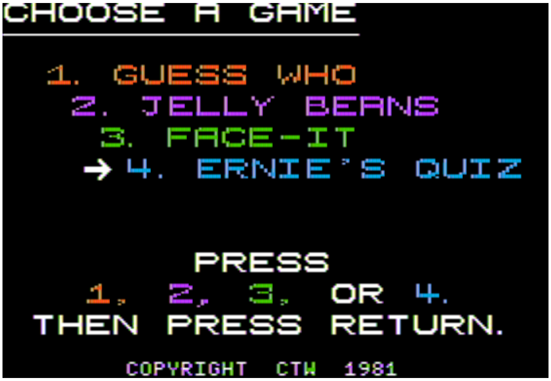
Ernie's Quiz

2016-12-31

4am
to deprotect
and preserve

Contents

```
---------------Ernie's Quiz---------------
A 4am crack                    2016-12-31
------------------------------------------





Name: Ernie's Quiz
Genre: educational
Year: 1981
Authors: Children's Television Workshop
Publisher: Apple ("Special Delivery"
  label)
Platform: Apple ][+ or later
Media: single-sided 5.25-inch floppy
OS: DOS 3.3P
Previous cracks: none
```



CHOOSE A GAME

1. GUESS WHO
2. JELLY BEANS
3. FACE-IT
→ 4. ERNIE'S QUIZ

PRESS
1, 2, 3, OR 4.
THEN PRESS RETURN.

COPYRIGHT CTW 1981

Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

COPYA
  immediate disk read error, but it
  gets a participation trophy just for
  showing up

Locksmith Fast Disk Backup
  unable to read anything except, oddly
  enough, track $11 and one sector on
  track $22:

```
                 --v--

     LOCKSMITH 7.0  FAST DISK BACKUP


   R*****************.*****************
   W
HEX 00000000000000001111111111111111222
TRK 0123456789ABCDEF0123456789ABCDEF012
   0AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDD.
   1AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   2AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   3AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   4AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   5AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   6AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   7AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   8AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   9AAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   AAAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   BAAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   CAAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   DAAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
12 EAAADDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
   FAADDDDDDDDDDDDDDD.DDDDDDDDDDDDDDDDD
[              ] PRESS [RESET] TO EXIT.

                 --^--
```

```
Copy ][+ nibble editor
  Track $00 has just a few standard
  sectors, but a bunch of other data.
  Track $01-$02 has data but not sure
  what format.
  Tracks $03+ have a standard address
  prologue ("D5 AA 96") but the data
  prologue varies per track. For
  example, track $03 uses "D5 AA E5":
```

```
                 --v--

   COPY ][ PLUS BIT COPY PROGRAM 8.4
(C) 1982-9 CENTRAL POINT SOFTWARE, INC.
---------------------------------------

TRACK: 03  START: 2DFD  LENGTH: 015F

2DD8: D2 F4 BD BD BD BD BD BD    VIEW
2DE0: BD BD BD BD BD BD BD BD
2DE8: BD BD BD BD BD BD BD BD
2DF0: BD BD BD BD BD BD BD BD
2DF8: BD BD BD BD BD D5 AA 96  <-2DFD
                     ^^^^^^^^
                 address prologue

2E00: FF FE AB AB AA AA FE FF
      ^^^^^ ^^^^^ ^^^^^ ^^^^^
      V=255 T=$03 S=$00 chksm

2E08: DE AA EB BD BD BD BD BD
      ^^^^^^^^
  address epilogue

2E10: BD BD D5 AA E5 9B CF 9B
            ^^^^^^^^
          data prologue

2E18: DA B7 B7 9B 9B 9B 9A 9B

---------------------------------------

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ

                 --^--
  Higher tracks use a different value
  for the third data prologue nibble.
  At track $09, it switches to #$DD.
  At track $0D, it switches to #$DA.
  And so on. No particular pattern.

Disk Fixer
  If I set "CHECKSUM ENABLED" = "NO",
  several sectors are readable on
  track $00 ($00, $07, $0D, $0E).
  Of particular interest is that byte
  $00 of sector $00 is #$02, not the
  usual #$01, meaning that the drive
  firmware will load two sectors from
  track $00 before passing control to
  $0801.

  Also, there's an interesting message
  about "your DOS 3.3.0P duplication
  master" on sector $0E:
```

```
                 --v--

-------------- DISK EDIT ---------------
TRACK $00/SECTOR $0E/VOLUME $FE/BYTE$00
---------------------------------------
$00:  >A<= . @ . J J J J I @ . G ) @ .
$10:   F X   . ~   / {   . ~   . ~   X
$20:   |   @ 9 4 H   p } H @ 6 P u   L
$30:   } , F @ Y O U R   3 . 3 . 0 P
$40:   D U P L I C A T I O N   M A S T
$50:   E R   C A N             N O T
$60:   B E   B O O T E D   A F T E R
$70:   I T   H A S   B E E N   U P D A
$80:   T E D   I T   S H O U L D   O N
$90:   L Y   B E   U S E D   F O R   D
$A0:   U P L I C A T I O N . . . .
$B0:       A P P L E   I I   E D I T O
$C0:   R   A S S E M B L E R   I I . .
$D0:           C R E A T E D   O N   1
$E0:   0 - N O V - 8 1   R E L   1 . 1
$F0:   . .                     B Y
---- -- --------------------------------
BUFFER 0/SLOT 6/DRIVE 1/MASK OFF/NORMAL

---------------------------------------
COMMAND : _


                 --^--
```

This disk sounds like it's loading DOS, then doing file-based disk reads later. The initial DOS load is sequential from track $00 to $02 (instead of the usual track $00, seek to track $02, then read backwards to track $00 again). After the initial DOS load, it displays a BASIC prompt and sounds like it's loading files.

Also, there's a readable disk catalog
on track $11 (shown here via Copy II
Plus):

```
                 --v--

CATALOG DISK            SLOT 6  DRIVE 1




  *I 044 MENU
  *I 113 JELLY BEANS
  *I 087 FACE-IT
  *I 080 GUESS WHO
  *I 120 ERNIE'S QUIZ
  *B 051 RAMLOADER

SECTORS FREE:1    USED:559    TOTAL:560

PRESS [RETURN]

                 --^--
```

If there's a DOS, there's an RWTS, and
if there's an RWTS, I should capture it
so I can use it to convert the disk to
a standard format.

Next steps:

1. Trace the boot
2. Capture the RWTS
3. Convert the disk to a standard format with Advanced Demuffin
4. Patch the RWTS (or replace the DOS entirely, depending on how things shake out)
5. 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.

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. Note that the
; drive firmware reads two sectors into
; $0800..$09FF, because the first
; sector had a #$02 at byte 0 instead
; of the usual #$01.
96F8-    A0 00          LDY   #$00
96FA-    B9 00 08       LDA   $0800,Y
96FD-    99 00 28       STA   $2800,Y
9700-    B9 00 09       LDA   $0900,Y
9703-    99 00 29       STA   $2900,Y
9706-    C8             INY
9707-    D0 F1          BNE   $96FA
9709-    AD E8 C0       LDA   $C0E8
970C-    4C 00 C5       JMP   $C500

; save this custom trace program to my
; work disk
*BSAVE TRACE,A$9600,L$10F

; and run it
*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOT0,A$2800,L$200

Hooray!
```

Chapter 2
In Which Every Exit Is An Entrance Somewhere Else

Now let's see what the bootloader looks
like.

```
]CALL -151

; move the code back to $0800 where it
; would be loaded by the drive firmware
*800<2800.29FFM

*801L

; The carry is always set coming out of
; the drive firmware, so this branch is
; never taken.
0801-   90 4A       BCC   $084D

; Zero page $27 is the high byte of the
; next address where the drive firmware
; would read the next sector, if there
; was one. Since it starts at #$08 and
; it already read two sectors (because
; $0800 is 2), it's now #$0A.
; Decrementing makes it #$09.
0803-   C6 27       DEC   $27      ; =09

; X is the boot slot x16, so #$60 for
; slot 6 (or #$50 for slot 5, &c.)
; $0900 was read by the drive firmware,
; and it apparently contains a sparse
; table of values spaced out to make
; this lookup work.
0805-   BD 31 09    LDA   $0931,X ;A=76
0808-   49 B0       EOR   #$B0    ;A=C6
080A-   48          PHA
080B-   C6 3D       DEC   $3D      ; =01
```

```
; Y is always 0 coming out of the drive
; firmware.
080D-   98              TYA             ;A=00
080E-   C8              INY             ;Y=01
080F-   48              PHA

; $0800 is never changed by the drive
; firmware, so it's still #$02.
; Decrementing makes it #$01.
0810-   CE 00 08        DEC   $0800     ; =01
0813-   A9 20           LDA   #$20      ;A=20
0815-   C6 27           DEC   $27       ; =08
0817-   48              PHA
```

OK, we've now pushed 3 values to the
stack: #$C6, #$00, #$20. The first was
dependent on the sparse table in page 2
which was indexed by the boot slot x16
(in X).

[God mode edit: booting from other
 slots does indeed set the first value
 to #$Cn where n is the boot slot.]

```
; ($26) now points to $0800 because we
; decremented zp$27 twice. Y is 1, so
; this EORs the accumulator with the
; value of $0801, which is #$90.
0818-   51 26           EOR   ($26),Y ;A=B0
```

```
; Set $0801 to #$B0, which is a "BCS"
; instruction. This means the next time
; we enter $0801 from the drive
; firmware, we'll take the branch
; instead of falling through. And that
; means this was all a fancy way of
; having a one-time setup routine,
; which we are still in the middle of.
081A-   91 26       STA   ($26),Y ; =B0
081C-   AA          TAX           ;X=B0
081D-   A5 27       LDA   $27     ;A=08
081F-   85 32       STA   $32     ; =08
0821-   CE 00 08    DEC   $0800   ; =00
0824-   A8          TAY           ;Y=08

; $33+$B0=$E3, which is uninitialized,
; so I don't know WTF is going on
0825-   B5 33       LDA   $33,X   ;A=??
0827-   84 29       STY   $29     ; =08
0829-   84 21       STY   $21     ; =08

; Now we're clobbering the accumulator
; anyway, so what was the LDA at $0825
; all about?
082B-   8A          TXA           ;A=B0
082C-   A2 17       LDX   #$17    ;X=17
082E-   86 31       STX   $31     ; =17

; again with the uninitialized $E3
0830-   D5 33       CMP   $33,X   ; ???

; zp$2B is the boot slot x16.
0832-   A6 2B       LDX   $2B     ;X=60

; Again, we're looking at the sparse
; lookup table in page 9. The value
; will vary by the boot slot.
0834-   5D 31 09    EOR   $0931,X ;A=C6
0837-   85 29       STA   $29     ; =C6
```

```
; Another sparse lookup table. The
; value at $0932+$60=$0992 is #$60, so
; this ends up as #$A6. (The values in
; this table and the first table are
; laid out such that this EOR value
; always ends up as #$A6, regardless
; of the boot slot.)
0839-   5D 32 09    EOR   $0932,X ;A=A6
083C-   C6 3D       DEC   $3D     ; =00
083E-   85 28       STA   $28     ; =A6

; And a third sparse lookup table, but
; all the values in this one are #$A3
; (not even kidding), so this always
; ends up as #$05.
0840-   5D 33 09    EOR   $0933,X ; =05
0843-   85 48       STA   $48     ; =05
0845-   A0 2B       LDY   #$2B    ;Y=2B
0847-   84 20       STY   $20     ; =2B

; zp$40 is always 0 coming out of the
; drive firmware. Decrementing always
; makes it #$FF.
0849-   C6 40       DEC   $40     ; =FF

; This branch is always taken.
084B-   30 3E       BMI   $088B

*88BL

088B-   A4 48       LDY   $48     ;Y=05
088D-   EA          NOP
088E-   BD 8C C0    LDA   $C08C,X
0891-   10 FB       BPL   $088E
```

```
; ($20) points to $082B (set at $0847
; and $0829). Y=5, so this EORs with
; the value at $0830, which is #$D5.
; Also, that's part of the code we just
; executed, and it was one of the
; instructions that made no sense at
; the time. Perhaps its only purpose in
; life was to frustrate and confuse me.
0893-   51 20        EOR   ($20),Y ; =D5
0895-   D0 F4        BNE   $088B
0897-   2C 40 40     BIT   $4040
089A-   BD 8C C0     LDA   $C08C,X
089D-   10 FB        BPL   $089A

; ($31) points to $0817 (set at $082E
; and $081F). Y is still 5, so this
; compares with the value at $081C,
; which is #$AA. Which is also part of
; the code we just executed.
089F-   D1 31        CMP   ($31),Y ; =AA
08A1-   D0 F0        BNE   $0893
08A3-   2C 40 40     BIT   $4040
08A6-   BD 8C C0     LDA   $C08C,X
08A9-   10 FB        BPL   $08A6

; This address is a constant across all
; Apple II models.
08AB-   CD D0 FF     CMP   $FFD0   ; =EF
08AE-   D0 E3        BNE   $0893
```

OK, so we're looking for a prologue of "D5 AA EF".

```
08B0-   38           SEC
08B1-   2C 40 40     BIT   $4040
```

```
; Now get a 4-and-4-encoded value...
08B4-   BD 8C C0    LDA   $C08C,X
08B7-   10 FB       BPL   $08B4
08B9-   2A          ROL
08BA-   8D C3 08    STA   $08C3
08BD-   BD 8C C0    LDA   $C08C,X
08C0-   10 FB       BPL   $08BD
08C2-   29 FF       AND   #$FF

; ...and use it as the target address
; for another sector read.
08C4-   85 27       STA   $27

; ($28) points to $CnA6 (where n is the
; boot slot, set at $082E and $0837).
08C6-   6C 28 00    JMP   ($0028)
```

$C6A6 is an entry point I've never seen before. Usually when disks want to re-use the drive firmware, they call $C65C to read an entire sector. What's $C6A6?

```
*C6A6L

C6A6-   A0 56       LDY   #$56
C6A8-   84 3C       STY   $3C
C6AA-   BC 8C C0    LDY   $C08C,X
C6AD-   10 FB       BPL   $C6AA
C6AF-   59 D6 02    EOR   $02D6,Y
C6B2-   A4 3C       LDY   $3C
C6B4-   88          DEY
C6B5-   99 00 03    STA   $0300,Y
C6B8-   D0 EE       BNE   $C6A8
C6BA-   84 3C       STY   $3C
C6BC-   BC 8C C0    LDY   $C08C,X
C6BF-   10 FB       BPL   $C6BC
C6C1-   59 D6 02    EOR   $02D6,Y
C6C4-   A4 3C       LDY   $3C
C6C6-   91 26       STA   ($26),Y
C6C8-   C8          INY
C6C9-   D0 EF       BNE   $C6BA
C6CB-   BC 8C C0    LDY   $C08C,X
C6CE-   10 FB       BPL   $C6CB
C6D0-   59 D6 02    EOR   $02D6,Y
...
```

This is just the data field decoding routine, taking nibbles on disk and turning them into bytes. That means the structure of the data we're reading on track 0 is

"D5 AA EF" aa bb <data field>

"aa bb" is a 4-and-4-encoded value that has the high byte of the address in memory where we're going to store the data in the data field that follows.

No address field. No address epilogue. No data prologue. Just a non-standard prologue, a non-standard addressing scheme, and a raw data field.

I don't even know what to capture, because the address where the decoded data ends up is part of the encoded data.

To make matters worse, the bootloader is self-modifying. The call to $C6A6 exits via $0801, but that instruction was modified (at $081A) to be a "BCS" instead of a "BCC". The carry bit is always set coming out of the drive firmware, due to a compare at the very end, so that "BCS" instruction becomes an unconditional branch. To where?

```
*801:B0

*801L

0801-   B0 4A       BCS   $084D

*84DL

084D-   24 40       BIT   $40
084F-   30 33       BMI   $0884
```

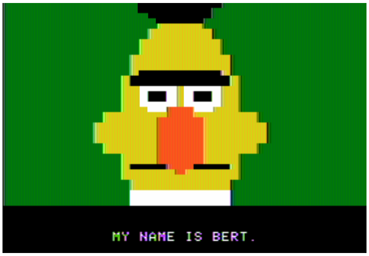
Zero page $40 was decremented to #$FF (at $0849), and the data field decoding routine at $C6A6 doesn't touch it, so this is another unconditional branch.

```
*884L

0884-    C5 48        CMP   $48
0886-    D0 03        BNE   $088B
0888-    4C 00 09     JMP   $0900
```

Aha! Coming out of the drive firmware, the accumulator is the next physical sector number. Zero page $48 was set to #$05 (at $0843), and now we have an exit condition. The next phase of the boot apparently starts at $0900.

```
96F8-    A9 4C        LDA   #$4C
96FA-    8D 88 08     STA   $0888
96FD-    A9 59        LDA   #$59
96FF-    8D 89 08     STA   $0889
9702-    A9 FF        LDA   #$FF
9704-    8D 8A 08     STA   $088A
9707-    4C 01 08     JMP   $0801

*BSAVE TRACE2,A$9600,L$10A
*9600G
...reboots slot 6...
...loads game...
...never breaks...
```

I've missed something.


MY NAME IS BERT.

Chapter 3
In Which We Celebrate Progress, Not Perfection

Now that I know the structure (minimal as it may be) of the data we're reading from track 0, I can go back to the Copy II Plus nibble editor and make sense of the raw data.

Searching for the custom prologue "D5 AA EF", I find five "sectors" and their addresses:

--v--

```
   COPY ][ PLUS BIT COPY PROGRAM 8.4
(C) 1982-9 CENTRAL POINT SOFTWARE, INC.
----------------------------------------

TRACK: 00  START: 1800  LENGTH: 3DFF

2420: FA FD FD FD FD FD FD FD
2428: FD FD FD FD FD FD FD FD
2430: FD FD FD FD FD FD FD FD
2438: FD FD F7 F7 F7 F7 F7 F7
2440: D5 AA EF AA BA CF AB A7
      ^^^^^^^^ ^^^^^
      prologue  $10

2448: A6 9F A6 E7 EE DA DF DF
2450: AC E9 F5 CF CE A6 96 FD
2458: DF B9 DC FA CB 9E 9F A6
2460: F3 96 97 AF AF 9B F2 D7

.
.
.
```

```
2730: EF EF EF EF EF EF EF EF
2738: EF EF EF EF EF EF EF EF
2740: EF EF EF EF EF EF EF EF
2748: EF EF FB FB FB FB FB FB
2750: D5 AA EF AA BB D3 B9 9D
      ^^^^^^^^ ^^^^^
      prologue  $11

2758: CD 9D 9E EE B9 E9 BD 96
2760: 96 CD DF AE B4 B4 D9 ED
2768: ED AF B2 BF B7 9A B2 F3
2770: B9 DB B2 B2 D6 D7 B5 AE

 .
 .
 .

2A40: E9 BA DD DD DD DD DD DD
2A48: DD DD DD DD DD DD DD DD
2A50: DD DD DD DD DD DD DD DD
2A58: DD DD EB EB EB EB EB EB
2A60: D5 AA EF AB BA B6 9D 9B
      ^^^^^^^^ ^^^^^
      prologue  $12

2A68: 97 96 D6 D6 96 96 96 96
2A70: 96 96 96 96 96 9B 96 97
2A78: 9A 9A D6 EF D9 ED D6 EE
2A80: D7 F2 D6 F2 B6 9B 9B 97

 .
 .
 .
```

```
2D48: DC B7 B7 B7 B7 B7 B7 B7
2D50: B7 B7 B7 B7 B7 B7 B7 B7
2D58: B7 B7 B7 B7 B7 B7 B7 B7
2D60: B7 B7 B7 B7 D7 D7 D7 D7
2D68: D7 D7 D5 AA EF AB BB DE
            ^^^^^^^^ ^^^^^
            prologue  $13

2D70: B4 AE CE ED F9 B4 AD EE
2D78: DD B4 F9 FE D3 CD D3 B3
2D80: 97 CE D9 B3 A7 EE BA FF
2D88: CB FD EC F7 EB FD AB AB

.
.
.

3050: FF FF BD EF CB F2 FC BF
3058: BF BF BF BF BF BF BF BF
3060: BF BF BF BF BF BF BF BF
3068: BF BF BF BF BF BF BF BF
3070: AF AF AF AF AF AF D5 AA
                        ^^^^^
                        prologue

3078: EF AA AB DF 96 96 96 96
      ^^ ^^^^^
          $01

3080: 96 96 96 96 96 96 96 96
3088: 96 96 96 96 96 96 96 96
3090: 96 96 96 96 96 96 96 96
```

--^--

If my interpretation is correct, we're reading four sectors worth of data into $1000-$13FF, then one final sector directly onto the stack ($0100-$01FF).

Let's test that theory.

```
*C500G
...
]CALL -151

; clear memory
*800:FD N 801<800.BEFEM

; copy drive firmware
*9600<C600.C6FFM

; set up a callback at $08C6 instead of
; jumping to $CnA6
96F8-   A9 4C        LDA   #$4C
96FA-   8D C6 08     STA   $08C6
96FD-   A9 0A        LDA   #$0A
96FF-   8D C7 08     STA   $08C7
9702-   A9 97        LDA   #$97
9704-   8D C8 08     STA   $08C8

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

; (callback is here)
; store the high byte of the target
; address (currently in accumulator)
; into consecutive memory locations
970A-   8D FB 97     STA   $97FB
970D-   EE 0B 97     INC   $970B
```

```
; branch after storing 5th byte (##$FB +
; ##$05 = ##$100, so the INC instruction
; will set $970B to ##$00 and flip the Z
; bit)
9710-   F0 03          BEQ   $9715

; continue the boot
9712-   4C A6 C6       JMP   $C6A6

; execution continues here (from $9710)
; after storing the 5th address byte --
; break to the monitor so I can see
; what the heck is going on
9715-   4C 59 FF       JMP   $FF59

*BSAVE TRACE3,A$9600,L$118
*9600G
...reboots slot 6...
<beep>

*97FB.97FF

97FB- 10 11 12 13 01
```

Indeed.

```
*1000L
.
.
.
1024-   85 26       STA   $26
1026-   0A          ASL
1027-   85 3C       STA   $3C
1029-   86 3E       STX   $3E
102B-   A9 00       LDA   #$00
102D-   85 2F       STA   $2F
102F-   A9 00       LDA   #$00
1031-   85 3D       STA   $3D
1033-   4C 41 10    JMP   $1041
1036-   E6 3D       INC   $3D
1038-   A5 3D       LDA   $3D
103A-   C5 3C       CMP   $3C
103C-   90 03       BCC   $1041
103E-   4C F0 10    JMP   $10F0
1041-   A6 2B       LDX   $2B
1043-   20 14 11    JSR   $1114
1046-   B0 EE       BCS   $1036
```

Well I'm not sure what it is, but it wasn't there before.

```
*C500G
...

]BSAVE OBJ.1000-13FF,A$1000,L$400
```

This feels like progress -- and it is! I am down to just one unanswered question: how does this disk ever boot? I thought it continued to $0900, but execution never reaches the "JMP" at $0888. Overwriting the stack page is a neat trick, but it doesn't change the fact that the bootloader is an infinite loop. It never returns, and if you never return, it doesn't really matter what's on your stack.

Chapter 4
"He Cheated."
"I Changed The Conditions Of The Test."

As frustrating as it can be, I do love reverse engineering. If I conclude that a disk can not boot because it is stuck in an infinite loop, but in reality the disk boots, my conclusion is what needs adjusting -- not reality. This keeps me humble.

In this case, attempts to deny reality has led me to the conclusion that this bootloader has no exit condition. Obviously it does; I just haven't found it.

What have I found? The bootloader reuses the drive firmware in a strange way, reading raw data fields preceded by a custom prologue. The address of each "sector" is encoded on the disk, wedged between each prologue and data field. I have successfully stopped the bootloader after it reads 4 sectors into $1000-$13FF, and I know the final sector is going to overwrite the stack page at $0100. From manual inspection in a nibble editor, I know there are no more prologues on track 0, thus no more "sectors" that could possibly be read.

Also, not shown above, but I just now verified -- the code at $0800 and $0900 (initially read by the drive firmware before execution was ever passed to the bootloader) does not get overwritten, with the exception of the one byte at $0801 that changes the "BCC" to a "BCS" instruction and redirects execution flow to $084D. But that happened ages ago (on the first loop), and it hasn't been changed since then.

So.

The routine at $088B -- which matches
the custom "D5 AA EF" prologue and an
address byte -- is not getting called
again. To verify this, I changed my
last trace program (TRACE2, above) to
try to capture 6 address bytes instead
of 5. No dice, the disk boots. There
are only 5 sectors, only 5 addresses.
The drive firmware is told to read a
sector into $0100, then it exits to
$0801, then the disk boots. We're never
getting to $088B again.

What else is there between $0801 and
$088B? Very little.

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

*801:B0

*801L

0801-   B0 4A       BCS   $084D

*84DL

084D-   24 40       BIT   $40
084F-   30 33       BMI   $0884

*884L

0884-   C5 48       CMP   $48
0886-   D0 03       BNE   $088B
0888-   4C 00 09    JMP   $0900
```

I've already verified that execution
never hits $0888. That was my first
trace attempt (TRACE, above). The carry
bit is most definitely always set on
the way out of the drive firmware. Here
is the code to prove it:

```
*C6EDL

C6ED-   E6 3D       INC   $3D
C6EF-   A5 3D       LDA   $3D
C6F1-   CD 00 08    CMP   $0800     <--
C6F4-   A6 2B       LDX   $2B
C6F6-   90 DB       BCC   $C6D3     <--
C6F8-   4C 01 08    JMP   $0801
```

There is absolutely no way out of the
drive firmware without setting the
carry bit.

That leaves this code at $084D:

```
``'-.,_,.-'``'-.,_,.='``'-.,_,.-'``'-.,
``'-.,_,.-'``'-.,_,.='``'-.,_,.-'``'-.,
``
                                     .,
..  084D-   24 40       BIT   $40    .,
..  084F-   30 33       BMI   $0884  .,
``
                                     .,
``'-.,_,.-'``'-.,_,.='``'-.,_,.-'``'-.,
``'-.,_,.-'``'-.,_,.='``'-.,_,.-'``'-.,
```

As I mentioned before, zp$40 was
decremented to #$FF (at $0849), and the
data field decoding routine at $C6A6
doesn't touch it, so this must be an
unconditional branch. Unless...

Why was zp$40 zero to begin with? Well,
it's set earlier in the drive firmware:

```
*C65CL

C65C-   18              CLC
C65D-   08              PHP
C65E-   BD 8C C0        LDA   $C08C,X
C661-   10 FB           BPL   $C65E
C663-   49 D5           EOR   #$D5
C665-   D0 F7           BNE   $C65E
C667-   BD 8C C0        LDA   $C08C,X
C66A-   10 FB           BPL   $C667
C66C-   C9 AA           CMP   #$AA
C66E-   D0 F3           BNE   $C663
C670-   EA              NOP
C671-   BD 8C C0        LDA   $C08C,X
C674-   10 FB           BPL   $C671
C676-   C9 96           CMP   #$96
C678-   F0 09           BEQ   $C683 ---+
C67A-   28              PLP            |
C67B-   90 DF           BCC   $C65C    |
C67D-   49 AD           EOR   #$AD     |
C67F-   F0 25           BEQ   $C6A6    |
C681-   D0 D9           BNE   $C65C    |
C683-   A0 03           LDY   #$03  <--+
C685-   85 40           STA   $40
```

This is why zp$40 is always zero the
first time out of the drive firmware.
It's set to zero at $Cn85, immediately
after matching the standard "D5 AA 96"
prologue of sector 0.

But this doesn't help us, because we
didn't call $Cn5C to read a normally
structured sector; we called $CnA6 to
read just a data field.

Unless...

There is one condition that would cause the drive firmware to branch back to $Cn5C: if the data field checksum does not validate. At $CnCB, after all the nibbles of the data field have been read, it reads one final nibble -- the checksum, which is the exclusive OR of all the data field nibbles before it.

```
*C6CBL

C6CB-   BC 8C C0    LDY   $C08C,X
C6CE-   10 FB       BPL   $C6CB
C6D0-   59 D6 02    EOR   $02D6,Y
C6D3-   D0 87       BNE   $C65C   <-- !
```

So maybe -- and bear with me here, because this is insane -- but maybe the non-standard "sector" we're reading into $0100 is intentionally corrupt. When the checksum doesn't match, the drive firmware branches back to $Cn5C, and we end up reading an entirely normal sector into $0100 instead. That would reset zp$40 back to 0, and it would still be 0 when the firmware exits via $0801. The branch at $0801 is still "BCS" (modified at $081A), so we end up at $084D with zp$40 still 0. The "BIT" instruction leaves the N bit 0, so we fall through the "BMI" branch (which I previously thought was unconditional, but we've changed the conditions of the test), and we end up at $0851. What's at $0851?

```
*851L

0851-   24 24       BIT   $24
0853-   24 24       BIT   $24
0855-   40          RTI           <-- !
```

Oh my God. We would "return" to an
address on the stack, which we just
overwrote with a sector from disk.

("RTI" is the relatively unknown cousin
of "RTS". It pops 3 bytes off the stack
instead of 2, then uses one of them for
the status flags -- C, N, Z, &c. -- and
the other two as the return address.)

Well, here goes nothing...

```
*9600<C600.C6FFM

; set up callback at $0855 instead of
; executing the "RTI"
96F8-   A9 4C       LDA   #$4C
96FA-   8D 55 08    STA   $0855
96FD-   A9 0A       LDA   #$0A
96FF-   8D 56 08    STA   $0856
9702-   A9 97       LDA   #$97
9704-   8D 57 08    STA   $0857

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

; (callback is here)
; copy stack page to higher ground, er,
; memory, so it survives a reboot
970A-   A0 00       LDY   #$00
970C-   B9 00 01    LDA   $0100,Y
970F-   99 00 21    STA   $2100,Y
9712-   C8          INY
9713-   D0 F7       BNE   $970C
```

```
; turn off slot 6 drive motor and
; reboot to my work disk
9715-    AD E8 C0      LDA   $C0E8
9718-    4C 00 C5      JMP   $C500

*BSAVE TRACE4,A$9600,L$11B

; clear memory
*800:FD N 801<800.BEFEM

; run the trace
*BRUN TRACE4
...reboots slot 6...
...reboots slot 5...
```

Unbelievable.

```
]BSAVE OBJ.0100-01FF,A$2100,L$100
```

Now I have a new problem. Or rather, I have 256 of them.

Chapter 5
Stackmaker, Stackmaker, Make Me A Stack

Here's the problem: the stack is stored in main memory at $0100-$01FF, but the "top" of the stack is stored separately (in a register called the "stack pointer"). The stack pointer is just a byte, with values ranging from #$00 to #$FF, and it's used as an index into the stack page in memory. Any address in page 1 can be the "top" of the stack at any time, depending on the stack pointer, and in fact it wraps around if the pointer decrements past 0.

The stack pointer is undefined at boot. It can literally be anything from #$00 to #$FF. The drive firmware does not modify it. Most operating systems like Apple DOS 3.3 and ProDOS will reset it to #$FF during early boot.

But this bootloader never sets it. So when it executes that "RTI" at $0855 and pops off three bytes (one of the status flags and two for the return address)... where does it go? That depends on the stack pointer, which could be one of 256 different values.

It turns out it doesn't matter.

```
]CALL -151

*2100.21FF

2100- 10 12 11 10 12 11 10 12
2108- 11 10 12 11 10 12 11 10
2110- 12 11 10 12 11 10 12 11
2118- 10 12 11 10 12 11 10 12
2120- 11 10 12 11 10 12 11 10
2128- 12 11 10 12 11 10 12 11
2130- 10 12 11 10 12 11 10 12
2138- 11 10 12 11 10 12 11 10
2140- 12 11 10 12 11 10 12 11
2148- 10 12 11 10 12 11 10 12
2150- 11 10 12 11 10 12 11 10
2158- 12 11 10 12 11 10 12 11
2160- 10 12 11 10 12 11 10 12
2168- 11 10 12 11 10 12 11 10
2170- 12 11 10 12 11 10 12 11
2178- 10 12 11 10 12 11 10 12
2180- 11 10 12 11 10 12 11 10
2188- 12 11 10 12 11 10 12 11
2190- 10 12 11 10 12 11 10 12
2198- 11 10 12 11 10 12 11 10
21A0- 12 11 10 12 11 10 12 11
21A8- 10 12 11 10 12 11 10 12
21B0- 11 10 12 11 10 12 11 10
21B8- 12 11 10 12 11 10 12 11
21C0- 10 12 11 10 12 11 10 12
21C8- 11 10 12 11 10 12 11 10
21D0- 12 11 10 12 11 10 12 11
21D8- 10 12 11 10 12 11 10 12
21E0- 11 10 12 11 10 12 11 10
21E8- 12 11 10 12 11 10 12 11
21F0- 10 12 11 10 12 11 10 12
21F8- 11 10 12 11 10 12 11 10
```

Suppose the stack pointer is #$F8. The "RTI" will pop the next three values from the stack: #$10, #$12, and #$11 (shown here at $01F9 / $01FA / $01FB):

```
01F8- 11 10 12 11 10 12 11 10
      ^^ ^^^^^^^^^
 SP=$F8   RTI values
```

The #$10 will be used to set the status flags, and the #$12 and #$11 combine to form the return address, $1112. What's at $1112?

```
*BLOAD OBJ.1000-13FF
*1112L

1112-   60          RTS
```

Oh! Now what happens? Well, "RTS" pops another "return" address off the stack, adds 1, and continues execution there. So what's next on the stack?

```
01F8- 11 10 12 11 10 12 11 10
                  ^^^^^
               RTS values
```

$1210 + 1 = $1211. What's at $1211?

```
*1211L

1211-   EA          NOP
1212-   D8          CLD
1213-   A9 10       LDA   #$10
1215-   48          PHA
1216-   A9 78       LDA   #$78
1218-   48          PHA
1219-   D0 E6       BNE   $1201

*1201L

1201-   60          RTS
```

More stack manipulation! We pushed the bytes #$10 and #$78 to the stack, then the "RTS" at $1201 will "return" to $1078 + 1 = $1079.

```
*1079L

1079-   86 2B       STX   $2B
107B-   A9 00       LDA   #$00
107D-   85 2A       STA   $2A
107F-   85 2D       STA   $2D
1081-   AC 00 10    LDY   $1000
1084-   A9 00       LDA   #$00
1086-   99 D8 02    STA   $02D8,Y
1089-   88          DEY
108A-   D0 FA       BNE   $1086
...
```

OK, that's real code. We'll come back to that in a minute.

But what if the stack pointer is some
other value? The stack pointer (SP) can
only point to one of three values:
#$10, #$11, or #$12. Looking at the
pattern on the stack:

```
...
01E0- 11 10 12 11 10 12 11 10
01E8- 12 11 10 12 11 10 12 11
01F0- 10 12 11 10 12 11 10 12
01F8- 11 10 12 11 10 12 11 10
```

If SP points to a #$10, the "RTI" at
$0855 "returns" to $1011.

```
*1011L

1011-   60          RTS
```

Same trick! This "RTS" at $1011 will
pop another two bytes off the stack,
#$12 and #11, and continue to
$1112 + 1 = $1113.

```
*1113L

1113-   60          RTS
```

Same trick! This "RTS" at $1113 will
pop *another* two bytes off the stack,
#$10 and #$12, and continue to
$1210 + 1 = $1211, which we've already
seen.

Finally, if SP points to a #$12, the
"RTI" at $0855 will "return" to $1210.

```
*1210L

1210-   EA          NOP
```

$1210 is a "NOP", so it falls through
to $1211.

But wait, there's more! The stack is
just one page in memory, and SP can
"wrap" from one end to the other. What
if SP wraps around from #$FF to #$00
during this mess?

If SP = #$FD at boot, then the "RTI" at
$0855 will pop #$11 and #$10 (from
$01FE and $01FF) and #$10 (from
$0100). So execution continues at
$1010, which we had not previously
considered.

```
*1010L

1010-   60          RTS
```

Same trick!

Ah, but what if SP = #$F9 at boot?

```
01F8- 11 10 12 11 10 12 11 10
         ^^ ^^^^^^^^^
    SP=$F9    RTI values
```

The "RTI" at $0855 will "return" to
$1011, which we already know is just an
"RTS".

```
01F8- 11 10 12 11 10 12 11 10
                     ^^^^^
                  RTS values
```

$1112 + 1 = $1113, another "RTS". Now what? Now SP will wrap while popping the two bytes for this "RTS".

```
01F8- 11 10 12 11 10 12 11 10
                           ^^
                      RTS value #1
.
.
0100- 10 12 11 10 12 11 10 12
      ^^
 RTS value #2
```

$1010 + 1 = $1011, another "RTS". Now SP keeps incrementing, we keep popping, and the world keeps turning:

```
0100- 10 12 11 10 12 11 10 12
         ^^^^^
      RTS values
```

$1112 + 1 = $1113, another "RTS". One more time:

```
0100- 10 12 11 10 12 11 10 12
               ^^^^^
            RTS values
```

$1210 + 1 = $1211, and here we are.

EVERY POSSIBLE STACK POINTER VALUE ENDS UP AT $1211.

And that's how this disk boots itself: circuitously but inevitably, through self-modifying code, malformed sectors, intentionally invalid checksums, and an uninitialized stack pointer.

At least things can't get any worse.

Chapter 6
In Which Things, Somehow, Manage To Get Even Worse

Continuing, then, from $1079:

```
*1079L

; initialization of things
1079-   86 2B        STX   $2B
107B-   A9 00        LDA   #$00
107D-   85 2A        STA   $2A
107F-   85 2D        STA   $2D
1081-   AC 00 10     LDY   $1000

; clearing of things
1084-   A9 00        LDA   #$00
1086-   99 D8 02     STA   $02D8,Y
1089-   88           DEY
108A-   D0 FA        BNE   $1086

; hmm
108C-   AD 1E 10     LDA   $101E
108F-   AE 21 10     LDX   $1021
1092-   20 24 10     JSR   $1024
1095-   B0 59        BCS   $10F0

1097-   20 13 10     JSR   $1013

; hmm again
109A-   AE 22 10     LDX   $1022
109D-   AD 1F 10     LDA   $101F
10A0-   20 24 10     JSR   $1024
10A3-   B0 4B        BCS   $10F0

10A5-   20 13 10     JSR   $1013

; hmm x3
10A8-   AE 23 10     LDX   $1023
10AB-   AD 20 10     LDA   $1020
10AE-   20 24 10     JSR   $1024
10B1-   B0 3D        BCS   $10F0
```

If I know anything about anything, this is calling some sort of disk reading code. I would guess that $1024 reads some or all of a track, then $1013 advances to the next track. $10F0 is The Badlands in case there's a disk read error.

```
*10F0L

; boot slot x16
10F0-   A5 2B       LDA   $2B
10F2-   4A          LSR
10F3-   4A          LSR
10F4-   A0 00       LDY   #$00
10F6-   4A          LSR
10F7-   4A          LSR
10F8-   09 C0       ORA   #$C0

; e.g. $C6 for slot 6
10FA-   48          PHA
10FB-   98          TYA

; 0
10FC-   48          PHA

; also 0
10FD-   48          PHA

; reboots from whence we came
10FE-   40          RTI
```

Yeah, let's not end up there if we can help it.

What's at $1024?

```
*1024L

1024-   85 26       STA   $26
1026-   0A          ASL
1027-   85 3C       STA   $3C
1029-   86 3E       STX   $3E
102B-   A9 00       LDA   #$00
102D-   85 2F       STA   $2F
102F-   A9 00       LDA   #$00
1031-   85 3D       STA   $3D
1033-   4C 41 10    JMP   $1041 ------+
1036-   E6 3D       INC   $3D   <--+  |
1038-   A5 3D       LDA   $3D      |  |
103A-   C5 3C       CMP   $3C      |  |
103C-   90 03       BCC   $1041    |  |
103E-   4C F0 10    JMP   $10F0    |  |
1041-   A6 2B       LDX   $2B  <---|--+
1043-   20 14 11    JSR   $1114    |
1046-   B0 EE       BCS   $1036 ---+
```

Ah! zp$3C is initialized as twice the accumulator on entry, then zp$3D is used as a retry counter for how many times the routine at $1114 returns with the carry set. If that happens too many times, we end up at $10F0 (from $103E), which we already know is The Badlands.

```
; set a marker based on the accumulator
; minus $19 (WTF) in the array we
; cleared earlier
1048-   98              TYA
1049-   E9 19           SBC   #$19
104B-   A8              TAY
104C-   99 D7 02        STA   $02D7,Y

; maybe a sector count for this track?
104F-   C6 26           DEC   $26
1051-   D0 EE           BNE   $1041

; check the retry counter
1053-   A5 3D           LDA   $3D

; no retries, branch forward
1055-   F0 0F           BEQ   $1066

; at least one retry -- increment a
; Death Counter
1057-   E6 2F           INC   $2F
1059-   A5 2F           LDA   $2F
105B-   C9 08           CMP   #$08

; after 8 attempts, give up entirely
105D-   B0 18           BCS   $1077

; otherwise try to read everything
; again with zero retries
105F-   A5 3C           LDA   $3C
1061-   85 26           STA   $26
1063-   4C 2F 10        JMP   $102F
```

```
; Check that the array we cleared
; earlier (at $1086) was filled (at
; $104C), in the indices we expected to
; be filled. Start with index zp$3E,
; which was passed in in the X register
; from the caller, and count up N items
; where N was passed in in the
; accumulator and stored in zp$3C (but
; originally multiplied by 2 for
; unrelated reasons). Yeah, that's all
; kinds of odd, but that's what we're
; doing -- verifying that we filled a
; slice of an array.
1066-    A6 3E          LDX   $3E
1068-    A5 3C          LDA   $3C
106A-    4A             LSR
106B-    A8             TAY
106C-    BD D8 02       LDA   $02D8,X

; blank array item means we missed one
106F-    F0 06          BEQ   $1077
1071-    E8             INX
1072-    88             DEY
1073-    D0 F7          BNE   $106C
1075-    18             CLC
1076-    60             RTS
1077-    38             SEC
1078-    60             RTS
```

I'm almost positive that this loop is keeping track of which sectors are being read from disk, then checking to ensure it all got read properly.

Regardless, this is just administrative stuff. The real meat is at $1114. What's at $1114?

Chapter 7

"I Drove 400 Miles
 To Ride On A Camel."

"Why?"

"Because That's Where
 The Camels Are."

Before I can explain the next chunk of code, I will pause and explain a little bit of theory. As you probably know if you're the sort of person who reads this sort of thing, Apple II floppy disks do not contain the actual data that ends up being loaded into memory. Due to hardware limitations of the original Disk II drive, data on disk must be stored in an intermediate format called "nibbles." Bytes in memory are encoded into nibbles before writing to disk, and nibbles that you read from the disk must be decoded back into bytes. The round trip is lossless but requires some bit wrangling.

Decoding nibbles-on-disk into bytes-in-memory is a multi-step process. In "6-and-2 encoding" (used by DOS 3.3, ProDOS, and all ".dsk" image files), there are 64 possible values that you may find in the data field (in the range $96..$FF, but not all of those, because some of them have bit patterns that trip up the drive firmware). We'll call these "raw nibbles."

Step 1: read $156 raw nibbles from the data field. These values will range from $96 to $FF, but as mentioned earlier, not all values in that range will appear on disk.

Now we have $156 raw nibbles.

Step 2: decode each of the raw nibbles into a 6-bit byte between 0 and 63 (%00000000 and %00111111 in binary). $96 is the lowest valid raw nibble, so it gets decoded to 0. $97 is the next valid raw nibble, so it's decoded to 1. $98 and $99 are invalid, so we skip them, and $9A gets decoded to 2. And so on, up to $FF (the highest valid raw nibble), which gets decoded to 63.

Now we have $156 6-bit bytes.

Step 3: split up each of the first $56 6-bit bytes into pairs of bits. In other words, each 6-bit byte becomes three 2-bit bytes. These 2-bit bytes are merged with the next $100 6-bit bytes to create $100 8-bit bytes. Hence the name, "6-and-2" encoding.

The exact process of how the bits are split and merged is... complicated. The first $56 6-bit bytes get split up into 2-bit bytes, but those two bits get swapped (so %01 becomes %10 and vice-versa). The other $100 6-bit bytes each get multiplied by 4 (a.k.a. bit-shifted two places left). This leaves a hole in the lower two bits, which is filled by one of the 2-bit bytes from the first group.

A diagram might help. "a" through "x"
each represent one bit.

```
             -------------

1 decoded      3 decoded
nibble in  +   nibbles in   =  3 bytes
first $56      other $100


00abcdef       00ghijkl
               00mnopqr
   |           00stuvwx
   |
 split            |
   &           shifted
swapped        left x2
   |              |
   V              V

000000fe   +   ghijkl00   =   ghijklfe
000000dc   +   mnopqr00   =   mnoprqdc
000000ba   +   stuvwx00   =   stuvwxba

             -------------
```

Tada! Four 6-bit bytes

```
  00abcdef
  00ghijkl
  00mnopqr
  00stuvwx
```

become three 8-bit bytes

```
  ghijklfe
  mnoprqdc
  stuvwxba
```

When DOS 3.3 reads a sector, it reads the first $56 raw nibbles, decoded them into 6-bit bytes, and stashes them in a temporary buffer (at $BC00). Then it reads the other $100 raw nibbles, decodes them into 6-bit bytes, and puts them in another temporary buffer (at $BB00). Only then does DOS 3.3 start combining the bits from each group to create the full 8-bit bytes that will end up in the target page in memory. This is why DOS 3.3 "misses" sectors when it's reading, because it's busy twiddling bits while the disk is still spinning.

The routine at $1114, which I'm about to show you, also uses "6-and-2" encoding. The first $56 nibbles in the data field are still split into pairs of bits that need to be merged with nibbles that won't come until later. But instead of waiting for all $156 raw nibbles to be read from disk, it "interleaves" the nibble reads with the bit twiddling required to merge the first $56 6-bit bytes and the $100 that follow. By the time it gets to the data field checksum, it has already stored all $100 8-bit bytes in their final resting place in memory.

To make it possible to do all the bit twiddling we need to do and not miss nibbles as the disk spins(*), some of the work is already done. We take each of the 64 possible decoded values and multiply by 4 and store them. (Since this is accomplished by bit shifting and we're doing it before we start reading the disk, this is called the "pre-shift" table.) We also store all possible 2-bit values in a repeating pattern that will make it easy to look them up later. Then, as we're reading from disk (and timing is tight), we can simulate all the bit math we need to do with a series of table lookups. There is just enough time to convert each raw nibble into its final 8-bit byte before reading the next nibble.

(*) The disk spins independently of the CPU, and we only have a limited time to read a nibble and do what we're going to do with it before WHOOPS HERE COMES ANOTHER ONE. So time is of the essence. Also, "As The Disk Spins" would make a great name for a retrocomputing-themed soap opera. I am going to continue making this joke until someone makes it happen, then I promise I will stop.

The first table, at $1300..$13FF, is
three columns wide and 64 rows deep.
Astute readers will notice that 3 x 64
is not 256. Only three of the columns
are used; the fourth (unused) column
exists because multiplying by 3 is hard
but multiplying by 4 is easy (in base 2
anyway). The three columns correspond
to the three pairs of 2-bit values in
those first $56 6-bit bytes. Since the
values are only 2 bits wide, each
column holds one of four different
values (%00, %01, %10, or %11).

The second table, at $1296..$12FF, is
the "pre-shift" table. This contains
all the possible 6-bit bytes, in order,
each multiplied by 4 (a.k.a. shifted to
the left two places, so the 6 bits that
started in columns 0-5 are now in
columns 2-7, and columns 0 and 1 are
zeroes). Like this:

```
       00ghijkl   -->   ghijkl00
```

Astute readers will notice that there
are only 64 possible 6-bit bytes, but
this second table is larger than 64
bytes. To make lookups easier, the
table has empty slots for each of the
invalid raw nibbles. In other words, we
don't do any math to decode raw nibbles
into 6-bit bytes; we just look them up
in this table (offset by $96, since
that's the lowest valid raw nibble) and
get the required bit shifting for free.

addr	raw	decoded 6-bit	pre-shift
$1296	$96	0 = %00000000	%00000000
$1297	$97	1 = %00000001	%00000100
$1298	$98	[invalid raw nibble]	
$1299	$99	[invalid raw nibble]	
$129A	$9A	2 = %00000010	%00001000
$129B	$9B	3 = %00000011	%00001100
$129C	$9C	[invalid raw nibble]	
$129D	$9D	4 = %00000100	%00010000
.			
.			
.			
$12FE	$FE	62= %00111110	%11111000
$12FF	$FF	63= %00111111	%11111100

Each value in this "pre-shift" table also serves as an index into the first table (with all the 2-bit bytes). The table of 2-bit bytes is arranged in such a way that we take one of the raw nibbles that needs to be decoded and split apart (from the first $56 raw nibbles in the data field), use that raw nibble as an index into the pre-shift table, then use that pre-shifted value as an index into the first table to get the 2-bit value we need. That's a neat trick.

6 + 2 =
Chapter 8

Now then, what's at $1114?

```
*1114L

; whole lotta self-modification here
1114-   86 27       STX   $27
1116-   8A          TXA
1117-   09 8C       ORA   #$8C
1119-   8D 3D 11    STA   $113D
111C-   8D 48 11    STA   $1148
111F-   8D 53 11    STA   $1153
1122-   8D 5F 11    STA   $115F
1125-   8D 6C 11    STA   $116C
1128-   8D 82 11    STA   $1182
112B-   8D 99 11    STA   $1199
112E-   8D AF 11    STA   $11AF
1131-   8D C3 11    STA   $11C3
1134-   8D D8 11    STA   $11D8
1137-   8D EA 11    STA   $11EA
113A-   EA          NOP
113B-   EA          NOP
```

In a standard DOS 3.3 RWTS, the softswitch to read the data latch is "LDA $C08C,X", where X is the boot slot times 16 (to allow disks to be in any slot). This routine also supports reading from any slot, but instead of using an index, each fetch instruction is preset based on the boot slot.

Not only does this free up the X register, it lets us juggle all the registers and put the raw nibble value in whichever one is convenient at the time. I've marked each softswitch with "o_O" to remind you that self-modifying code is awesome.

There are several other instances of
addresses and constants that get
modified while the routine is running.
I've marked these with "/!\" to remind
you that self-modifying code is
dangerous and you should not try this
at home.

```
; match a custom prologue "D5 AA DB"
113C-   AD 00 C0     LDA   $C000      o_O
113F-   10 FB        BPL   $113C
1141-   49 D5        EOR   #$D5
1143-   D0 F5        BNE   $113A
1145-   EA           NOP
1146-   EA           NOP
1147-   AD 00 C0     LDA   $C000      o_O
114A-   10 FB        BPL   $1147
114C-   C9 AA        CMP   #$AA
114E-   D0 F1        BNE   $1141
1150-   EA           NOP
1151-   EA           NOP
1152-   AD 00 C0     LDA   $C000      o_O
1155-   10 FB        BPL   $1152
1157-   C9 DB        CMP   #$DB
1159-   D0 E6        BNE   $1141
115B-   38           SEC
115C-   A0 AA        LDY   #$AA

; get a 4-and-4-encoded value
115E-   AD 00 C0     LDA   $C000      o_O
1161-   10 FB        BPL   $115E
1163-   2A           ROL
1164-   8D 71 11     STA   $1171
1167-   A9 09        LDA   #$09
1169-   85 2E        STA   $2E
116B-   AD 00 C0     LDA   $C000      o_O
116E-   10 FB        BPL   $116B

; modified at $1164
1170-   29 FF        AND   #$FF       /!\
; store the decoded 4-and-4 value
; later in this routine
1172-   8D D6 11     STA   $11D6

; and two other places, but minus 1
1175-   E9 01        SBC   #$01
1177-   8D BE 11     STA   $11BE
117A-   8D 97 11     STA   $1197
117D-   D0 02        BNE   $1181
```

Loop #1 reads nibbles $00..$55, looks up the corresponding offset in the preshift table at $1296, and stores that offset in a temporary buffer at $0D00.

```
; initialize rolling checksum to $00
117F-   85 2E        STA   $2E

; read a raw nibble from disk
1181-   AE 00 C0     LDX   $C000      o_O
1184-   10 FB        BPL   $1181

; The nibble value is in the X register
; now. The lowest possible nibble value
; is $96 and the highest is $FF. To
; look up the offset in the table at
; $1296, we subtract $96 from $1296 and
; add X.
1186-   BD 00 12     LDA   $1200,X

; Now the accumulator has the offset
; into the table of individual 2-bit
; combinations ($1300..$13FF). Store
; that offset in the temporary buffer
; at $0D00, in the order we read the
; nibbles. But the Y register started
; counting at $AA, so we subtract $AA
; from $0D00 and add Y.
1189-   99 56 0C     STA   $0C56,Y
```

```
; The EOR value is set at $117F
; each time through loop #1.
118C-   45 2E       EOR   $2E
118E-   C8          INY
118F-   D0 EE       BNE   $117F
```

Here endeth loop #1.

Loop #2 reads nibbles $56..$AB, combines them with bits 0-1 of the appropriate nibble from the first $56, and stores them in bytes $00..$55 of the target page in memory (which was self-modified earlier, based on the single 4-and-4 encoded value we read after the "D5 AA DB" prologue).

```
1191-   A0 AA       LDY   #$AA
1193-   D0 03       BNE   $1198

; modified at $117A (based on the
; same target page, but minus 1
; so we can add Y from $AA..$FF)
1195-   99 55 0F    STA   $0F55,Y    /!\
1198-   AE 00 C0    LDX   $C000      o_O
119B-   10 FB       BPL   $1198
119D-   5D 00 12    EOR   $1200,X
11A0-   BE 56 0C    LDX   $0C56,Y
11A3-   5D 00 13    EOR   $1300,X
11A6-   C8          INY
11A7-   D0 EC       BNE   $1195
11A9-   48          PHA
```

Here endeth loop #2.

Loop #3 reads nibbles $AC..$101,
combines them with bits 2-3 of the
appropriate nibble from the first $56,
and stores them in bytes $56..$AB of
the target page in memory.

```
11AA-   29 FC          AND   #$FC
11AC-   A0 AA          LDY   #$AA
11AE-   AE 00 C0       LDX   $C000      o_O
11B1-   10 FB          BPL   $11AE
11B3-   5D 00 12       EOR   $1200,X
11B6-   BE 56 0C       LDX   $0C56,Y
11B9-   5D 01 13       EOR   $1301,X

; modified at $1177 (based on the
; same target page, but minus 1
; so we can add Y from $AA..$FF)
11BC-   99 AC 0F       STA   $0FAC,Y    /!\
11BF-   C8             INY
11C0-   D0 EC          BNE   $11AE
```

Here endeth loop #3.

Loop #4 reads nibbles $102..$155,
combines them with bits 4-5 of the
appropriate nibble from the first $56,
and stores them in bytes $AC..$FF of
the target page in memory.

```
11C2-   AE 00 C0    LDX   $C000     o_O
11C5-   10 FB       BPL   $11C2
11C7-   29 FC       AND   #$FC
11C9-   A0 AC       LDY   #$AC
11CB-   5D 00 12    EOR   $1200,X
11CE-   BE 54 0C    LDX   $0C54,Y
11D1-   5D 02 13    EOR   $1302,X

; target page (modified at $1172)
11D4-   99 00 10    STA   $1000,Y   /!\

11D7-   AE 00 C0    LDX   $C000     o_O
11DA-   10 FB       BPL   $11D7
11DC-   C8          INY
11DD-   D0 EC       BNE   $11CB
```

Here endeth loop #4.

```
; Finally, the last nibble, which
; is the checksum of all the
; previous nibbles.
11DF-    29 FC          AND   #$FC
11E1-    5D 00 12       EOR   $1200,X
11E4-    A6 27          LDX   $27
11E6-    A8             TAY

; if checksum fails, branch forward
; to set the carry flag to indicate
; to the caller that the read failed
11E7-    D0 09          BNE   $11F2

; match a custom epilogue "BE"
11E9-    AD 00 C0       LDA   $C000      o_O
11EC-    10 FB          BPL   $11E9
11EE-    C9 BE          CMP   #$BE
11F0-    F0 03          BEQ   $11F5

; failure path is here (either from
; $11E7 or by falling through if the
; epilogue doesn't match) -- just set
; the carry and get out as quickly
; as possible
11F2-    38             SEC
11F3-    B0 01          BCS   $11F6
11F5-    18             CLC

; one final self-modification, just a
; few lines down, to store the final
; byte
11F6-    AD D6 11       LDA   $11D6
11F9-    8D 00 12       STA   $1200
11FC-    A8             TAY
11FD-    68             PLA

; modified at $11F9
11FE-    8D 55 10       STA   $1055      /!\
1201-    60             RTS
```

And that's all she wrote^H^H^H^H^Hread.

To sum up: DOS is stored on tracks 0-2, in sectors of standard size ($100 bytes of data each), but the raw nibbles on disk are arranged as

```
   D5 AA DB      ; custom prologue
   xx yy         ; 4-and-4 encoded address
                 ; (high byte only)
   <data field>; 6-and-2 encoded, just
                 ; like a DOS 3.3 sector
   BE            ; custom epilogue
```

No address prologue. No address field. No address epilogue. Just a custom prologue, one byte worth of addressing information, and 6-and-2 encoded data.

Chapter 9
In Which We See The Light At The End Of The Tunnel, And It Turns Out To Be A Misshapen DOS, Which Is A Weird Thing To See At The End Of A Tunnel, Really

Revisiting the caller at $108C, I can now better understand what's going on.

```
; read from track 0
108C-    AD 1E 10     LDA   $101E
108F-    AE 21 10     LDX   $1021

*101E
101E- 06

*1021
1021- 00

; read $06 sectors from track $00 and
; fill slots $00..$05 in the array at
; $02D8
1092-    20 24 10     JSR   $1024
1095-    B0 59        BCS   $10F0

; advance to track $01
1097-    20 13 10     JSR   $1013

109A-    AE 22 10     LDX   $1022
109D-    AD 1F 10     LDA   $101F

*101F
101F- 10

*1022
1022- 06

; read $10 sectors from track $01 and
; fill slots $06..$15 in the array at
; $02D8
10A0-    20 24 10     JSR   $1024
10A3-    B0 4B        BCS   $10F0

; advance to track $02
10A5-    20 13 10     JSR   $1013
```

```
10A8-   AE 23 10    LDX   $1023
10AB-   AD 20 10    LDA   $1020

*1020
1020- 0F

*1023
1023- 16

; read $0F sectors from track $02 and
; fill slots $16-$24 in the array at
; $02D8
10AE-   20 24 10    JSR   $1024
10B1-   B0 3D       BCS   $10F0
```

So we're reading a total of $25 sectors from tracks 0-2. Still don't know where we're putting them in memory, but one step at a time.

Continuing from $10B3, after we've read whatever we're going to read...

```
*10B3L

; boot slot (x16)
10B3-   A6 2B       LDX   $2B
10B5-   8E C7 3F    STX   $3FC7
10B8-   8E D5 3F    STX   $3FD5
10BB-   A9 01       LDA   #$01
10BD-   8D D6 3F    STA   $3FD6
10C0-   8D C8 3F    STA   $3FC8

; ??
10C3-   A9 AD       LDA   #$AD
10C5-   85 31       STA   $31
```

```
; looks like we're initializing a DOS-
; shaped RWTS (these are the markers
; that keep track of which track we're
; on, to prevent that grinding noise
; when the disk ends up on the wrong
; track and has to recalibrate)
10C7-   8A          TXA
10C8-   4A          LSR
10C9-   4A          LSR
10CA-   4A          LSR
10CB-   4A          LSR
10CC-   AA          TAX
10CD-   A9 04       LDA   #$04
10CF-   9D F8 04    STA   $04F8,X
10D2-   9D 78 04    STA   $0478,X
10D5-   A2 FF       LDX   #$FF
10D7-   9A          TXS
10D8-   8E C9 3F    STX   $3FC9

; machine initialization (NORMAL, PR#0,
; IN#0, TEXT, HOME, &c.)
10DB-   20 84 FE    JSR   $FE84
10DE-   20 89 FE    JSR   $FE89
10E1-   20 93 FE    JSR   $FE93
10E4-   20 2F FB    JSR   $FB2F
10E7-   20 58 FC    JSR   $FC58
10EA-   A0 03       LDY   #$03
10EC-   A9 1B       LDA   #$1B

; unconditional branch
10EE-   D0 0A       BNE   $10FA
```

```
; entry point for failures (from many
; places, including any disk read
; failures)
10F0-   A5 2B          LDA   $2B
10F2-   4A             LSR
10F3-   4A             LSR
10F4-   A0 00          LDY   #$00
10F6-   4A             LSR
10F7-   4A             LSR
10F8-   09 C0          ORA   #$C0

; execution continues here (from $10EE)
10FA-   48             PHA
10FB-   98             TYA
10FC-   48             PHA
10FD-   48             PHA
10FE-   40             RTI
```

OK, so the success path (via $10EA) pushes ##$1B, ##$1B, and ##$03, then does an "RTI". (Boy, these developers really love their "RTI".) The failure path (via $10F0) pushes ##$Cn based on the boot slot, then ##$00 twice. The "RTI" will either "return" to $1B03 or $Cn00.

I would guess that DOS is loaded into lower memory ($1B00..$3FFF) then moved to higher memory on machines that have it. $1B03 is the standard entry point for DOS 3.3 to relocate itself to higher memory, say from $1B00 to $9B00. But I can't verify that just by looking at the code, because the address that each sector is loaded into is encoded on the disk itself.

Un. Frickin'. Believable.

```
*9600<C600.C6FFM

; set up callback #1 before the "RTI"
96F8-   A9 4C       LDA   #$4C
96FA-   8D 55 08    STA   $0855
96FD-   A9 0A       LDA   #$0A
96FF-   8D 56 08    STA   $0856
9702-   A9 97       LDA   #$97
9704-   8D 57 08    STA   $0857

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

; (callback #1 is here)
; set up callback #2 after we've loaded
; the next few tracks
970A-   A9 4C       LDA   #$4C
970C-   8D E7 10    STA   $10E7
970F-   A9 1C       LDA   #$1C
9711-   8D E8 10    STA   $10E8
9714-   A9 97       LDA   #$97
9716-   8D E9 10    STA   $10E9

; continue the boot
9719-   4C 10 12    JMP   $1210

; (callback #2 is here)
; turn off the drive motor and reboot
; to my work disk
971C-   AD E8 C0    LDA   $C0E8
971F-   4C 00 C5    JMP   $C500

*BSAVE TRACE5,A$9600,L$122
```

```
; clear memory
*800:FD N 801<800.BEFEM

; and go
*BRUN TRACE5
...reboots slot 6...
...read read read...
...reboots slot 5...]
CALL -151
...
```

After some manual inspection, I confirmed that the only range modified was $1B00..$3FFF.

```
*BSAVE OBJ.1B00-3FFF,A$1B00,L$2500

*3D00L

3D00-   00          BRK
3D01-   00          BRK
3D02-   00          BRK
```

Uh oh.

Poking around in memory, this DOS is not, well, DOS-shaped. I would expect $3800..$3FFF to be the RWTS, with an entry point at $3D00. Obviously, it's not there. $3944 to match the address prologue? No. $38DC to match the data prologue? Also no. $1D84 to cold boot DOS? Nope nope nope.

I did finally find what appears to be the RWTS entry point, at $36D5 (WTF):

```
*36D5L

36D5-   84 48       STY   $48
36D7-   85 49       STA   $49
36D9-   A0 02       LDY   #$02
36DB-   8C F8 06    STY   $06F8
36DE-   A0 04       LDY   #$04
36E0-   8C F8 04    STY   $04F8
36E3-   A0 01       LDY   #$01
36E5-   B1 48       LDA   ($48),Y
36E7-   AA          TAX
36E8-   A0 0F       LDY   #$0F
36EA-   D1 48       CMP   ($48),Y
36EC-   F0 1B       BEQ   $3709
...
```

And the routine that matches the data prologue and decodes the data field, at $3895:

```
*3895L

3895-   A0 20       LDY   #$20
3897-   88          DEY
3898-   F0 72       BEQ   $390C
389A-   BD 8C C0    LDA   $C08C,X
389D-   10 FB       BPL   $389A
389F-   49 D5       EOR   #$D5
38A1-   D0 F4       BNE   $3897
38A3-   EA          NOP
38A4-   BD 8C C0    LDA   $C08C,X
38A7-   10 FB       BPL   $38A4
38A9-   C9 AA       CMP   #$AA
38AB-   D0 F2       BNE   $389F
38AD-   48          PHA
38AE-   68          PLA
```

```
; Mega weirdness here. The third nibble
; must be $EE, otherwise we branch back
; to the beginning of the data prologue
; check. But then we also check for $AD
; and, if found, branch to the middle
; of an instruction!
38AF-   BD 8C C0    LDA   $C08C,X
38B2-   10 FB       BPL   $38AF
38B4-   C9 EE       CMP   #$EE
38B6-   D0 E7       BNE   $389F
38B8-   49 AD       EOR   #$AD
38BA-   F0 0B       BEQ   $38C7 ----+
38BC-   D0 00       BNE   $38BE     |
38BE-   BC 8C C0    LDY   $C08C,X   |
38C1-   10 FB       BPL   $38BE     |
38C3-   B9 00 3B    LDA   $3B00,Y   |
38C6-   2C A9 00    BIT   $00A9   <-+
```

Meanwhile, there's a loop at $38BE that gets an extra nibble from disk and uses it as an index into a table to set the starting value of the data checksum (in the accumulator). That's all kinds of messed up. I've never seen anything like it. A few disks use a non-standard data checksum value, but it's constant, not something that varies per sector. That's insane.

Also, in my initial investigation with a nibble editor, $EE was not the third value of the data field prologue. It varies per track, but it's never $EE. So this code may be modified mid-RWTS, possibly in the track change routine (wherever that is).

Finally, the routine that matches the address prologue and parses the address field starts at $39CD. Not listed here, but it's completely normal. I checked every byte to make sure. Every. Single. Byte.

Now let's put it to good use.

MY NAME IS MR. SNUFFLE-UPAGUS.

Chapter 10
In Which We Grandiloquently Announce That We Will Be Using The Original Disk As A Weapon Against Itself

Given an RWTS that can read a disk --
no matter how crazy that RWTS may be --
the cracker's tool of choice is
Advanced Demuffin. I've included a copy
of the latest version on my work disk.

```
*BLOAD ADVANCED DEMUFFIN 1.5
```

Since this RWTS does not have the usual
entry point (at $BD00), I get to write
an IOB module to tell Advanced Demuffin
how to call the custom RWTS.

```
; load disk's DOS and RWTS in place
*BLOAD OBJ.1B00-3FFF

*BLOAD ADVANCED DEMUFFIN 1.5

*1400L

; standard Advanced Demuffin setup
; (unchanged)
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

; change Advanced Demuffin to read into
; memory starting at $4000 instead of
; $2000, to avoid overwriting the DOS
; and RWTS that end at $3FFF
1412-   A9 40       LDA   #$40
1414-   8D F0 1C    STA   $1CF0
```

```
; call the RWTS entry point
1417-   A9 0F         LDA   #$0F
1419-   A0 1E         LDY   #$1E
141B-   4C D5 36      JMP   $36D5

*BSAVE IOB,A$1400,L$100

; launch Advanced Demuffin
*800G

["C" to convert disk]

["Y" to change default values]

                 --v--

ADVANCED DEMUFFIN 1.5    (C) 1983, 2014
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======================================


INPUT ALL VALUES IN HEX


SECTORS PER TRACK? (13/16) 16

START TRACK: $03        <-- change this
START SECTOR: $00
END TRACK: $22
END SECTOR: $0F

INCREMENT: 1

MAX # OF RETRIES: 0

COPY FROM DRIVE 1
TO DRIVE: 2
=======================================
16SC $03,$00-$22,$0F BY1.0 S6,D1->S6,D2

                 --^--
```

```
[S6,D1=original disk]
[S6,D2=blank formatted disk]

And here we go...
```

```
                 --v--

ADVANCED DEMUFFIN 1.5    (C) 1983, 2014
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======PRESS ANY KEY TO CONTINUE=======
TRK:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
+.5:
    0123456789ABCDEF0123456789ABCDEF012
SC0:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRR.
SC1:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC2:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC3:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC4:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC5:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC6:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC7:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC8:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SC9:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SCA:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SCB:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SCC:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SCD:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SCE:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
SCF:   RRRRRRRRRRRRRR.RRRRRRRRRRRRRRRRRR
=======================================
16SC $03,$00-$22,$0F BY1.0 S6,D1->S6,D2

                 --^--
```

Those would be read errors on every sector of every track -- except track $11, which I could already read because it's standard. Gotta be honest; this falls quite short of my expectations of the code that can allegedly read every sector of every track.

Let's back up.

Chapter 11
In Which We Back Up

My first clue that this RWTS was not
going to work without some fiddling
should have been this suspicious "STA"
at $10C5:

```
10C3-   A9 AD        LDA   #$AD
10C5-   85 31        STA   $31
```

Why this is suspicious: I've seen many,
many disk reading routines that use
zero page $31 as a way to vary one of
the nibbles in the address or data
prologue (or both).

Why this is not suspicious: this RWTS
doesn't ever use zp$31. The only thing
even slightly unusual about the RWTS
(other than the fact that everything is
in the wrong place) is the weirdness
around $38AF, matching the third nibble
of the data prologue.

```
38AF-   BD 8C C0     LDA   $C08C,X
38B2-   10 FB        BPL   $38AF
38B4-   C9 EE        CMP   #$EE    <-- !
38B6-   D0 E7        BNE   $389F
38B8-   49 AD        EOR   #$AD
38BA-   F0 0B        BEQ   $38C7 ; never
38BC-   D0 00        BNE   $38BE
```

Combining these two suspicious things,
I used my trusty Copy][+ sector editor
to search the captured file for any
references to addresses around $38AF,
and I eventually found this code:

```
2ADD-   A9 C5         LDA   #$C5
2ADF-   8D B4 38      STA   $38B4    <-- !
2AE2-   A9 31         LDA   #$31
2AE4-   8D B5 38      STA   $38B5    <-- !
```

Presumably executed during DOS startup,
this changes the "CMP #$EE" at $38B4
which was confusing me earlier. (This
code is not being executed by Advanced
Demuffin, which tells me that it's a
one-time change that happens outside
the RWTS. Just because f--- you.)

So by the time the disk goes to read a
sector, the code at $38AF will actually
look like this:

```
38AF-   BD 8C C0      LDA   $C08C,X
38B2-   10 FB         BPL   $38AF
38B4-   C5 31         CMP   $31
38B6-   D0 E7         BNE   $389F
38B8-   49 AD         EOR   #$AD
38BA-   F0 0B         BEQ   $38C7
38BC-   D0 00         BNE   $38BE
```

Now this makes slightly more sense. The
value of zp$31 could be #$AD, in which
case the EOR/BEQ at $38B8 would match
and we might end up at $38C7. But if
zp$31 is anything other than #$AD, we
end up at $38BE and read that extra
nibble that determines the expected
data field checksum.

Oh my God. This is why track $11 was
perfectly readable on the original disk
(even by Locksmith Fast Disk Backup,
which is not at all forgiving about any
deviations from the norm). Look, look:

```
; third data prologue nibble
38AF-   BD 8C C0    LDA   $C08C,X
38B2-   10 FB       BPL   $38AF

; must match current value of zp$31
38B4-   C5 31       CMP   $31

; otherwise we start over
38B6-   D0 E7       BNE   $389F

; if the prologue nibble (and zp$31)
; are #$AD, branch into the middle of
; a later instruction
38B8-   49 AD       EOR   #$AD
38BA-   F0 0B       BEQ   $38C7 ----+
38BC-   D0 00       BNE   $38BE     |
                                    |
; read the extra nibble             |
38BE-   BC 8C C0    LDY   $C08C,X   |
38C1-   10 FB       BPL   $38BE     |
                                    |
; look up the expected checksum     |
38C3-   B9 00 3B    LDA   $3B00,Y   |
                                    |
; hide an instruction               |
38C6-   2C A9 00    BIT   $00A9   <-+
```

What's the hidden instruction at $38C7?

```
38C7-   A9 00       LDA   #$00
```

This disk supports perfectly normal tracks. As long as zp$31 is #$AD, it will skip the extra nibble, set the accumulator to 0, and continue decoding the data field. At some point before doing any disk catalog work (on track $11), it must be setting zp$31 to #$AD and letting the RWTS take the branch to $38C7.

Now convinced that zp$31 plays a vital role in this RWTS, I searched the same file for references to zp$31. You'll never guess what happened next!

```
; find the last byte in an array that
; is NOT #$A0 (space character)
2604-   A2 1D       LDX   #$1D
2606-   A9 A0       LDA   #$A0
2608-   DD 69 1F    CMP   $1F69,X
260B-   D0 04       BNE   $2611
260D-   CA          DEX
260E-   10 F8       BPL   $2608
2610-   E8          INX

; munge that (MOD $10)
2611-   8A          TXA
2612-   29 0F       AND   #$0F

; and store it in zp$31!
2614-   85 31       STA   $31

; oh, but also take the last non-space
; character we found
2616-   BD 69 1F    LDA   $1F69,X

; munge that (MOD $20)
2619-   29 1F       AND   #$1F
```

```
; add that to the first thing
261B-   18            CLC
261C-   65 31         ADC   $31
261E-   AA            TAX

; and use that as a lookup into another
; array
261F-   BD 62 3D      LDA   $3D62,X

; and put THAT in zp$31
2622-   85 31         STA   $31
2624-   4C 77 28      JMP   $2877
```

I'm not sure when this is getting called (and my usual references, like "Beneath Apple DOS," are useless since everything is in the wrong place in memory), but I can easily see what is at $1F69:

```
*1F69.

1F69- .. CD C5 CE D5 A0 A0 A0 ; "MENU"
1F70- A0 A0 A0 A0 A0 A0 A0 A0 ;(spaces)
1F78- A0 A0 A0 A0 A0 A0 A0 A0 ;
1F80- A0 A0 A0 A0 A0 A0 A0 A0 ;
1F88- A0 A0 A0 A0 A0 A0 A0 A0 ;
1F90- A0 A0 A0 A0 A0 A0 A0 A0 ;
```

That's the name of one of the files on disk. (It's probably the startup file.)

So we're counting... spaces? In a file name? Then munging that, combining it with a munged version of the last non-space letter of the filename, and using the final value as a lookup into... what exactly?

```
*3D62.

3D62-  ..  ..  AE  AF  B2  B3  B4  B5
3D68-  B6  B7  B9  BA  BB  BC  BD  BE
3D70-  BF  CB  CD  CE  CF  D3  D6  D7
3D78-  D9  DA  DB  DC  DD  DE  DF  E5
3D80-  E6  E7  E9  EA  EB  EC  ED  EE
3D88-  EF  F2  F3  F4  F5  F6  F7  F9
3D90-  FA  FB  FC  FD  FE  FF  00  00
```

That appears to be part of the nibble translate table, but that's not how we're using it here. Instead, we're (re)using it as an array of possible values for the third nibble of the data prologue.

I was wrong. The data prologue doesn't vary by track; it varies by sector. But it's so much worse than that. It varies by file, based on some combination of the length of the filename and the last letter in the filename.

Which means that this RWTS, which I've spent all this effort to capture, can't actually read the original disk unless I hook it up to the surrounding DOS. Or maybe parse the disk catalog and each file's track/sector lists. Or just give up and start drinking again.

Chapter 12
In Which We Do Not Start Drinking Again (Yet)

Re-re-re-visiting the core of this
problematic RWTS, the third nibble of
the data prologue:

```
38AF-   BD 8C C0    LDA   $C08C,X
38B2-   10 FB       BPL   $38AF
38B4-   C5 31       CMP   $31
38B6-   D0 E7       BNE   $389F
38B8-   49 AD       EOR   #$AD
38BA-   F0 0B       BEQ   $38C7 ----+
38BC-   D0 00       BNE   $38BE     |
38BE-   BC 8C C0    LDY   $C08C,X   |
38C1-   10 FB       BPL   $38BE     |
38C3-   B9 00 3B    LDA   $3B00,Y   |
38C6-   2C A9 00    BIT   $00A9   <-+
```

There are two mutually exclusive
conditions

1. zp$31 is #$AD, in which case the
   track is entirely normal (like
   track $11), or

2. zp$31 is not #$AD, in which case
   the track is entirely f*cked
   (non-standard data prologue, extra
   nibble before data field that
   serves as the checksum, &c.)

But in the second condition, we can be
more specific. Based on our research in
the calling function (shown above at
$2604), the range of possible values
for the third nibble of a non-standard
data prologue is #$AE..#$FF -- in other
words, always greater than the standard
value of #$AD.

This gives me an idea.

I've seen other disks that use a trick
to allow reading of data disks (which
are generally in a standard format) and
the master disk (protected) in the same
RWTS. The epilogue bytes of the master
disk are "FF FF FF" (instead of the
standard "DE AA EB"), then the RWTS
code to match epilogue bytes looks like
this (example listing taken from 4am
crack no. 541, "Survey Taker"):

```
B92F-   BD 8C C0     LDA   $C08C,X
B932-   10 FB        BPL   $B92F
B934-   C9 DE        CMP   #$DE
B936-   90 0A        BCC   $B942   <-- !
B938-   EA           NOP
B939-   BD 8C C0     LDA   $C08C,X
B93C-   10 FB        BPL   $B939
B93E-   C9 AA        CMP   #$AA
B940-   B0 5C        BCS   $B99E   <-- !
B942-   38           SEC
B943-   60           RTS
```

Did you see it? BCC instead of BNE at
$B936, and BCS instead of BEQ at $B940.
This will accept the standard epilogue
"DE AA", but it will also accept "FF FF
(or anything in between), because those
values are greater than the comparison
values.

BCC and BCS operations take exactly the
same amount of time as BNE and BEQ
operations, so the delicate CPU count
is preserved. (Remember, this low-level
RWTS code is sensitive to any timing
changes, since the disk being read is
spinning independently of the CPU
trying to read it as it goes by.)

So, getting back to this disk's RWTS, I can imagine a patch that would turn the data prologue matching code into

1. third nibble = #$AD? --> branch to standard path

2. third nibble > #$AD? --> fall through to read extra nibble

This has several advantages. Firstly, it doesn't involve parsing the disk catalog, which I *really* would not have enjoyed. Secondly, it should work on every track -- even the normal track $11 -- so I can convert the disk in one shot. Thirdly, and related to the previous "ly", I could use this patch on the converted disk and retain the rest of the DOS unmodified, which means I don't have to worry about any DOS-level modifications (like non-standard command names or any other weirdness) that I don't even know about yet.

Minimal. Elegant. Might even work.

```
]PR#5
...
]BLOAD OBJ.1B00-3FFF
]CALL -151

*38B6: C9 AD 90 E5

*38AFL

; read raw nibble from disk
38AF-   BD 8C C0    LDA   $C08C,X
38B2-   10 FB       BPL   $38AF

; compare to zp$31
38B4-   C5 31       CMP   $31

; ignore the previous compare; instead
; compare to the constant value #$AD
38B6-   C9 AD       CMP   #$AD

; if it's less than #$AD, something is
; horribly wrong, so branch back and
; start over
38B8-   90 E5       BCC   $389F

; if it's equal to #$AD, branch forward
; to the "standard" path
38BA-   F0 0B       BEQ   $38C7

; if we're here, it must be greater
; than #$AD, which means this is a
; "special" sector, so branch forward
; to the "special" path
38BC-   D0 00       BNE   $38BE
38BE-   BC 8C C0    LDY   $C08C,X
38C1-   10 FB       BPL   $38BE
38C3-   B9 00 3B    LDA   $3B00,Y
38C6-   2C A9 00    BIT   $00A9
```

We've turned the original disk's RWTS, which only worked if the surrounding DOS set zp$31 properly based on the current filename, into a "universal" RWTS that can read any sector on this disk (track $03+). This patch has one final advantage: it can also read the converted disk, which gives us the opportunity to reuse the original DOS (with this 4-byte patch) on the final product.

```
*BSAVE OBJ.1B00-3FFF PATCHED
```

(Diversi-DOS 64K automatically adds the starting address and length of the last file you loaded.)

```
*BRUN ADV=
```

(Diversi-DOS 64K supports wildcards in filenames.)

```
[S6,D1=original disk]
[S6,D2=blank disk]
[S5,D1=my work disk]
]PR#5
...

]BRUN ADVANCED DEMUFFIN 1.5

["5" to switch to slot 5]

["I" to load an IOB module]
  --> load "IOB" from drive 1

["6" to switch to slot 6]

["C" to convert disk]

["Y" to change default values]
```

```
                 --v--

ADVANCED DEMUFFIN 1.5    (C) 1983, 2014
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======================================



INPUT ALL VALUES IN HEX



SECTORS PER TRACK? (13/16) 16

START TRACK: $03        <-- change this
START SECTOR: $00
END TRACK: $22
END SECTOR: $0F

INCREMENT: 1

MAX # OF RETRIES: 0

COPY FROM DRIVE 1
TO DRIVE: 2
=======================================
16SC $03,$00-$22,$0F BY1.0 S6,D1->S6,D2

                 --^--

[S6,D1=original disk]
[S6,D2=blank formatted disk]
```

And here we go...

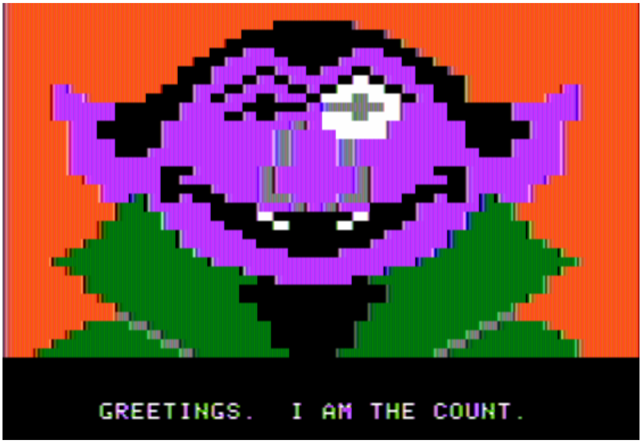
```
                 --v--

ADVANCED DEMUFFIN 1.5    (C) 1983, 2014
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======PRESS ANY KEY TO CONTINUE=======
TRK:   ................................
+.5:
    0123456789ABCDEF0123456789ABCDEF012
SC0:   ................................
SC1:   ................................
SC2:   ................................
SC3:   ................................
SC4:   ................................
SC5:   ................................
SC6:   ................................
SC7:   ................................
SC8:   ................................
SC9:   ................................
SCA:   ................................
SCB:   ................................
SCC:   ................................
SCD:   ................................
SCE:   ................................
SCF:   ................................
=======================================
16SC $03,$00-$22,$0F BY1.0 S6,D1->S6,D2

                 --^--
```

Wa-frickin'-hoo.

Now let's see if it, you know, actually works.

```
[S6,D1=DOS 3.3 system master]
[S6,D2=demuffin'd copy (still)]

]PR#6
...
]RUN MENU,D2
```

The game loads and runs -- even from drive 2! That tells me there are no DOS modifications like renamed commands or direct calls to any of the non-standard entry points. Also, there don't appear to be any secondary protection checks to verify that we booted through the original bootloader. Each phase of the boot is independent from the previous phase. Hooray for abstractions!



Chapter 13
In Which We Reach For The Brass Ring

So that's it, right? We're done. Slap a
standard DOS 3.3 on this puppy(*),
declare victory, and go to the gym.

Well, OK. But I don't crack disks in
isolation anymore. In my inbox, I have
twelve other unpreserved disks with
copy protection identical to this one.
Which means there are 10x that many,
waiting to be discovered and preserved.

Because, you see, this was not a
one-off protection scheme for Ernie's
Quiz. DOS 3.3P was a turnkey solution
developed and licensed by Apple
Computer. Yes, *that* Apple Computer.
The Apple Computer, maker of the
hardware on which this disk runs. They
licensed the protection to several
companies and used it on their own
"Special Delivery" label.

I don't want to crack one thing. I want
to crack all the things.

So now I get to think about automation.

(*) No puppies were harmed in the
    making of this crack. Yet.

In case you missed it, I released an
fully automated cracking tool earlier
this year. It's called "Passport":

http://archive.org/details/Passport4am

Passport automates a four-step process:

1. IDENTIFY the bootloader by reading
   T00,S00

2. TRACE the boot to capture the RWTS

3. CONVERT the disk by reading it
   with its own RWTS and writing a
   copy in a standard format

4. PATCH the copy so it can read
   itself, and disable any runtime
   protection checks so it can boot

DOS 3.3P is easily identifiable. The
boot sector is identical across all my
samples, so a simple pattern match will
suffice.

I've already determined how to trace
the bootloader far enough to capture
the DOS and RWTS. The DOS lives at
$1B00..$3FFF, which is free space.
(Passport occupies $4000..$B5FF.) It
gets a bit tricky because disks might
be legitimately damaged, but I can trap
the failure path at $10F0 to ensure
Passport always regains control.

I know how to call the DOS 3.3P RWTS; I
made an IOB module for Advanced
Demuffin, and I can build that same
logic into Passport.

Which brings us to step 4: the patching of the shrew. Passport doesn't include a copy of DOS 3.3; it assumes copies will reuse the original bootloader, RWTS, and DOS (with a little patching). I know how to patch the RWTS, but I can't reuse the original bootloader, because it's so heavily intertwined with the custom data format(s) on tracks 0-2.

Obviously, we need a new bootloader.

Chapter 14
Introducing "Standard Delivery"

Since so many of the disks with this
copy protection were shipped under
Apple's "Special Delivery" label, I
named this bootloader "Standard
Delivery" in their honor. I wrote the
first version, which fit in $F8 bytes
and hardcoded everything to the way DOS
3.3P needed it. Then qkumba did that
thing he does, and now it's completely
customizable and less than $50 bytes.

```
; The accumulator is the most recently
; read (physical) sector, plus 1.
0801-   A8              TAY

; Self-modify the next instruction to
; Increment the index into the array of
; addresses.
0802-   EE 06 08        INC   $0806

; Get the high byte of the address for
; this sector.
0805-   AD 4E 08        LDA   $084E

; #$C0 means we're completely done and
; it's time to pass control to the next
; phase of the boot. (This value was
; chosen because you can't ever write
; into $C000.)
0808-   C9 C0           CMP   #$C0
080A-   F0 40           BEQ   $084C

; Store the address high byte where the
; drive firmware expects it.
080C-   85 27           STA   $27
```

```
; Y is the physical sector to read. The
; drive firmware increments it by 1 by
; the time we get control (at $0801),
; but we increment it again because we
; want to read every other sector, like
; $00, $02, $04, $06, $08, $0A, $0C,
; $0E, then wrap back around to $01,
; $03, $05, $07, $09, $0B, $0D, $0F.
; This is the fastest order; the drive
; firmware is too slow to read sectors
; in monotonically increasing order.
080E-   C8          INY
080F-   C0 10       CPY   #$10
0811-   90 09       BCC   $081C

; If Y = $10, it means we entered at
; $0801 having just read sector $0E,
; so now we need to wrap around to
; sector $01 and continue reading this
; track. See previous comment for the
; optimal sector order.
0813-   F0 05       BEQ   $081A

; If we fall through to here, it means
; we entered at $0801 having just read
; sector $0F, so we need to advance to
; the next track before starting over
; on sector $00. This subroutine moves
; the drive head to the next track.
0815-   20 2F 08    JSR   $082F

; A=0 on exiting the subroutine, so
; this always sets Y=0.
0818-   A8          TAY

; There's an "LDY #$01" hidden in here,
; which is executed if we take the BEQ
; from $0813. This is how we wrap from
; sector $0E to sector $01.
0819-   2C A0 01    BIT   $01A0
```

```
; Store the sector number where the
; drive firmware expects it.
081C-   84 3D         STY   $3D

; Increment the sector number. This is
; only needed if we're skipping this
; sector (see next comparison), in
; which case we would never call the
; drive firmware for this sector.
081E-   C8            INY

; Check if we actually want to read
; this sector.
081F-   A5 27         LDA   $27

; No, branch back and try the next one.
0821-   F0 DF         BEQ   $0802

; Yes, so exit via the drive firmware
; entry point ($Cn5C, where n is the
; boot slot). This will read the sector
; we set up (sector number in zp$27,
; address in zp$3D) and exit via $0801.
; So this entire thing is a giant loop
; that only exits via the BEQ at $080A.
0823-   8A            TXA
0824-   4A            LSR
0825-   4A            LSR
0826-   4A            LSR
0827-   4A            LSR
0828-   09 C0         ORA   #$C0
082A-   48            PHA
082B-   A9 5B         LDA   #$5B
082D-   48            PHA
082E-   60            RTS
```

```
; This subroutine advances the drive
; head to the next track by twiddling
; the appropriate stepper motors for
; exactly the right amount of time.
082F-   E6 41       INC   $41
0831-   06 40       ASL   $40
0833-   20 37 08    JSR   $0837
0836-   18          CLC
0837-   20 3C 08    JSR   $083C
083A-   E6 40       INC   $40
083C-   A5 40       LDA   $40
083E-   29 03       AND   #$03
0840-   2A          ROL
0841-   05 2B       ORA   $2B
0843-   A8          TAY
0844-   B9 80 C0    LDA   $C080,Y
0847-   A9 30       LDA   #$30
0849-   4C A8 FC    JMP   $FCA8

; Execution continues here (from $080A)
; once we've read all the sectors from
; all the tracks into all the pages in
; memory. As you can see from the array
; of addresses (below), we've read the
; original bootloader code into $1000..
; $13FF and the entire DOS into $1B00..
; $3FFF. Now we jump into the middle of
; the next phase of the original disk's
; bootloader, immediately after it
; read DOS into memory.
084C-   4C B3 10    JMP   $10B3
```

```
; Here is the entire array of addresses
; to read. It's $30 bytes long -- $2F
; addresses (because Standard Delivery
; is smart enough to skip T00,S00) plus
; the end delimiter.
0848-  ..  ..  ..  ..  ..  ..  ..  1E
0850-  1D  1C  1B  00  00  00  00  00
0858-  00  13  12  11  10  1F  20  2E
0860-  2D  2C  2B  2A  29  28  27  26
0868-  25  24  23  22  21  2F  30  3E
0870-  3D  3C  3B  3A  39  38  37  36
0878-  35  34  33  32  31  3F  C0
```

This array says that track $00 contains part of the DOS code at $1B00..$1FFF, phase 2 of the original bootloader at $1000..$13FF, and six unused sectors. Track $01 contains DOS code at $2000..$2FFF. Track $02 contains DOS code at $3000..$3FFF.

Here is the complete disk layout, in the order in which sectors are read:

Track	Sector	Address
$00	$02	$1E00
$00	$04	$1D00
$00	$06	$1C00
$00	$08	$1B00
$00	$0A	---
$00	$0C	---
$00	$0E	---
$00	$01	---
$00	$03	---
$00	$05	---
$00	$07	$1300
$00	$09	$1200
$00	$0B	$1100
$00	$0D	$1000
$00	$0F	$1F00
$01	$00	$2000
$01	$02	$2E00
$01	$04	$2D00
$01	$06	$2C00
$01	$08	$2B00
$01	$0A	$2A00
$01	$0C	$2900
$01	$0E	$2800
$01	$01	$2700
$01	$03	$2600
$01	$05	$2500
$01	$07	$2400
$01	$09	$2300
$01	$0B	$2200
$01	$0D	$2100
$01	$0F	$2F00

[...]

```
 $02   |   $00    |   $3000
 $02   |   $02    |   $3E00
 $02   |   $04    |   $3D00
 $02   |   $06    |   $3C00
 $02   |   $08    |   $3B00
 $02   |   $0A    |   $3A00
 $02   |   $0C    |   $3900
 $02   |   $0E    |   $3800
 $02   |   $01    |   $3700
 $02   |   $03    |   $3600
 $02   |   $05    |   $3500
 $02   |   $07    |   $3400
 $02   |   $09    |   $3300
 $02   |   $0B    |   $3200
 $02   |   $0D    |   $3100
 $02   |   $0F    |   $3F00
```

All that in $4F bytes of code and $30 bytes of data.

And closing the loop, we can see how this integrates with Passport. After Passport traces the boot, captures the DOS, patches the RWTS, and writes out tracks 0-2 according to the table above, it writes Standard Delivery to T00,S00. Then it uses the patched RWTS to convert the rest of the disk (tracks $03-$22).

When you boot the copy, Standard Delivery loads the original bootloader code into $1000..$13FF and the patched RWTS + DOS into $1B00..$3FFF, then jumps to $10B3 to finish the rest of the boot and start the game.

The Passport log will look like this:

--v--

```
READING FROM S6,D1
T00,S00 FOUND DOS 3.3P BOOTLOADER
WRITING TO S5,D2
T02,S08,$B6: D0E749AD -> C9AD90E5
USING DISK'S OWN RWTS
CRACK COMPLETE.
```

--^--

But you know how much complexity is hiding behind that deceptively simple log.

Quod erat liberandum.

Acknowledgments

Thanks to qkumba, John Brooks, John Aycock, Ange Albertini, and Paul Hagstrom for reviewing drafts of this write-up.

Many thanks to qkumba for the "Standard Delivery" bootloader. Find the latest version at http://github.com /peterferrie/standard-delivery