T03 -> "D7 AA 97"
    T04 -> "D5 AA 96"
  then the cycle repeats

  modified data prologues on T02+
    ("D5 AA B5" instead of "D5 AA AD")

Disk Fixer
  everything seems encrypted/garbled

That could be the result of a non-
standard nibble translate table, or it
could be extra code in the RWTS that
decrypts sectors based on some key that
is only accessible on the original
disk. (I've seen it done both ways.)
```

Why didn't COPYA work?
  modified prologues/epilogues

Why didn't Locksmith FDB work?
  modified prologues/epilogues

Why didn't my EDD copy work?
  I don't know. Probably a nibble check
  during boot.

Next steps:

1. capture RWTS with AUTOTRACE
2. convert disk to standard format
   with Advanced Demuffin
3. patch RWTS to read standard format

Chapter 1
In Which We Find
A Most Unwelcoming Bootloader

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

]PR#5
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0
/!\ BOOT0 JUMPS TO $B6F0
CAPTURING BOOT1
...reboots slot 6...
...reboots slot 5...
SAVING BOOT1
/!\ BOOT1 IS ENCRYPTED
DECRYPTING BOOT1
SAVING BOOT1
```

Lots going on here. I'll take it one step at a time.

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

*801L
.
. all normal, until...
.
084A-   4C C0 B6    JMP   $B6F0
```

My AUTOTRACE program warned me about this -- a little something extra before the boot1 code. I don't like extra. Extra is bad.

In a normal DOS 3.3 disk, the code on T00,S00 is actually loaded twice: once at $0800 and then again at $B600, where it remains in memory until you reboot or do something to intentionally wipe it out. So I can see what's going to be at $B6F0 by looking at $08F0.

```
*8F0L

; odd
08F0-   A9 AA        LDA   #$AA
08F2-   85 31        STA   $31

; odd x2
08F4-   A9 AD        LDA   #$AD
08F6-   85 4E        STA   $4E

; suspicious (since this code is loaded
at $B600, this will overwrite the $AA
byte in the LDA instruction above)
08F8-   8D F1 B6     STA   $B6F1

; continue with boot1
08FB-   4C 00 B7     JMP   $B700
```

I'm pretty sure I know why boot0 is setting seemingly random zero page locations. (I've seen this before on other disks.) But I won't be able to verify it until I get a bit further down the rabbit hole.

The next part of AUTOTRACE's output is exciting(*), because I added all this automation then used it twice and never found another disk that used the same protection. Until now!

```
]CATALOG

C1983 DSR^C#254
280 FREE

 A 015 HELLO
 B 003 AUTOTRACE
 B 024 ADVANCED DEMUFFIN 1.5
 T 147 ADVANCED DEMUFFIN 1.5 DOCS
 B 003 BOOT0
 B 012 BOOT1 ENCRYPTED
 B 012 BOOT1
```

My AUTOTRACE program has captured two copies of the boot1 code. One is encrypted; the other is not.

```
]BLOAD BOOT1 ENCRYPTED,A$2600
]CALL -151

*B700<2700.27FFM
```

(*)not guaranteed, excitement may vary

```
*B700L

B700-   A0 1A       LDY   #$1A
B702-   B9 00 B7    LDA   $B700,Y
B705-   49 D9       EOR   #$D9
B707-   99 00 B7    STA   $B700,Y
B70A-   C8          INY
B70B-   D0 F5       BNE   $B702
B70D-   EE 04 B7    INC   $B704
B710-   EE 09 B7    INC   $B709
B713-   AD 09 B7    LDA   $B709
B716-   C9 C0       CMP   #$C0
B718-   D0 E8       BNE   $B702
B71A-   57          ???
B71B-   30 6E       BMI   $B78B
B71D-   57          ???
B71E-   2E 6E 70    ROL   $706E
B721-   B2          ???
B722-   54          ???
B723-   2B          ???
B724-   DA          ???
B725-   70 6E       BVS   $B795
```

The first thing that boot1 does is decrypt the rest of boot1. Everything from $B71A..$BFFF is encrypted with a simple XOR key, given in $B706. I've seen this pattern before (in "Math Blaster" and "Bingo Bugglebee Presents Home Alone," just to name two), so I added support for it in AUTOTRACE. Here is the code:

```
*3D0G

]FP
]LOAD HELLO
```

```
]LIST 200,250

 200  REM  BOOT1 WAS CAPTURED, NO
     W SAVE IT
 205  PRINT "SAVING BOOT1"
 210  PRINT  CHR$ (4)"BSAVE BOOT1
     ,A$2000,L$A00"
 211 KEY = 0: GOSUB 1300: IF KEY =
     0 THEN 220
 212  PRINT "/!\ BOOT1 IS ENCRYPT
     ED": PRINT "DECRYPTING BOOT1
     "
 213  POKE 38826,KEY: CALL 38820
 214  PRINT  CHR$ (4)"RENAME BOOT
     1,BOOT1 ENCRYPTED"
 215  PRINT "SAVING BOOT1"
 216  PRINT  CHR$ (4)"BSAVE BOOT1
     ,A$2000,L$A00"
.
.
.
 1300  REM   CHECK FOR SIMPLE DEC
     RYPTION LOOP AT $B700
 1301  REM   (KEY<>0 ON EXIT IF F
     OUND)
 1310 KEY = 0
 1320  IF  PEEK (8448) <  > 160 THEN
      RETURN
 1321  IF  PEEK (8449) <  > 26 THEN
      RETURN
 1322  IF  PEEK (8450) <  > 185 THEN
      RETURN
 1333  IF  PEEK (8451) <  > 0 THEN
      RETURN
 1334  IF  PEEK (8452) <  > 183 THEN
      RETURN
 1335  IF  PEEK (8453) <  > 73 THEN
      RETURN
 1340 KEY =  PEEK (8454): RETURN
```

The subroutine at line 1300 checks the first six bytes of the boot1 code (in memory at $2100 at this point) for the sequence "A0 1A B9 00 B7 49". The next byte would be the decryption key (part of the EOR instruction).

The actual decryption is part of the AUTOTRACE binary. Line 213 POKEs the decryption key into memory and CALLs the decryption routine at $97A4.

```
97A4-   A0 1A       LDY   #$1A

; $B700 from disk is at $2100 right now
97A6-   B9 00 21    LDA   $2100,Y

; decryption key POKEd from line 213
97A9-   49 FF       EOR   #$FF
97AB-   99 00 21    STA   $2100,Y
97AE-   C8          INY
97AF-   D0 F5       BNE   $97A6
97B1-   EE A8 97    INC   $97A8
97B4-   EE AD 97    INC   $97AD
97B7-   AD AD 97    LDA   $97AD
97BA-   C9 2A       CMP   #$2A
97BC-   D0 E8       BNE   $97A6
97BE-   60          RTS
```

And there you have it: automatic decryption of encrypted boot1 code.

Kick. Ass.

But I still don't have an RWTS file. Let's look at the (now decrypted) boot1 code and see what's going on.

Chapter 2
Beware of False Prophets And Boot Sectors

```
]BLOAD BOOT1,A$2600
]CALL -151

*B700<2700.27FFM
*B700L

; decryption loop is untouched
B700-   A0 1A       LDY   #$1A
B702-   B9 00 B7    LDA   $B700,Y
B705-   49 D9       EOR   #$D9
B707-   99 00 B7    STA   $B700,Y
B70A-   C8          INY
B70B-   D0 F5       BNE   $B702
B70D-   EE 04 B7    INC   $B704
B710-   EE 09 B7    INC   $B709
B713-   AD 09 B7    LDA   $B709
B716-   C9 C0       CMP   #$C0
B718-   D0 E8       BNE   $B702

; decrypted code starts here
B71A-   8E E9 B7    STX   $B7E9
B71D-   8E F7 B7    STX   $B7F7

; unfriendly reset vector
B720-   A9 6B       LDA   #$6B
B722-   8D F2 03    STA   $03F2
B725-   A9 B7       LDA   #$B7
B727-   8D F3 03    STA   $03F3
B72A-   49 A5       EOR   #$A5
B72C-   8D F4 03    STA   $03F4
B72F-   EA          NOP
```

```
; more RWTS parameters (normal)
B730-   A9 01       LDA   #$01
B732-   8D F8 B7    STA   $B7F8
B735-   8D EA B7    STA   $B7EA
B738-   AD E0 B7    LDA   $B7E0
B73B-   8D E1 B7    STA   $B7E1
B73E-   A9 02       LDA   #$02
B740-   8D EC B7    STA   $B7EC
B743-   A9 04       LDA   #$04
B745-   8D ED B7    STA   $B7ED
B748-   AC E7 B7    LDY   $B7E7
B74B-   88          DEY
B74C-   8C F1 B7    STY   $B7F1
B74F-   A9 01       LDA   #$01
B751-   8D F4 B7    STA   $B7F4
B754-   8A          TXA
B755-   4A          LSR
B756-   4A          LSR
B757-   4A          LSR
B758-   4A          LSR
B759-   AA          TAX
B75A-   A9 00       LDA   #$00
B75C-   9D F8 04    STA   $04F8,X
B75F-   9D 78 04    STA   $0478,X

; multi-sector read routine (normal)
B762-   20 93 B7    JSR   $B793

; reset stack (normal)
B765-   A2 FF       LDX   #$FF
B767-   9A          TXS

; slightly odd (usually $9D84 is the
; boot2 entry point, but OK)
B768-   4C 82 9D    JMP   $9D82
```

That all looks relatively normal. I
don't see anything that would explain
why my copy is hanging. It's not
grinding, and it's not rebooting. If
the RWTS was trying to read the disk
and failing, the disk drive would be
grinding. (You know what that sounds
like.) But it's just hanging, like it's
in an infinite loop somewhere. That is
most likely intentional, like a nibble
check that retries infinitely. Or maybe
a nibble check that gives up and fails
by going into an infinite loop with the
drive motor still on.

Let's follow the white rabbit, starting
at $B793, the entry point for the
multi-sector read routine.

```
*B793L

; this is not normal
B793-   4C 00 B8    JMP   $B800
```

```
; but the rest of the loop looks
; entirely normal
B796-    AD E4 B7     LDA   $B7E4
B799-    20 B5 B7     JSR   $B7B5
B79C-    AC ED B7     LDY   $B7ED
B79F-    88           DEY
B7A0-    10 07        BPL   $B7A9
B7A2-    A0 0F        LDY   #$0F
B7A4-    EA           NOP
B7A5-    EA           NOP
B7A6-    CE EC B7     DEC   $B7EC
B7A9-    8C ED B7     STY   $B7ED
B7AC-    CE F1 B7     DEC   $B7F1
B7AF-    CE E1 B7     DEC   $B7E1
B7B2-    D0 DF        BNE   $B793
B7B4-    60           RTS
```

Down the rabbit hole we go...

```
*B800L

; Hmm, the first thing this routine
; does is restore the code that should
; have been at $B793 (but wasn't,
; because it jumped here instead).
; Which tells me that this is designed
; to be run exactly once, during boot,
; the first time anything uses the
; multi-sector read routine at $B793.
B800-    A9 AC        LDA   #$AC
B802-    8D 93 B7     STA   $B793
B805-    A9 E5        LDA   #$E5
B807-    8D 94 B7     STA   $B794
B80A-    A9 B7        LDA   #$B7
B80C-    8D 95 B7     STA   $B795
B80F-    A9 07        LDA   #$07
B811-    85 4F        STA   $4F
```

```
; oh look, we're turning on the drive
; motor manually
B813-   AE E9 B7     LDX   $B7E9
B816-   BD 8D C0     LDA   $C08D,X
B819-   BD 8E C0     LDA   $C08E,X
B81C-   10 12        BPL   $B830

; do something (below)
B81E-   20 3E B8     JSR   $B83E
B821-   8D 00 02     STA   $0200

; do it again
B824-   20 3E B8     JSR   $B83E

; got the same result?
B827-   CD 00 02     CMP   $0200

; apparently "no" is the correct answer
B82A-   D0 0F        BNE   $B83B

; try again
B82C-   C6 4F        DEC   $4F
B82E-   D0 F4        BNE   $B824

; give up
B830-   A9 08        LDA   #$08
B832-   8D 7A B7     STA   $B77A
B835-   8D F4 03     STA   $03F4

; jump to The Badlands
B838-   4C 6B B7     JMP   $B76B

; success path ($B82A branches here) --
; continue to real multi-sector read
; routine
B83B-   4C 93 B7     JMP   $B793
```

```
; main subroutine starts here -- looks
; for the standard address prologue
B83E-   AE E9 B7    LDX   $B7E9
B841-   BD 8C C0    LDA   $C08C,X
B844-   10 FB       BPL   $B841
B846-   C9 D5       CMP   #$D5
B848-   D0 F7       BNE   $B841
B84A-   EA          NOP
B84B-   EA          NOP
B84C-   BD 8C C0    LDA   $C08C,X
B84F-   10 FB       BPL   $B84C
B851-   C9 AA       CMP   #$AA
B853-   D0 F1       BNE   $B846
B855-   EA          NOP
B856-   EA          NOP
B857-   BD 8C C0    LDA   $C08C,X
B85A-   10 FB       BPL   $B857
B85C-   C9 96       CMP   #$96
B85E-   D0 E1       BNE   $B841
B860-   48          PHA
B861-   68          PLA

; skips over the first half of the
; address field
B862-   A0 04       LDY   #$04
B864-   BD 8C C0    LDA   $C08C,X
B867-   10 FB       BPL   $B864
B869-   48          PHA
B86A-   68          PLA
B86B-   88          DEY
B86C-   D0 F6       BNE   $B864
```

```
; look for track number 0
B86E-   BD 8C C0    LDA   $C08C,X
B871-   10 FB       BPL   $B86E
B873-   C9 AA       CMP   #$AA
B875-   D0 CA       BNE   $B841
B877-   48          PHA
B878-   68          PLA

; look for sector number 0
B879-   BD 8C C0    LDA   $C08C,X
B87C-   10 FB       BPL   $B879
B87E-   C9 AA       CMP   #$AA
B880-   D0 BF       BNE   $B841

; skip the rest of the address field,
; then get the value of the raw nibble
; that follows
B882-   A0 05       LDY   #$05
B884-   BD 8C C0    LDA   $C08C,X
B887-   10 FB       BPL   $B884
B889-   48          PHA
B88A-   68          PLA
B88B-   88          DEY
B88C-   D0 F6       BNE   $B884
B88E-   60          RTS
```

Aha! The original disk has two address fields for T00,S00. One of them is the start of the actual sector data. The other one is a decoy that has an address field but no data field. The raw nibbles immediately following the two address prologues are different, and this routine checks to ensure that they are different.

The routine in the disk controller ROM (usually at $C65C) that looks for track 0 sector 0 will ignore the decoy if it happens to find it before the real one. (Technically, it will look for the data field, not find it in a reasonable time frame, and start over, and eventually it will find the real address field as the disk continues to spin.) This decoy is apparently enough to fool bit copy programs.

This is all very interesting -- and it explains why my bit copy would just hang during boot -- but it doesn't get me any closer to understanding this disk's custom RWTS.

Let's back up.

Chapter 3
In Which Everything Is Terrible

```
*B793L

B793-   4C 00 B8    JMP   $B800
B796-   AD E4 B7    LDA   $B7E4
B799-   20 B5 B7    JSR   $B7B5
```

Ignoring the JMP for the moment, the multi-sector read routine calls the standard $B7B5 entry point to actually read a single sector.

```
*B7B5L

; this is normal
B7B5-   08          PHP
B7B6-   78          SEI

; definitely not normal (usually $BD00)
B7B7-   20 00 BA    JSR   $BA00

; the rest is all normal
B7BA-   B0 03       BCS   $B7BF
B7BC-   28          PLP
B7BD-   18          CLC
B7BE-   60          RTS
B7BF-   28          PLP
B7C0-   38          SEC
B7C1-   60          RTS
```

That explains why I couldn't find the RWTS code I expected in the location I expected. This RWTS is laid out completely differently in memory than the standard DOS 3.3 RWTS. Even the entry point is different ($BA00 instead of $BD00).

```
*BA00L

BA00-   85 48        STA   $48
BA02-   84 49        STY   $49
BA04-   A0 02        LDY   #$02
BA06-   8C F8 06     STY   $06F8
BA09-   A0 04        LDY   #$04
BA0B-   8C F8 04     STY   $04F8
BA0E-   A0 01        LDY   #$01
BA10-   B1 48        LDA   ($48),Y
BA12-   AA           TAX
BA13-   A0 0F        LDY   #$0F
BA15-   D1 48        CMP   ($48),Y
BA17-   F0 1B        BEQ   $BA34
```

Yup, that looks like an RWTS entry point.

Oh, and remember that weird code at $B6F0 that set two zero page locations for no apparent reason? Here's the reason: the RWTS uses them. (I've seen this pattern before, too.) After seconds of furious investigation, I found the RWTS code that looks for the data prologue:

```
*BDE1L

BDE1-   BD 8C C0     LDA   $C08C,X
BDE4-   10 FB        BPL   $BDE1
BDE6-   49 D5        EOR   #$D5
BDE8-   D0 F4        BNE   $BDDE
BDEA-   BD 8C C0     LDA   $C08C,X
BDED-   10 FB        BPL   $BDEA
BDEF-   C5 31        CMP   $31       <-- !
BDF1-   D0 F3        BNE   $BDE6
BDF3-   A0 56        LDY   #$56
BDF5-   BD 8C C0     LDA   $C08C,X
BDF8-   10 FB        BPL   $BDF5
[...]
```

```
BDFA-   C5 4E          CMP   $4E     <-- !
BDFC-   D0 E8          BNE   $BDE6
```

And there it is, in living color: this RWTS uses two magic zero page values to find the data prologue while it's reading a sector from disk.

Why? Because f--- you, that's why. Because it makes the extracted RWTS useless without initializing the magic zero page location with the right magic number. Automated RWTS extraction programs wouldn't find this. If I load this RWTS into Advanced Demuffin, it will not be able to read the original disk, because the RWTS itself is not what initializes the magic zero page location.

I can save this RWTS into a separate file, but I won't be able to use it in Advanced Demuffin without an IOB module. See the Advanced Demuffin documentation on my work disk for all the gory details about IOB modules. Basically, Advanced Demuffin only knows how to call a custom RWTS if it

1. is loaded at $B800..$BFFF

2. uses a standard RWTS parameter table

3. has an entry point at $BD00 that takes the address of the parameter tables in A and Y

4. doesn't require initialization

As it turns out, that covers a *lot* of copy protected disks, but it doesn't cover this one. This disk fails assumption #3 (the entry point is at $BA00, not $BD00) and #4 (the RWTS relies on the values of zero page $31 and $4E, which are initialized outside the RWTS).

So, let's make an IOB module.

```
; Most of this is identical to the
; standard IOB module that comes with
; Advanced Demuffin
1400-   4A              LSR
1401-   8D 22 0F        STA   $0F22
1404-   8C 23 0F        STY   $0F23
1407-   8E 27 0F        STX   $0F27
140A-   A9 01           LDA   #$01
140C-   8D 20 0F        STA   $0F20
140F-   8D 2A 0F        STA   $0F2A

; initialize the magic zero page values
1412-   A9 AA           LDA   #$AA
1414-   85 31           STA   $31
1416-   A9 AD           LDA   #$AD
1418-   85 4E           STA   $4E

; get the address of the RWTS parameter
; table at $0F1E and call the RWTS at
; its non-standard entry point, $BA00
141A-   A9 0F           LDA   #$0F
141C-   A0 1E           LDY   #$1E
141E-   4C 00 BA        JMP   $BA00
```

Wait wait wait... I've made this mistake before. This IOB module won't work. Advanced Demuffin will crash. Learn from your mistakes so you have the opportunity to make interesting new ones.

I'll explain. Let's back up.

```
*B793L

B793-   4C 00 B8    JMP   $B800
B796-   AD E4 B7    LDA   $B7E4
B799-   20 B5 B7    JSR   $B7B5
```

That "JMP $B800" instruction gets replaced immediately at $B800.

```
B800-   A9 AC       LDA   #$AC
B802-   8D 93 B7    STA   $B793
B805-   A9 E5       LDA   #$E5
B807-   8D 94 B7    STA   $B794
B80A-   A9 B7       LDA   #$B7
B80C-   8D 95 B7    STA   $B795
```

So, the routine at $B793 ends up looking like this:

```
B793-   AC E5 B7    LDY   $B7E5
B796-   AD E4 B7    LDA   $B7E4
B799-   20 B5 B7    JSR   $B7B5
```

Perfectly ordinary, no? Actually, no. Here's what it looks like on an ordinary (unprotected) DOS 3.3 disk.

```
B793-   AD E5 B7    LDA   $B7E5
B796-   AC E4 B7    LDY   $B7E4
B799-   20 B5 B7    JSR   $B7B5
```

Spot the difference. Go ahead, I'll
wait.

A and Y get passed through to the RWTS
entry point, which is usually at $BD00
but on this disk is at $BA00.

DOS 3.3 disk:

```
*BD00L

BD00-   84 48       STY   $48
BD02-   85 49       STA   $49
```

This disk:

```
*BA00L

BA00-   85 48       STA   $48
BA02-   84 49       STY   $49
```

Now do you see it? On a normal disk,
the Y register holds the low byte of
the RWTS parameter table address, and
the accumulator holds the high byte.
But on this disk, those are reversed;
the accumulator holds the low byte, and
the Y register holds the high byte.

Why? Because f--- you, that's why.

Of course, the IOB module I created to
interface with this RWTS was still
putting the low byte in Y and the high
byte in A, so the RWTS was reading a
completely bogus parameter table and
God only knows what happened next.
(Thank goodness the original disk was
write-protected.)

I need to make one little change to my
IOB module.

```
1400-   4A          LSR
1401-   8D 22 0F    STA   $0F22
1404-   8C 23 0F    STY   $0F23
1407-   8E 27 0F    STX   $0F27
140A-   A9 01       LDA   #$01
140C-   8D 20 0F    STA   $0F20
140F-   8D 2A 0F    STA   $0F2A
1412-   A9 AA       LDA   #$AA
1414-   85 31       STA   $31
1416-   A9 AD       LDA   #$AD
1418-   85 4E       STA   $4E
141A-   A0 0F       LDY   #$0F ; Y=high
141C-   A9 1E       LDA   #$1E ; A=low
141E-   4C 00 BA    JMP   $BA00
```

```
*BSAVE IOB,A$1400,L$FB
```

Now let's go.

```
*BRUN ADVANCED DEMUFFIN 1.5
```

[press "5" to switch to slot 5]

[press "R" to load a new RWTS module]
  --> At $B6, load "BOOT1" from drive 1

[press "I" to load a new IOB module]
  --> load "IOB" from drive 1

[press "6" to switch to slot 6]

[press "C" to convert disk]

```
                 --v--

ADVANCED DEMUFFIN 1.5    (C) 1983, 2014
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======================================
TRK:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
+.5:
    0123456789ABCDEF0123456789ABCDEF012
SC0:...RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC1:...RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC2:...RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC3:...RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC4:...RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC5:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC6:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC7:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC8:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SC9:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SCA:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SCB:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SCC:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SCD:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SCE:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SCF:..RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
=======================================
16SC $00,$00-$22,$0F BY1.0 S6,D1->S6,D2

                 --^--
```

Make no mistake: this is definitely
progress. I have converted a little
more than two tracks, which means that
the RWTS I extracted *can* read (at
least part of) the disk, and the IOB
module I created *can* call the RWTS
correctly. But this combination only
works from T00,S00 to T02,S04.

Chapter 4
In Which We Witness The Power Of This Fully Armed And Operational RWTS

That track/sector sounds suspiciously
familiar. It's the last sector of DOS,
and it's the first sector read by the
boot1 code.

```
; relevant boot1 code
B73E-   A9 02         LDA   #$02
B740-   8D EC B7      STA   $B7EC
B743-   A9 04         LDA   #$04
B745-   8D ED B7      STA   $B7ED
```

After DOS is loaded, I guess the RWTS
is modified to look for a different
data epilogue sequence. But remember,
the third byte of the data epilogue is
stored in zero page $4E (initially set
up at $B6F0). So the DOS doesn't even
need to modify the RWTS code directly;
it just changes zero page $4E.

Turning to the Copy][+ nibble editor,
it appears that every sector from
T02,S05 to T22,S0F uses "D5 AA B5" as
the data prologue.

```
                 --v--

TRACK: 03  START: 34C5  LENGTH: 015F
       ^^

34A0: DB DB DB DB DB DB DB DB   VIEW
34A8: DB DB DB DB DB DB DB DB
34B0: DB DB DB DB DB DB DB DB
34B8: DB DB DB DB DB DB DB DB
34C0: DB DB DB DB DB D7 AA 97  <-34C5
                     ^^^^^^^^
                 address prologue

34C8: AA AA AB AB AA AA AB AB
34D0: AF FF FF FF FF FF FF FF
      ^^^^^^^^
  address epilogue

34D8: FF D5 AA B5 D5 D7 D6 97
         ^^^^^^^^
      data prologue

34E0: 96 D5 97 D5 97 96 9A D5

                 --^--
```

(Unrelated to my current task, but notice how the disk is using $DB as a sync byte instead of $FF. That confused early nibble copiers. They threw Every Single Trick they knew into this copy protection scheme.)

Anyway, I need another IOB module.

```
]PR#5
...
]BLOAD IOB,A$1400
]CALL -151

*1417:B5

*1400L

1400-   4A          LSR
1401-   8D 22 0F    STA   $0F22
1404-   8C 23 0F    STY   $0F23
1407-   8E 27 0F    STX   $0F27
140A-   A9 01       LDA   #$01
140C-   8D 20 0F    STA   $0F20
140F-   8D 2A 0F    STA   $0F2A
1412-   A9 AA       LDA   #$AA
1414-   85 31       STA   $31
1416-   A9 B5       LDA   #$B5   ; new
1418-   85 4E       STA   $4E
141A-   A0 0F       LDY   #$0F
141C-   A9 1E       LDA   #$1E
141E-   4C 00 BA    JMP   $BA00

*BSAVE IOB 3+,A$1400,L$FB

[S6,D1=original disk]
[S6,D2=partially demuffin'd disk]
[S5,D1=my work disk]
```

```
*BRUN ADVANCED DEMUFFIN 1.5

[press "5" to switch to slot 5]

[press "R" to load a new RWTS module]
  --> At $B6, load "BOOT1" from drive 1

[press "I" to load a new IOB module]
  --> load "IOB 3+" from drive 1

[press "6" to switch to slot 6]

[press "C" to convert disk]

[press "Y" to change default values]

                 --v--

INPUT ALL VALUES IN HEX


SECTORS PER TRACK? (13/16) 16

START TRACK: $02        <-- change this
START SECTOR: $05       <-- change this

END TRACK: $22
END SECTOR: $0F

INCREMENT: 1

MAX # OF RETRIES: 0

COPY FROM DRIVE 1
TO DRIVE: 2
=======================================
16SC $02,$05-$22,$0F BY$01 S6,D1->S6,D2

                 --^--
```

And here we go...

```
                 --v--

TRK:  .................................
+.5:
    0123456789ABCDEF0123456789ABCDEF012
SC0:   ................................
SC1:   ................................
SC2:   ................................
SC3:   ................................
SC4:   ................................
SC5:  .................................
SC6:  .................................
SC7:  .................................
SC8:  .................................
SC9:  .................................
SCA:  .................................
SCB:  .................................
SCC:  .................................
SCD:  .................................
SCE:  .................................
SCF:  .................................
=======================================
16SC $02,$05-$22,$0F BY$01 S6,D1->S6,D2

                 --^--
```

This is the power and the genius of
Advanced Demuffin. Every disk must be
able to read itself. So, let it read
itself, then capture the data and write
it out in a standard format.

```
]PR#5
...
]CATALOG,S6,D2

C1983 DSR^C#254
203 FREE

 A 060 HELLO
 I 006 APPLESOFT
 B 042 FPBASIC
 T 002 SPEED 1.OBJ
 T 002 SPEED 2.OBJ
 A 030 SPEED 1
 A 046 SPEED 2
 A 058 SPEED READER II DEMONSTRATION
 A 047 SPEED READER II EDITOR

]RUN HELLO

ERROR #6 FILE NOT FOUND
```

Wait, what?

Chapter 5
The DOS Strikes Back

Firing up Disk Fixer and pointing it to
my newly demuffin'd copy, I see the
problem: all of the files on this disk
have control characters in their names.

```
                 --v--
TRACK $11/SECTOR $0F/VOLUME $FE/BYTE$00
---------------------------------------
$00:>00<11 0E 00 00 00 00 00    @QN@@@@@
$08: 00 00 00 12 0F 02 C8 9A    @@@ROBH.
                          ^^
                        Ctrl-Z

$10: C5 CC CC CF A0 A0 A0 A0    ELLO
$18: A0 A0 A0 A0 A0 A0 A0 A0
$20: A0 A0 A0 A0 A0 A0 A0 A0
$28: A0 A0 A0 A0 3C 00 16 0F        <@VO
$30: 01 C1 9A D0 D0 CC C5 D3    AA.PPLES
           ^^
         Ctrl-Z

$38: CF C6 D4 A0 A0 A0 A0 A0    OFT
$40: A0 A0 A0 A0 A0 A0 A0 A0
$48: A0 A0 A0 A0 A0 A0 A0 06           F
$50: 00 17 0F 04 C6 9A D0 C2    @WODF.PB
                    ^^
                  Ctrl-Z

$58: C1 D3 C9 C3 A0 A0 A0 A0    ASIC
$60: A0 A0 A0 A0 A0 A0 A0 A0
$68: A0 A0 A0 A0 A0 A0 A0 A0
$70: A0 A0 2A 00 18 0F 00 D3      *@XO@S
$78: 9A D0 C5 C5 C4 A0 B1 AE     .PEED 1.
     ^^
   Ctrl-Z

---------------------------------------
BUFFER 0/SLOT 6/DRIVE 2/MASK OFF/NORMAL
                 --^--
```

OK, one thing at a time. I have a non-bootable disk with a standard disk catalog and what appear to be standard, though awkwardly named, files. So let's put a standard DOS on this puppy. I'm not even going to try to patch the DOS from the original disk. The sooner I can forget about that DOS, the better.

Using Copy][+, I can "copy DOS" from a freshly initialized DOS 3.3 disk onto the demuffin'd copy. This function of Copy][+ just sector-copies tracks 0-2 from one disk to another, but it's easier than setting that up manually in some other copy program.

```
Copy ][+
  --> COPY
    --> DOS
      --> from slot 6, drive 2
      -->   to slot 6, drive 1

[S6,D1=demuffin'd copy]
[S6,D2=newly formatted DOS 3.3 disk]

...read read read...
...write write write...
```

Now I need to change the boot program to "H<Ctrl-Z>ELLO". This feature of Copy][+ just presents a list interface to choose a file from the catalog, then sector-edits T01,S09 to set the name of the program that DOS runs (instead of "HELLO" without the control character).

```
Copy ][+
  --> CHANGE BOOT PROGRAM
    --> on slot 6, drive 1
      --> H<Ctrl-Z>ELLO
```

The catalog listing doesn't actually show the control character, so it looks like I'm changing the boot program from "HELLO" to "HELLO". But it does make the necessary changes.

Rebooting loads DOS (of course, I just put it there), appears to load the H<Ctrl-Z>ELLO program successfully... then immediately reboots.

There is more copy protection.

Chapter 6
Maybe Yes, Maybe No, Maybe Go F--- Yourself

```
]PR#6
...
<Ctrl-C>

BREAK
]LIST

 10  POKE 104,32: RUN
 65535  REM COPYRIGHT 1983
 65535  REM DAVIDSON & ASSOCIATES
```

According to the framed Beagle Bros. "Peeks, Pokes and Pointers" chart that hangs above my desk and reminds me that technical writing should be wondrous, useful, and fun (but not always in that order), zero page 104 ($68) is the high byte of the starting address of the Applesoft BASIC program in memory. Which means that this HELLO program contains an entirely separate, entirely hidden BASIC program within it.

```
]POKE 104,32
]LIST

 10  REM
 400  REM  PEEK (40324) = 173 OR
      PEEK (47094) <  > 0 THEN 10
     00
 410  POKE 216,0: ONERR  GOTO 100
     0
 510  REM  --MUSIC ROUTINE INIT
 530  FOR THE = 768 TO 1000
 550  READ FIRE
 570  POKE THE,FIRE
 590  NEXT
 595  POKE 765,32: REM TIMBRE
 630  REM  -MUSIC ROUTINE DATA
 650  DATA  76, 55, 3, 164, 1, 17
     3, 48, 192, 230, 2, 208, 5,
     230, 3, 208, 5, 96, 234, 76,
      21, 3, 136, 240, 5
 670  DATA  76, 27, 3, 208, 235,
     164, 0, 173, 48, 192, 230, 2
     , 208, 5, 230, 3, 208, 5, 96
     , 234, 76, 47, 3, 136
 690  DATA  240, 209, 76, 53, 3,
     208, 235, 173, 255, 2, 10, 1
     68, 185, 127, 3, 133, 0, 173
     , 253, 2, 74, 240, 4, 70
 710  DATA  0, 208, 249, 185, 127
     , 3, 56, 229, 0, 133, 1, 200
     , 185, 127, 3, 101, 0, 133,
     0, 169, 0, 56, 237, 254
 730  DATA  2, 133, 3, 169, 0, 13
     3, 2, 165, 1, 208, 152, 234,
      234, 76, 112, 3, 230, 2, 20
     8, 5, 230, 3, 208, 5
[...]
```

```
750  DATA  96, 234, 76, 125, 3,
    208, 236, 0, 0, 246, 246, 23
    2, 232, 219, 219, 207, 207,
    195, 195, 184, 184, 174, 174
    , 164
770  DATA  164, 155, 155, 146, 1
    46, 138, 138, 130, 130, 123,
     123, 116, 116, 109, 110, 10
    3, 104, 97, 98, 92, 92, 87,
    87, 82
790  DATA  82, 77, 78, 73, 73, 6
    9, 69, 65, 65, 61, 62, 58, 5
    8, 54, 55, 51, 52, 48, 49, 4
    6, 46, 43, 44, 41
810  DATA  41, 38, 39, 36, 37, 3
    4, 35, 32, 33, 30, 31, 29, 2
    9, 27, 28, 26, 26, 24, 25, 2
    3, 23, 21, 22, 20
830  DATA  21, 19, 20, 18, 18, 1
    7, 17, 16, 16, 15, 16, 14, 1
    5, 255, 255, 255, 0
900  POKE 2049,104: POKE 2050,16
    8: POKE 2051,104: POKE 2052,
    166: POKE 2053,223: POKE 205
    4,154
910  POKE 2055,72: POKE 2056,152
    : POKE 2057,72: POKE 2058,96

920  PRINT  CHR$ (4);"OPEN SPEED
     1.OBJ": PRINT  CHR$ (4);"RE
    AD SPEED 1.OBJ": INPUT YES,N
    O,MAYBE,YY,ZZ: PRINT  CHR$ (
    4);"CLOSE SPEED 1.OBJ"
930  POKE YY,ZZ: GOTO 10
1000  CALL 54915: POKE 216,0: ONERR
      GOTO 1000
1010  PRINT  CHR$ (4);"PR#6"
1020  REM ---------------
```

But wait... there's more. I mean, there has to be more. Other than creating a little assembly language routine at 768 ($300), this program doesn't actually *do* anything. It doesn't even call the assembly language routine it creates. It pokes and pokes and... GOTO 10? How does that do, well, anything?

Line 911 reads a series of values from a text file ("SPEED 1.OBJ", although I'm pretty sure there are some control characters in there somewhere). Looks innocuous, until line 930 where you realize that it's using those values to POKE something. Using Copy][+'s "view file as text" function, here are the entire contents of "SPEED 1.OBJ":

--v--

```
8131
-936
6084
104
64
```

--^--

The first three values go into the variables YES, NO, and MAYBE. (Really.) The last two go into YY and ZZ, and that's what gets POKE'd in line 930.

Hey, poking address 104. That sounds familiar...

```
]POKE 104,64
]LIST

 10  CALL YES: CALL NO: HGR : CALL
     MAYBE: HCOLOR= 3: HPLOT 0,0 TO
     278,0 TO 278,191 TO 0,191 TO
     0,0: HPLOT 1,1 TO 277,1 TO 2
     77,190 TO 1,190 TO 1,1
 11  POKE 216,0: ONERR  GOTO 1000

 20  HCOLOR= 2: HPLOT 4,3 TO 274,
     3 TO 274,188 TO 4,188 TO 4,3

 40  HCOLOR= 3: HPLOT 56,44 TO 22
     0,44 TO 220,73 TO 56,73 TO 5
     6,44: HPLOT 52,41 TO 224,41 TO
     224,76 TO 52,76 TO 52,41
 50  VTAB 8: HTAB 14
 50  VTAB 8: HTAB 14
 60  PRINT "2SPEED READER II"
 62  VTAB 20: HTAB 8
 64  PRINT "2BE SURE CAPS LOCK IS
      DOWN"
 70  VTAB 22: HTAB 5
 80  PRINT "2PRESS D TO RUN THE D
     EMONSTRATION"
 90  VTAB 10
 100 P =  PEEK ( - 16384)
 110  IF P = 196 THEN  PRINT  CHR$
     (4);"RUN  SPEED READER II DE
     MONSTRATION"
 120  IF P = 197 THEN  PRINT  CHR$
     (4);"RUN SPEED READER II EDI
     TOR"
 130  PRINT  CHR$ (4);"RUN SPEED
     1"
[...]
```

```
 1000  CALL 54915: POKE 216,0: ONERR
      GOTO 1000
 1010  PRINT  CHR$ (4);"PR#6"
```

Un-freaking-believable. This BASIC program changes the starting memory address of the currently running BASIC program and re-runs itself. Twice.

Anyway, back to the... I don't even know what to call it. Back to the second program-within-a-program, I guess.

```
]POKE 104,32
]LIST 400

 400  IF  PEEK (40324) = 173 OR  PEEK
     (47094) <  > 0 THEN 1000
```

This is the problem.

40324 is $9D84, which (reaching waaay back to the beginning of this journey when I decrypted the boot1 code) is *not* the entry point to the boot2 code. On a standard DOS 3.3 disk, it is, but on this disk, the entry point is at $9D82 instead. So this line of BASIC is spot-checking the DOS in memory to ensure that we booted from the original non-standard DOS. (Hint: we didn't, because I just replaced that DOS with a standard DOS 3.3.)

It also checks 47094 ($B7F6), which is
part of the RWTS parameter table. On a
standard DOS 3.3 disk, this location
would be the actual volume number found
the last time the RWTS successfully
read a sector. Apparently the original
disk's RWTS (which, again, I just
replaced with a standard DOS 3.3 RWTS)
always sets it to 0 instead. Or maybe
the original disk had disk volume 0?
You can't create that with a normal
"INIT" command, but this disk is
anything but normal.

Let's see if I can skip past it...

```
]RUN 402
```

Success! The program loads and runs all
the way up to the main menu.

But how can I patch this program? It's
not even the real program; it's the
second-level program-within-a-program.
There's a program above it and another
program below it, all self-contained in
the same "A" type file. If I delete the
line, all of that will be ruined.

I'm going to have to hack the Applesoft
opcodes from the monitor.

```
]PR#6
...
<Ctrl-C>

]POKE 104,32
]CALL-151

*2000.202F

2000- 00 07 20 0A 00 B2 00 2A
2008- 20 90 01 AD E2 28 34 30
               ^^ ^^^^^ ^^^^^
               IF PEEK(  4  0

2010- 33 32 34 29 D0 31 37 33
      ^^^^^^^^^^^ ^^ ^^^^^^^^
       3  2  4  )  =  1  7  3

2018- CE E2 28 34 37 30 39 34
      ^^ ^^^^^ ^^^^^^^^^^^^^^
      OR PEEK(  4  7  0  9  4

2020- 29 D1 CF 30 C4 31 30 30
      ^^ ^^^^^ ^^ ^^ ^^^^^^^^
       )  <  > 0 THEN 1  0  0

2028- 30 00 3C 20 9A 01 B9 32
      ^^
       0
```

Looking at address $2005, it appears that the opcode for a "REM" statement is $B2. Let's try changing the "IF" statement to a "REM" statement.

```
*200B:B2

*3D0G        ; return to BASIC prompt

]LIST 400

 400  REM  PEEK (40324) = 173 OR
      PEEK (47094) <  > 0 THEN 10
     00
```

Success! Line 400 is now a comment and shouldn't do any harm. (Listing the rest of the code confirms that this hasn't disturbed the delicate balance of the three programs in memory.)

```
]RUN
```

Success! It runs without complaint.

Now to make this patch permanent. Turning to my trusty Copy][+ sector editor (version 5.5, the last version that can "follow" files), I press "F" to follow, select "HELLO" from the disk catalog listing, "S" to scan and "H" for hex. Searching for "34 30 33 32 34" (the string "40324" as it's represented in hex within an Applesoft program), I find it on T13,S06.

```
T13,S06,$0C change "AD" to "B2"
```

Success! The disk boots and loads with no complaint. That is, until -- and I am not making this up -- I select a game and try to play it. Then it reboots.

There is still more copy protection.

Chapter 7
I Like Garden Paths Am Delightful

I'm beginning to suspect that replacing the original DOS with a bog-standard copy of DOS 3.3 was a mistake. If there are going to be checks upon checks of subtle differences scattered throughout the program, perhaps I'd be better off trying to adapt the original DOS than replace it outright.

Backing up, I recreated the fully demuffin'd copy I had at the end of chapter 4. Now I have a disk that doesn't boot because it fails a nibble check at $B800, which I can't easily patch because most of the bootloader is encrypted.

Next steps:

1. Write decrypted bootloader to disk
2. Patch boot0 to skip decryption (since it's already decrypted)
3. Patch boot1 to skip nibble check
4. Patch RWTS to look for standard prologues and epilogues
5. Always set $B7F6 to 0 after every RWTS call
6. Maybe that's "all"?

Here we go.

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

```
; straightforward multi-sector write
; loop, via the RWTS vector at $03D9
08C0-   A9 08          LDA   #$08
08C2-   A0 E8          LDY   #$E8
08C4-   20 D9 03       JSR   $03D9
08C7-   AC ED 08       LDY   $08ED
08CA-   88             DEY
08CB-   10 05          BPL   $08D2
08CD-   A0 0F          LDY   #$0F
08CF-   CE EC 08       DEC   $08EC
08D2-   8C ED 08       STY   $08ED
08D5-   CE F1 08       DEC   $08F1
08D8-   CE E1 08       DEC   $08E1
08DB-   D0 E3          BNE   $08C0
08DD-   60             RTS

08E0- 00 0A 00 00 00 00 00 00
         ^^
    sector count

08E8- 01 60 01 00 00 09 FB 08
         ^^ ^^    ^^ ^^
         S6 D1    T0 S9

08F0- 00 2F 00 00 02 00 FE 60
      ^^^^^       ^^
     address    write

08F8- 01 00 00 00 01 EF D8 00

*BSAVE WRITE BOOT1,A$8C0,L$40
*BLOAD BOOT1,A$2600
*8C0G
...write write write...
```

Step 1 complete. Now I have a copy that crashes instantly because it's trying to decrypt a bootloader that's already decrypted.

To skip the decryption loop, I need to
change the JMP at the end of T00,S00,
which is called once it's loaded at
$B6F0:

```
B6F0-   A9 AA       LDA   #$AA
B6F2-   85 31       STA   $31
B6F4-   A9 AD       LDA   #$AD
B6F6-   85 4E       STA   $4E
B6F8-   8D F1 B6    STA   $B6F1
B6FB-   4C 00 B7    JMP   $B700   <-- !
```

To skip the decryption loop, I want to
jump to $B71A instead of $B700. Thus:

```
T00,S00,$FC change "00" to "1A"
```

Step 2 complete. Now I have a copy that
reboots endlessly after failing the
protection check at $B800. But at least
it no longer tries to decrypt an
already-decrypted bootloader, so I've
got that going for me, which is nice.

To bypass the protection check, I need
to restore the proper instruction at
$B793. The code at $B800 shows me what
to put there, because it does it before
even starting the protection check.

```
T00,S01,$93 change "4C 00 B8"
                to "AC E5 B7"
```

Step 3 complete. Now I have a copy that
loads DOS then grinds and reboots. Wait
a minute... I haven't patched the RWTS
yet. How can it even read DOS on tracks
$00-$02?

```
]PR#5
...
]BLOAD BOOT1,A$2600
]CALL -151

*FE89G FE93G
*B600<2600.2FFFM

.
. [manually scan through this RWTS
.  where nothing is in the right place]
.
```

Ah, here we go. The code to match the prologues and epilogues and convert between nibbles and bytes looks almost identical to standard DOS 3.3, but it starts at $BD00 instead of $B800.

```
; match data prologue
BDE1-   BD 8C C0    LDA   $C08C,X
BDE4-   10 FB       BPL   $BDE1
BDE6-   49 D5       EOR   #$D5
BDE8-   D0 F4       BNE   $BDDE
BDEA-   BD 8C C0    LDA   $C08C,X
BDED-   10 FB       BPL   $BDEA
BDEF-   C5 31       CMP   $31      <-- !
BDF1-   D0 F3       BNE   $BDE6
BDF3-   A0 56       LDY   #$56
BDF5-   BD 8C C0    LDA   $C08C,X
BDF8-   10 FB       BPL   $BDF5
BDFA-   C5 4E       CMP   $4E      <-- !
BDFC-   D0 E8       BNE   $BDE6
```

Instead of matching constants (#$AA and #$AD), we're comparing against the zero page values we set earlier in the boot. I knew about this, because I had to set them myself to get Advanced Demuffin to work.

Also, remember how all the sectors past
T02,S04 used a different data prologue
("D5 AA B5" instead of "D5 AA AD")? I
bet there's some logic somewhere after
the bootloader loads DOS that changes
the value of zero page $4E from #$AD to
#$B5.

But I still don't know why this RWTS is
able to load DOS at all. The address
prologues were all screwed up; they
rotated between 4 different values on
different tracks!

Going further through the RWTS, I
discovered the answer:

```
; match address prologue
BE4D-   BD 8C C0    LDA   $C08C,X
BE50-   10 FB       BPL   $BE4D
BE52-   29 D5       AND   #$D5     <-- !
BE54-   C9 D5       CMP   #$D5     <-- !
BE56-   D0 EE       BNE   $BE46
BE58-   BD 8C C0    LDA   $C08C,X
BE5B-   10 FB       BPL   $BE58
BE5D-   C5 31       CMP   $31
BE5F-   D0 F1       BNE   $BE52
BE61-   A0 03       LDY   #$03
BE63-   BD 8C C0    LDA   $C08C,X
BE66-   10 FB       BPL   $BE63
BE68-   29 96       AND   #$96     <-- !
BE6A-   C9 96       CMP   #$96     <-- !
BE6C-   D0 DF       BNE   $BE4D
```

This is really brilliant, but it may
require some bit math to understand.

Chapter 8
Bit Math Is Best Math

Recall the pattern of address prologues
on this disk:

```
    T01 -> "D5 AA 97"
    T02 -> "D7 AA 96"
    T03 -> "D7 AA 97"
    T04 -> "D5 AA 96"
```

Then the cycle repeats. (Track $00 is
also the standard "D5 AA 96", same as
track $04, $08, &c.)

The code to find prologue nibble #1
explains how this disk can read the
prologue on track $02 ("D7 AA 96").

Normal address prologue byte 1 is $D5.

```
$D5           = 1101 0101
$D5           = 1101 0101
                ---------
$D5 AND $D5 = 1101 0101 = $D5
```

Of course $D5 ANDed with itself is $D5.
No big surprise there. But track $02
uses $D7 instead of $D5 for the first
prologue nibble.

```
$D7           = 1101 0111
$D5           = 1101 0101
                ---------
$D7 AND $D5 = 1101 0101 = $D5
```

Because the only difference is a single
bit (the second from the right in this
diagram), and that bit is 0 in $D5, the
result is the same: $D5. The comparison
at $BE54 passes, and we move on with
the rest of the prologue.

Meanwhile, track $01 has a non-standard THIRD nibble ($97 instead of $96). But we're doing the same thing -- ANDing it and comparing it.

```
$96           1001 0110
$96           1001 0110
              ---------
$96 AND $96 = 1001 0110 = $96
```

Again, any value ANDed with itself is going to be itself. No big surprise.

```
$97           1001 0111
$96           1001 0110
              ---------
$97 AND $96 = 1001 0110 = $96
```

So the check of the third prologue is flexible, like the check of the first prologue. The RWTS doesn't care about the track number at all. It's totally willing to match $D5 or $D7 as the first prologue nibble on any track, and it's equally willing to match $96 or $97 as the third prologue nibble on any track. And since my demuffin'd copy has $D5 on every track and $96 on every track, the RWTS never complains. The standard address prologue is one of the combinations it supports.

Furthermore, RWTS code is time-critical between reading the last bit of one nibble and reading the first bit of the next. If it's too fast or too slow, it will get out of phase (because the disk spins independently of the CPU).

Compare DOS 3.3 (cycle count in margin)

```
B94F-   BD 8C C0    LDA   $C08C,X
B952-   10 FB       BPL   $B94F
B954-   C9 D5       CMP   #$D5     | 2
B956-   D0 F0       BNE   $B948    | 2 *
B958-   EA          NOP            | 2
B959-   BD 8C C0    LDA   $C08C,X
B95C-   10 FB       BPL   $B959
```

...and this disk's RWTS:

```
BE4D-   BD 8C C0    LDA   $C08C,X
BE50-   10 FB       BPL   $BE4D
BE52-   29 D5       AND   #$D5     | 2
BE54-   C9 D5       CMP   #$D5     | 2
BE56-   D0 EE       BNE   $BE46    | 2 *
```

Despite being more "flexible" (matching $D5 or $D7), this disk's RWTS uses the same number of bytes of code and runs in the same number of cycles. Nice.

Oh, and of course this RWTS also uses that zero page value at $31 that we initialized during boot:

```
BE58-   BD 8C C0    LDA   $C08C,X
BE5B-   10 FB       BPL   $BE58
BE5D-   C5 31       CMP   $31
BE5F-   D0 F1       BNE   $BE52
```

But since we initialized it to the normal value (#$AA) and we're still executing that code during boot, it's set to the proper value by the time this RWTS relies on it.

(*) on the time-critical path, this
    branch is not taken, so always 2

Further-furthermore, here is the code that checks the address epilogue:

```
BE8C-    BD 8C C0     LDA   $C08C,X
BE8F-    10 FB        BPL   $BE8C
BE91-    C9 DE        CMP   #$DE
BE93-    F0 09        BEQ   $BE9E
BE95-    08           PHP
BE96-    28           PLP
BE97-    BD 8C C0     LDA   $C08C,X
BE9A-    C9 08        CMP   #$08
BE9C-    B0 A2        BCS   $BE40
BE9E-    18           CLC
BE9F-    60           RTS
```

It appears that this RWTS has two distinct code paths for the address epilogue: either the first nibble is #$DE, or the first nibble is anything-except-#$DE-that-is-followed-by-a-timing-bit.

Here's why.

Each bit on disk takes 4 CPU cycles to come around as the disk is spinning. The data latch is the softswitch in the Apple II memory ($C0EC for slot 6, but usually written as "$C08C,X" to be slot-independent) that corresponds to the "current" value that's been read from the disk so far. Normally you just poll the data latch until the high bit goes on, at which point you have a full nibble. That's why most RWTS code has an LDA/BPL loop, including this one:

```
BE8C-    BD 8C C0     LDA   $C08C,X
BE8F-    10 FB        BPL   $BE8C
```

However, you aren't required to poll
the data latch constantly. It acts as a
micro-cache, keeping the "current"
value as bits go flying by, whether
you're polling it or not.

For example, the epilogue on this disk
is "AF FF FF", with a timing bit after
the "AF". So the bitstream looks like
this:

```
       1010111101111111111111111
       |--AF--| |--FF--||--FF--|
```

As the disk spins(*), these bits are
shifted into the data latch at a rate
of 1 bit every 4 CPU cycles.

(*) I still maintain that "As The Disk
    Spins" would make a great name for
    a retrocomputing-themed soap opera.

Thus:

```
Time | <-- as the disk spins |  $C0EC
-----+---------^---------------+--------
 -28 | .......10101110111111 | 00000001
-----+---------^---------------+--------
 -24 | ......101011101111111 | 00000010
-----+---------^---------------+--------
 -20 | .....1010111011111111 | 00000101
-----+---------^---------------+--------
 -16 | ....10101110111111111 | 00001010
-----+---------^---------------+--------
 -12 | ...101011101111111111 | 00010101
-----+---------^---------------+--------
  -8 | ..1010111011111111111 | 00101011
-----+---------^---------------+--------
  -4 | .10101110111111111111 | 01010111
-----+---------^---------------+--------
   0 | 101011101111111111111 | 10101111
-----+---------^---------------+--------
  +4 | 010111011111111111111 | 10101111
-----+---------^---------------+--------
  +8 | .......11111111111111 | 00000001
-----+---------^---------------+--------
 +12 | ......111111111111111 | 00000011
-----+---------^---------------+--------
 +16 | .....1111111111111111 | 00000111
-----+---------^---------------+--------
 +20 | ....1111111111111111. | 00001111
-----+---------^---------------+--------
```

At T+0, the high bit of the data latch goes to 1 for the first time, so the LDA/BPL loop will exit. After that, it's literally a race against time, because the disk keeps spinning (and the bits keep coming) independently of the CPU.

At T+4, the disk sees the extra 0 bit (a.k.a. timing bit) after the $AF nibble. This does not change the value of the data latch; it "holds" its value until it sees a new 1 bit. If the timing bit had not been present, the data latch would have seen a 1 bit here instead (the high bit of every nibble is always 1), which would have caused the data latch to reset and start accumulating the new value ($01). But because the timing bit is present, that reset gets delayed by 4 CPU cycles.

At T+8, the first bit of the $FF nibble shows up. This is a 1, so now the data latch resets and starts accumulating the new value ($01).

At T+12, the second bit of the $FF nibble shows up. This is also a 1. All the bits of $FF are 1. It gets shifted into the data latch, which is now $03.

At T+16, the third bit of $FF shows up. The data latch is now $07.

At T+20, the 4th bit of $FF shows up. The data latch is now $0F.

Timing bits are easy to write to disk, if you know where you want them to go. (You literally do nothing for 4 CPU cycles after writing a nibble to disk.) But bit copiers like EDD and Copy II Plus could not reliably preserve timing bits on copies they made. The presence of a timing bit is an indicator that the disk is an original, and the absence of a timing bit means the disk is an unauthorized bit copy.

Thus, the "race against time" looks like this:

	---$C0EC-data-latch--		
Time	original	copy	identical?
-28	00000001	00000001	yes
-24	00000010	00000010	yes
-20	00000101	00000101	yes
-16	00001010	00001010	yes
-12	00010101	00010101	yes
-8	00101011	00101011	yes
-4	01010111	01010111	yes
0	10101111	10101111	yes
+4	10101111	00000001	NO!
+8	00000001	00000011	NO!
+12	00000011	00000111	NO!
+16	00000111	00001111	NO!
+20	00001111	00011111	NO!

As you can see, there is a short window of time -- after you read a nibble from disk but before the next nibble has fully shifted into place -- where the value of the data latch will indicate whether the previous nibble had a timing bit after it. Combined with the fact that bit copiers do not reliably preserve timing bits in non-standard places, and you've got yourself a protection check BUILT INTO THE RWTS.

Thus, counting cycles again:

```
Time

        BD 8C C0    LDA   $C08C,X
  0     10 FB       BPL   $BE8C    | 2 *
 +2     C9 DE       CMP   #$DE     | 2
 +4     F0 09       BEQ   $BE9E    | 2 *
 +6     08          PHP            | 3
 +9     28          PLP            | 4
 +13    BD 8C C0    LDA   $C08C,X  | 4
        C9 08       CMP   #$08
        B0 A2       BCS   $BE40
        18          CLC
        60          RTS
```

The two extra "useless" instructions at $BE95 and $BE96 burn an additional 7 CPU cycles. By the time we poll the data latch again, we're in the tiny window of time where the value of the data latch will be different if the last nibble was followed by a timing bit. If there was a timing bit, the data latch will by $07. If there was no timing bit, the data latch will be $0F.

Boom.

Ironically, this code path is never taken on my partially cracked copy, because the demuffin process converted all the sectors to use the standard epilogue sequence. The first comparison (to #$DE) succeeds, so we branch over the second poll of the data latch and all this timing stuff is irrelevant. Its threat model was bit copiers, not crackers, and against that threat it was wildly successful.

I just wanted to explain how it worked.

Chapter 9
The Old Zero Page Switcheroo

Returning to my partially cracked copy,
it can now load DOS, but then it starts
grinding. Why? Not because of timing
bits! It's because the RWTS is now
expecting a non-standard data prologue
("D5 AA B5" instead of "D5 AA AD"). I
need to find where that change happens.

It's not built into the RWTS. The code
to check the data prologue doesn't do
any fancy bit math; it just compares
against zero page $4E (at $BDFA). So I
need to find where that memory location
is being changed.

Turning again to my trusty Disk Fixer
sector editor, I search the disk for
"85 4E" (STA $4E), but find nothing.
Hmm. Let's try "84 4E" (STY $4E). Also
nothing. Hmm. Maybe "86 4E" (STX $4E)?
Aha! I find a match on T01,S02, which
is loaded at $A300.

```
T01,S02
----------- DISASSEMBLY MODE ----------
0095:C9 1E              CMP   #$1E
0097:D0 0F              BNE   $00A8

; This appears to be the entry point
; for "standard mode" (tracks $00-$02).
; First, set the third data prologue
; to #$AD for reading and writing.
0099:A2 AD              LDX   #$AD
009B:86 4E              STX   $4E
009D:8E 5D BD           STX   $BD5D

; Second, set one of the indices in the
; nibble translation table to its
; standard value. (This is the table
; starts at $BA29 in DOS 3.3.)
00A0:A2 9B              LDX   #$9B
00A2:8E 2C BF           STX   $BF2C

; and skip ahead
00A5:4C BB A3           JMP   $A3BB
00A8:AE 6E AA           LDX   $AA6E
00AB:E0 9B              CPX   #$9B
00AD:B0 EA              BCS   $0099

; This appears to be the entry point
; for "protected mode" (tracks $03+).
; First, set the third data prologue
; to #$B5 for reading and writing.
00AF:A2 B5              LDX   #$B5
00B1:86 4E              STX   $4E
00B3:8E 5D BD           STX   $BD5D
```

```
; Second, set one of the indices in the
; nibble translation table to a non-
; standard value. (This explains why
; the disk catalog track looked garbled
; when I first examined it in a sector
; editor. All I did was change the
; prologues and epilogues, not the
; nibble-to-byte parameters!)
00B6:A2 D5              LDX   #$D5
00B8:8E 2C BF           STX   $BF2C
00BB:AA                 TAX
00BC:BD 1F 9D           LDA   $9D1F,X
00BF:48                 PHA
00C0:BD 1E 9D           LDA   $9D1E,X
00C3:48                 PHA
00C4:60                 RTS
```

--^--

I can leave the first half of this routine alone; I just need to change the second half that sets non-standard values. I can't simply disable the second half, because the program is written in BASIC, and BASIC constantly overwrites zero page $4E. The RWTS depends on this routine being called to set it back to the correct value. But I can change the #$B5 prologue value (at $A3B0) to #$AD and the #$D5 nibble translation index (at $A3B7) to #$9B.

```
T01,S02,$B0 change "B5" to "AD"
T01,S02,$B7 change "D5" to "9B"
```

Now the second half of the routine has the same effect as the first, setting everything to standard values. It looks like this:

```
                 --v--

----------- DISASSEMBLY MODE ----------
00AF:A2 AD          LDX   #$AD   <-- OK
00B1:86 4E          STX   $4E
00B3:8E 5D BD       STX   $BD5D
00B6:A2 9B          LDX   #$9B   <-- OK
00B8:8E 2C BF       STX   $BF2C
00BB:AA             TAX
00BC:BD 1F 9D       LDA   $9D1F,X
00BF:48             PHA
00C0:BD 1E 9D       LDA   $9D1E,X
00C3:48             PHA
00C4:60             RTS

                 --^--
```

Step 4 complete. Now I have a copy that boots and loads the HELLO program, then reboots.

Chapter 10
Ol' Faithful

This is where I was with my previous attempt that replaced the original DOS with a full copy of DOS 3.3. But now I know why my copy is rebooting: the HELLO program checks the value of $B7F6 to ensure that it's 0.

$B7F6 is the actual disk volume number found after the RWTS returns. Every address field contains a disk volume number. It's stored temporarily in zero page (along with the track, sector, and address field checksum), then it gets copied to the RWTS parameter table that starts at $B7E8. Normally this transfer occurs at $BE15, but in this DOS it's located at $BB15.

```
; get track number from zero page
BB15-   A5 2F          LDA   $2F

; calculate offset into RWTS parameter
; table
BB17-   A0 0E          LDY   #$0E

; store it
BB19-   91 48          STA   ($48),Y
```

(Although it's located in a different page, this is identical to the code in DOS 3.3. That's interesting! It means that the original disk really did have a disk volume 000 embedded in every sector on every track. The RWTS isn't doing anything special here -- it's faithfully reporting the disk volume number it found, just like always.)

There's not enough time (or space) to hack the address field parsing code to zero out the disk volume number, but I don't think the temporary zero page location is ever checked. I should be able to ignore the value of zp$2F and load the accumulator with 0 instead (at $BB15).

T00,S05,$15 change "A5 2F" to "A9 00"

Step 5 complete. Now I have a copy that
...
...
...works!

After extensive testing, there doesn't appear to be any further protection.

Quod erat liberandum.

Changelog

2016-10-07

- typos (thanks Andrew R)

2016-03-04

- initial release