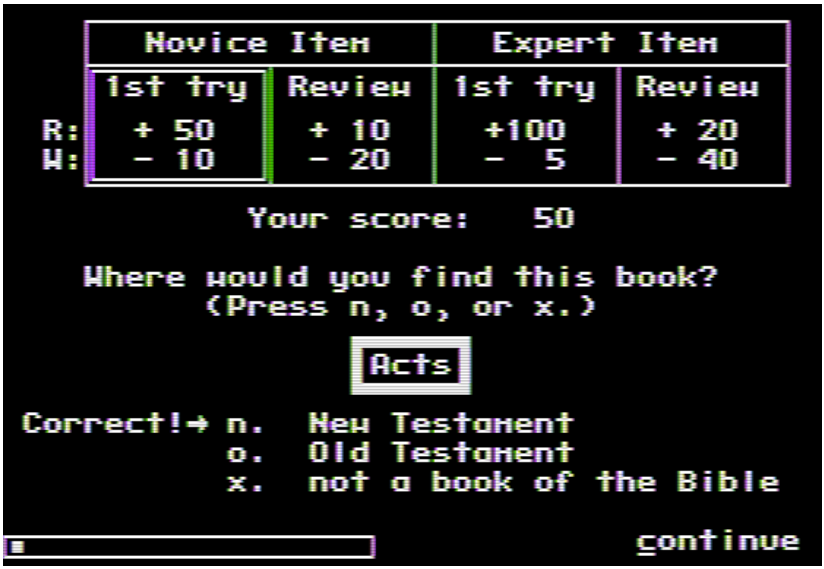
Overview of The Bible



2015-08-07

4am
to deprotect
and preserve

Contents

0 Thou Shalt Not Copy This Floppy 4

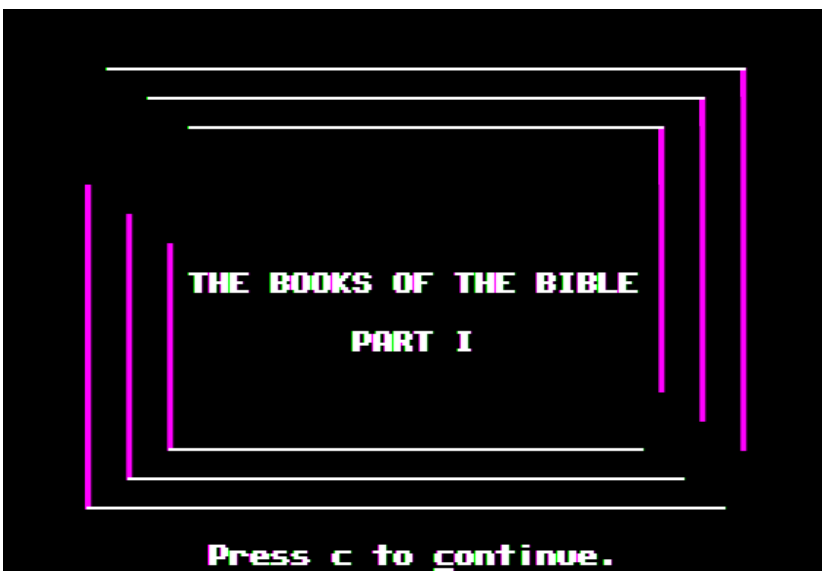
1 God Learns His Lesson, Copy Protects This Apple 8

2 Beware of False Prophets And Boot Sectors 15

3 On The Seventh Day He RWTSed 23

4 These Wineskins Were New When We Filled Them 3

A Epilogue 41



Name: Overview of The Bible
Genre: educational
Year: 1983
Credits:
  Design: Martin A. Siegel
          Elizabeth J. Clapp
  Programming: J. Michael Felty
               Stokely J. Boast
               John G. Chmela
  Content: Rev. Steve Clapp
  A Product of the C-4 Computer Company
Publisher: ADAM Christian Educational
  Software
Media: 3 single-sided 5.25-inch discs
OS: DOS 3.3 variant ("Protected.DOS")
Previous cracks: none
Similar cracks:
  Spell It! (crack no. 138)
  Classmate (crack no. 131)

There are 3 disks, all bootable. They
appear to be independent of each other.

Chapter 0
Thou Shalt Not Copy This Floppy

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 just hangs
  on boot

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

For example, after setting the proper prologues ("D5 AA 97"/"D5 AA B5") and ignoring epilogues, this is what T11,S0F looks like:

--v--

```
--------------- DISK EDIT ---------------
TRACK $11/SECTOR $0F/VOLUME $00/BYTE$00
-----------------------------------------
$00:>58<49 57 01 01 01 01 01   XIWAAAAA
$08: 01 01 01 13 0F 5A 90 C5   AAASOZ.E
$10: CC CC CE A1 A1 A1 A1 A1   LLN!!!!!
$18: A1 A1 A1 A1 A1 A1 A1 A1   !!!!!!!!
$20: A1 A1 A1 A1 A1 A1 A1 A1   !!!!!!!!
$28: A1 A1 A1 A1 09 01 15 0F   !!!!IAUO
$30: 05 CB C0 97 9C F9 F9 F9   EK@..yyy
$38: F9 F9 F9 F9 F9 F9 F9 F9   yyyyyyyy
$40: F9 F9 F9 F9 F9 F9 F9 F9   yyyyyyyy
$48: F9 F9 F9 F9 F9 F9 F9 73   yyyyyyy3
$50: 59 4C 56 5D 83 9C 88 F8   YLV]...x
$58: FA FA FA FA FA FA FA FA   zzzzzzzz
$60: FA F8 F8 F8 F8 F8 F8 F8   zxxxxxxx
$68: FA FA FA FA FA FA FA FA   zzzzzzzz
$70: FA FA 5D 5A 4C 55 5E 97   zz]ZLU^.
$78: 93 91 C7 A2 A2 A2 A2 A2   ..G"""""
-----------------------------------------
BUFFER 0/SLOT 6/DRIVE 1/MASK OFF/NORMAL

-----------------------------------------
COMMAND :
```

--^--

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
God Learns His Lesson, Copy Protects This Apple

```
[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!

(*)not guaranteed, excitement may vary

```
]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
*B700L

B700-   A0 1A       LDY   #$1A
B702-   B9 00 B7    LDA   $B700,Y
B705-   49 54       EOR   #$54
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-   DA          ???
B71B-   BD E3 DA    LDA   $DAE3,X
B71E-   A3          ???
B71F-   E3          ???
B720-   FD 3F D9    SBC   $D93F,X
```

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 54       EOR   #$54
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
On The Seventh Day He RWTSed

```
*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
These Wineskins Were New When We Filled Them

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 epilogue.

```
                    --v--

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

TRACK: 03  START: 38A3  LENGTH: 015F

3880: B5 9A A6 B9 B6 D5 9A A6   VIEW
3888: FC DE AA EB DB DB DB DB
         ^^^^^^^^
      data epilogue

3890: DB DB DB DB DB D7 AA 97
                        ^^^^^^^^
                  address prologue

3898: AA AA AB AB AF AB AE AA
38A0: AF FF FF FF FF FF FF FF  <-38A3
      ^^^^^^^^
  address epilogue

38A8: FF D5 AA B5 CD F3 DF D6
         ^^^^^^^^
      data prologue

38B0: B4 F3 AE AE DF D6 CD ED
38B8: CD FC AE F3 F7 ED B4 96
38C0: D6 DF ED B9 9D 9D DB A7

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

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ

                    --^--
```

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--

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: $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--

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

 A 008 HELLO
 B 042 JANE
 B 007 ZEP
 B 002 MIKE
 B 002 STEVE
 B 004 MARTY
 B 002 GROUP
 B 002 R1
 B 002 R2
 B 002 R3
 B 002 R4
 B 002 R5
 B 002 R6
 B 002 R7
 B 002 R8
 B 002 R9
 B 002 R10
 B 002 R12
 B 002 R13
 A 015 SIGNON
 A 017 RECORDS
 A 021 MULTIPLE
 A 032 STUDENT
 A 031 LESSON1
*T 006 ITEMS.1
 A 030 LESSON2
*T 041 ITEMS.2
 A 043 LESSON3
*T 042 ITEMS.3

]RUN HELLO
...works...
```

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 ][+ 8.4]
  --> 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...
```

Disks 2 and 3 use identical protection.

Quod erat liberandum.

Epilogue

Each disk has an management mode for managing student records. The default administrative password is COVENANT. No ark jokes, please.

EOF