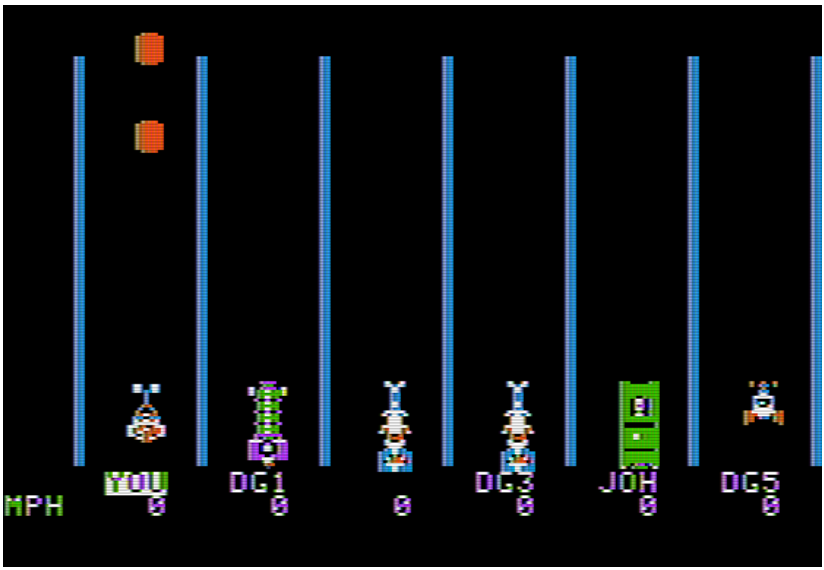
the Quarter Mile



2015-01-08

The Quarter Mile

A 4am crack 2015-01-08

The Quarter Mile is a 1987 educational game by Daniel Barnum and distributed by Barnum Software. It is preserved here for the first time.

The main program is self-contained on a single floppy, but it also supports "accessory disks" that offer additional games on different topics. Once you boot the main program disk, you can swap in an accessory disk at the main menu.

I have three such accessory disks:

* "Fractions I", dated 1987-09-23
* "Integers I" dated 1988-04-11
* "Equations I", dated 1988-09-28

The in-game news bulletin makes reference to other accessory disks that cover decimals, whole numbers, and percents. If anyone comes across those, I hope this write-up is useful in preserving them.

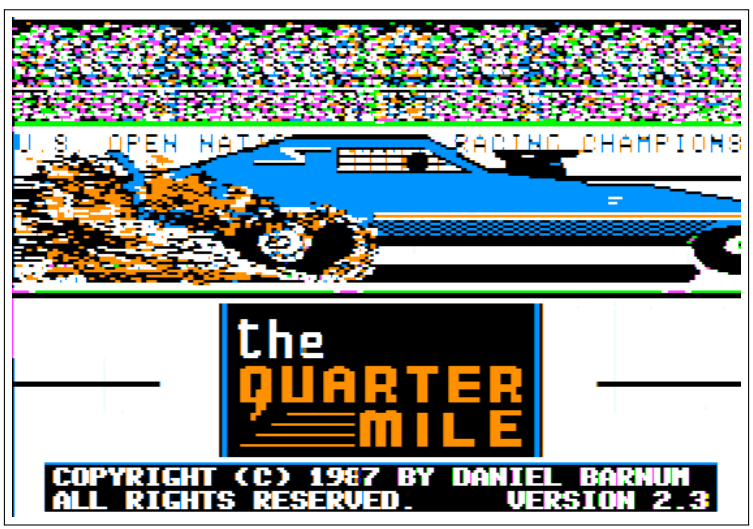
> "It's so overt, it's covert."
>
> Sherlock Holmes: Game of Shadows

COPYA fails on the program disk about halfway through. Locksmith Fast Disk Backup shows why COPYA failed: it can read everything except track $11. EDD 4 bit copy gives no read errors, but the copy does not work. It boots into DOS, displays a prompt, then fills the screen with null bytes and crashes.

Each accessory disk gives the same results as the program disk: COPYA fails halfway through, and Locksmith Fast Disk Backup skips track $11.

Turning to my trusty Disk Fixer sector editor, I can't find any evidence of a standard DOS 3.3 disk catalog anywhere. Maybe it's on the unreadable track $11? The original disk sounds like a DOS 3.3 disk during boot, and the first few tracks look like DOS 3.3 (to a first approximation).

Time for boot tracing with AUTOTRACE.



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

]PR#5
...
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0
CAPTURING BOOT1
...reboots slot 6...
...reboots slot 5...
SAVING BOOT1
SAVING RWTS
```

AUTOTRACE captures everything up to and including the RWTS. But poking through the RWTS, I don't see any evidence of anything that could read track $11. The boot1 code is slightly unusual, but I don't see any nibble checks that would explain why my copy behaves differently than the original disk. Boot0 jumps to boot1 via ($08FD), and boot1 jumps to the usual $9D84 entry point to warm-start DOS.

Time to trace further.

```
]CALL -151

*9600<C600.C6FFM

; set up callback #1 after boot0
96F8-   A9 4C       LDA   #$4C
96FA-   8D 4A 08    STA   $084A
96FD-   A9 0A       LDA   #$0A
96FF-   8D 4B 08    STA   $084B
9702-   A9 97       LDA   #$97
9704-   8D 4C 08    STA   $084C

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

; callback #1 is here
; set up callback #2 after boot1
970A-   A9 4C       LDA   #$4C
970C-   8D 47 B7    STA   $B747
970F-   A9 1C       LDA   #$1C
9711-   8D 48 B7    STA   $B748
9714-   A9 97       LDA   #$97
9716-   8D 49 B7    STA   $B749

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

; callback #2 is here
; capture entire DOS and reboot to my
; work disk
971C-   A2 23       LDX   #$23
971E-   A0 00       LDY   #$00
9720-   B9 00 9D    LDA   $9D00,Y
9723-   99 00 2D    STA   $2D00,Y
9726-   C8          INY
9727-   D0 F7       BNE   $9720
9729-   EE 22 97    INC   $9722
972C-   EE 25 97    INC   $9725
972F-   CA          DEX
9730-   D0 EE       BNE   $9720
9732-   4C 00 C5    JMP   $C500
```

```
*BSAVE TRACE,A$9600,L$135

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

]BSAVE BOOT2,A$2D00,L$2300

Let's see where it all goes awry.

]CALL -151

*FE89G FE93G      ; disconnect DOS
*9D00<2D00.4FFFM ; move DOS into place

*9D84L
.
. nothing unusual, until...
.

; set reset vector (normal)
9E30-   AD 53 9E    LDA   $9E53
9E33-   8D F3 03    STA   $03F3
9E36-   49 A5       EOR   #$A5
9E38-   8D F4 03    STA   $03F4
9E3B-   AD 52 9E    LDA   $9E52
9E3E-   8D F2 03    STA   $03F2

; wait, what?
9E41-   4C 63 A2    JMP   $A263

; Didn't even bother to fix up the code
; trampled by the custom jump. I don't
; think we're ever coming back here.
9E44-   05 AD       ORA   $AD
9E46-   62          ???
9E47-   AA          TAX
9E48-   F0 06       BEQ   $9E50
9E4A-   8D 5F AA    STA   $AA5F
9E4D-   4C 80 A1    JMP   $A180
```

In case you missed it, that JMP $A263 is completely non-standard. A normal DOS 3.3 disk has some non-jumpy code there, then eventually jumps to $A180 (at $9E4D). This disk doesn't get that far.

Let's see what horror lurks at $A263.

```
*A263L

A263-   4C 91 A2    JMP   $A291

*A291L

A291-   20 F5 A4    JSR   $A4F5
A294-   20 12 A4    JSR   $A412
A297-   90 03       BCC   $A29C
A299-   20 66 A2    JSR   $A266
```

Right off the bat, the BCC instruction at $A297 catches my eye. It's only skipping over a single instruction, but the skipped instruction is a JSR $A266, which could be anything.

```
*A266G
...screen fills with null bytes and
the machine crashes...
```

I think I found The Badlands.

Rebooting my work disk and retracing my steps back to this point, let's look at the rest of the code, starting with the first JSR to $A4F5.

```
*A4F5L

A4F5-   EA          NOP

; set the Applesoft RUN flag so that
; any command from a BASIC prompt will
; RUN instead of doing what you wanted
A4F6-   A9 80       LDA   #$80
A4F8-   85 D6       STA   $D6

; set the reset vector (partial)
A4FA-   A9 66       LDA   #$66
A4FC-   8D F2 03    STA   $03F2
A4FF-   A9 A2       LDA   #$A2
A501-   8D F3 03    STA   $03F3
A504-   60          RTS
```

The next call is to $A412, followed by the BCC to skip over The Badlands.

```
*A412L

A412-   EA          NOP
A413-   A9 60       LDA   #$60
A415-   8D EC A4    STA   $A4EC
A418-   A9 05       LDA   #$05
A41A-   8D ED A4    STA   $A4ED
A41D-   AE EC A4    LDX   $A4EC

; turn on the drive motor manually
; (always suspicious)
A420-   BD 8E C0    LDA   $C08E,X
A423-   BD 89 C0    LDA   $C089,X
```

```
; initialize the death counter
A426-   A9 00       LDA   #$00
A428-   8D EE A4    STA   $A4EE
A42B-   A0 00       LDY   #$00
A42D-   C8          INY
A42E-   D0 FD       BNE   $A42D
A430-   EE EE A4    INC   $A4EE
A433-   D0 F6       BNE   $A42B
A435-   A9 00       LDA   #$00
A437-   8C EE A4    STY   $A4EE

; get a nibble
A43A-   20 E0 A4    JSR   $A4E0
A43D-   C8          INY
A43E-   D0 08       BNE   $A448

; if the death counter wraps around to
; zero, give up
A440-   EE EE A4    INC   $A4EE
A443-   D0 03       BNE   $A448
A445-   4C DB A4    JMP   $A4DB

; look for custom prologue (D5 AA BB)
A448-   C9 D5       CMP   #$D5
A44A-   D0 EE       BNE   $A43A
A44C-   20 E0 A4    JSR   $A4E0
A44F-   C9 AA       CMP   #$AA
A451-   D0 F5       BNE   $A448
A453-   20 E0 A4    JSR   $A4E0
A456-   C9 BB       CMP   #$BB
A458-   D0 EE       BNE   $A448
```

```
; get address field (4-4 encoded)
A45A-   A0 00       LDY   #$00
A45C-   20 E0 A4    JSR   $A4E0
A45F-   38          SEC
A460-   2A          ROL
A461-   8D EE A4    STA   $A4EE
A464-   20 E0 A4    JSR   $A4E0
A467-   2D EE A4    AND   $A4EE
A46A-   99 EF A4    STA   $A4EF,Y
A46D-   C8          INY
A46E-   C0 02       CPY   #$02
A470-   D0 EA       BNE   $A45C
A472-   A0 00       LDY   #$00
A474-   20 E0 A4    JSR   $A4E0
A477-   C8          INY
A478-   C0 04       CPY   #$04
A47A-   D0 F8       BNE   $A474

; skip sync byte
A47C-   BD 8C C0    LDA   $C08C,X
A47F-   10 FB       BPL   $A47C
A481-   C9 FF       CMP   #$FF
A483-   D0 4E       BNE   $A4D3

; kill some time to get out of sync
; with the "proper" start of nibbles)
A485-   BD 8D C0    LDA   $C08D,X
A488-   A0 10       LDY   #$10
A48A-   A5 09       LDA   $09
```

```
; now start looking for nibbles that
; don't really exist (except they do,
; because we're out of sync and reading
; timing bits as data)
A48C-   BD 8C C0    LDA   $C08C,X
A48F-   10 FB       BPL   $A48C
A491-   88          DEY
A492-   F0 3F       BEQ   $A4D3
A494-   C9 EE       CMP   #$EE
A496-   D0 F4       BNE   $A48C
A498-   A0 00       LDY   #$00

; store out-of-sync nibbles
A49A-   BD 8C C0    LDA   $C08C,X
A49D-   10 FB       BPL   $A49A
A49F-   99 F1 A4    STA   $A4F1,Y
A4A2-   C8          INY
A4A3-   C0 04       CPY   #$04
A4A5-   D0 F3       BNE   $A49A
A4A7-   AD EF A4    LDA   $A4EF
A4AA-   CD E6 A4    CMP   $A4E6
A4AD-   D0 24       BNE   $A4D3
A4AF-   AD F0 A4    LDA   $A4F0
A4B2-   CD E7 A4    CMP   $A4E7
A4B5-   D0 1C       BNE   $A4D3

; check whether the out-of-sync nibbles
; are correct
A4B7-   A0 00       LDY   #$00
A4B9-   B9 F1 A4    LDA   $A4F1,Y
A4BC-   49 87       EOR   #$87
A4BE-   38          SEC
A4BF-   E9 01       SBC   #$01
A4C1-   D9 E8 A4    CMP   $A4E8,Y
A4C4-   D0 0D       BNE   $A4D3
A4C6-   99 F1 A4    STA   $A4F1,Y
A4C9-   C8          INY
A4CA-   C0 04       CPY   #$04
A4CC-   D0 EB       BNE   $A4B9
```

```
; success path falls through to here --
; turns off drive motor, clears carry,
; and exits
A4CE-   BD 88 C0    LDA   $C088,X
A4D1-   18          CLC
A4D2-   60          RTS

; failure path is here -- decrement a
; counter and eventually give up
A4D3-   CE ED A4    DEC   $A4ED
A4D6-   F0 03       BEQ   $A4DB
A4D8-   4C 35 A4    JMP   $A435

; turn off drive motor, set carry, and
; exit
A4DB-   BD 88 C0    LDA   $C088,X
A4DE-   38          SEC
A4DF-   60          RTS

; subroutine used above
A4E0-   BD 8C C0    LDA   $C08C,X
A4E3-   10 FB       BPL   $A4E0
A4E5-   60          RTS
```

This is similar to nibble checks I've seen on other disks. On success, it clears the carry bit and exits; on failure, it sets the carry instead. Either way, it returns to the caller.

I tried changing the "SEC" to "CLC" at $A4DE. That did successfully bypass the nibble check, but it takes a looooong time to get there. So I decided to bypass the call to $A412 instead.

```
[S6,D1=non-working copy created with
       Locksmith Fast Disk Backup]

T01,S01,$94 change "20 12 A4"
                 to "EA EA 18"

(That's two "NOP" instructions and a
"CLC", which means the BCC instruction
at $A297 will always branch.)

]PR#6
```

Success! Well, sort of. The game boots and loads a graphical title screen. And some other stuff. But then it crashes.

On the theory that the program might be calling this same nibble check later, I changed the "SEC" to "CLC" again. Nope. There appear to be two independent copy protection routines on this disk.

Why build one when you can have two at twice the price?

Back to boot tracing. This is what the loader at $A291 looks like now:

```
*A291L

A291-   20 F5 A4    JSR   $A4F5

; my patch
A294-   EA          NOP
A295-   EA          NOP
A296-   18          CLC

; always taken
A297-   90 03       BCC   $A29C
```

```
; [skipped]
; A299-   20 66 A2    JSR   $A266

; loads from disk via RWTS calls
A29C-   20 05 A5    JSR   $A505
A29F-   20 E1 A2    JSR   $A2E1
A2A2-   A9 34       LDA   #$34
A2A4-   8D 5F AA    STA   $AA5F

; HGR (literally, this is the exact
; routine called when you type "HGR" at
; a BASIC prompt or call it from
; Applesoft)
A2A7-   20 E2 F3    JSR   $F3E2

; show full graphics screen
A2AA-   2C 52 C0    BIT   $C052

; more disk loads via RWTS
A2AD-   20 6F A3    JSR   $A36F

; this one is interesting...
A2B0-   20 42 A5    JSR   $A542

*A542L

A542-   EA          NOP
A543-   A9 00       LDA   #$00
A545-   85 00       STA   $00
A547-   85 02       STA   $02
A549-   A9 40       LDA   #$40
A54B-   85 01       STA   $01
A54D-   A9 20       LDA   #$20
A54F-   85 03       STA   $03

OK, ($00) points to $4000
and ($02) points to $2000. Now what?
```

```
A551-   A0 00        LDY   #$00

; get a byte from $4000
A553-   B1 00        LDA   ($00),Y

; decrypt it, using the raw data of the
; graphical title screen (at $2000) as
; the decryption key
A555-   51 02        EOR   ($02),Y

; and put it back, in place, at $4000
A557-   91 00        STA   ($00),Y
A559-   C8           INY
A55A-   D0 F7        BNE   $A553
A55C-   E6 03        INC   $03
A55E-   E6 01        INC   $01
A560-   A5 03        LDA   $03

; do this for $1000 bytes
A562-   C9 30        CMP   #$30
A564-   D0 ED        BNE   $A553
A566-   60           RTS
```

So $1000 bytes of code are decrypted on the fly. Lovely. Honestly, I'm not sure whether this is about protecting the code or protecting the title screen. I suppose it accomplishes both.

Continuing the listing of the caller, from $A2B3:

```
A2B3-   A9 00       LDA   #$00
A2B5-   85 00       STA   $00
A2B7-   A9 44       LDA   #$44
A2B9-   85 01       STA   $01
A2BB-   A9 02       LDA   #$02
A2BD-   85 02       STA   $02
A2BF-   A0 00       LDY   #$00
A2C1-   B1 00       LDA   ($00),Y
A2C3-   D0 07       BNE   $A2CC
A2C5-   C6 02       DEC   $02
A2C7-   F0 0F       BEQ   $A2D8
A2C9-   4C D0 A2    JMP   $A2D0
A2CC-   A9 02       LDA   #$02
A2CE-   85 02       STA   $02
A2D0-   E6 00       INC   $00
A2D2-   D0 ED       BNE   $A2C1
A2D4-   E6 01       INC   $01
A2D6-   D0 E9       BNE   $A2C1
A2D8-   E6 00       INC   $00
A2DA-   D0 02       BNE   $A2DE
A2DC-   E6 01       INC   $01
A2DE-   6C 00 00    JMP   ($0000)
```

At first glance, I thought this was another decryption loop, but it's not. It loads bytes (at $A2C1), but it never writes anything. The only effect of the loop is to change the pointer at ($00). Of course, the previous loop loop needs to have decrypted the code properly, or it will end up executing garbage. But this is just obfuscating the entry point within the decrypted code.

On my non-working copy, I used a sector editor to change the JMP ($0000) to JMP $FF59, which breaks into the monitor unconditionally.

```
T01,S01,$DE change "6C 00 00"
                to "4C 59 FF"

]PR#6
...
<beep>

*0.1

0000- D9 44
```

The program's entry point is at $44D9.

For posterity, I'm going to reboot to my work disk and save the contents of memory. The author went to great length to prevent anyone from seeing it. Who knows, it might come in handy.

```
*C500G
...

]BSAVE QM.OBJ 2000-5FFF,A$2000,L$4000
]PR#6
...
<beep>

*2000<6000.9FFFM
*C500G
...

]BSAVE QM.OBJ 6000-9FFF,A$2000,L$4000
]PR#6
...
<beep>
```

```
And before I forget, let's restore the
original code to jump to ($0000)...

T01,S01,$DE change "4C 59 FF"
            back to "6C 00 00"

Now I can start tracing the main
program to figure out why my copy still
doesn't work.

]PR#5
...
]BLOAD QM.OBJ 2000-5FFF,A$2000
]BLOAD QM.OBJ 6000-9FFF,A$6000
]CALL -151

*44D9L

44D9-   AD 00 40    LDA   $4000
44DC-   C9 18       CMP   #$18
44DE-   D0 24       BNE   $4504

*4000

4000- 18

OK, so this branch is not taken. (On
further reflection, I'm betting this is
a kind of "first-run vs. second-run"
thing where it needs to do some
initialization the first time.)

; set Ctrl-Y vector (WTF)
44E0-   20 3F 40    JSR   $403F

; save part of zero page
44E3-   20 32 43    JSR   $4332
```

```
; this is just a memory move
44E6-   A9 49        LDA   #$49
44E8-   85 00        STA   $00
44EA-   A9 88        LDA   #$88
44EC-   85 01        STA   $01
44EE-   A9 00        LDA   #$00
44F0-   85 02        STA   $02
44F2-   A9 09        LDA   #$09
44F4-   85 03        STA   $03
44F6-   A9 00        LDA   #$00
44F8-   85 04        STA   $04
44FA-   A9 04        LDA   #$04
44FC-   85 05        STA   $05
44FE-   20 74 43     JSR   $4374

; continue below (part of the first-run
; vs. second-run logic)
4501-   4C 0D 45     JMP   $450D

; [skipped]
; 4504-   20 55 40     JSR   $4055
; 4507-   20 90 40     JSR   $4090
; 450A-   4C 40 45     JMP   $4540

450D-   EA           NOP
450E-   A9 00        LDA   #$00
4510-   8D A1 03     STA   $03A1

; disk read via RWTS
4513-   20 C9 5C     JSR   $5CC9

4516-   AD F6 1F     LDA   $1FF6
4519-   8D A7 03     STA   $03A7
451C-   AD F7 1F     LDA   $1FF7
451F-   8D A2 03     STA   $03A2
4522-   AD F8 1F     LDA   $1FF8
4525-   8D A1 03     STA   $03A1
4528-   AD F9 1F     LDA   $1FF9
452B-   8D CA 03     STA   $03CA
```

```
; hmm
452E-   AD FF 1F    LDA   $1FFF
4531-   C9 01       CMP   #$01
4533-   F0 03       BEQ   $4538
4535-   20 8C 41    JSR   $418C
```

I can't tell (without formally tracing) what the value of $1FFF is at this point, but I can guess whether I want to take that branch by looking at the routine at $418C:

```
*418CL

418C-   EA          NOP
418D-   A9 00       LDA   #$00
418F-   85 00       STA   $00
4191-   A9 04       LDA   #$04
4193-   85 01       STA   $01
4195-   A0 00       LDY   #$00
4197-   A9 00       LDA   #$00
4199-   91 00       STA   ($00),Y
419B-   E6 00       INC   $00
419D-   D0 02       BNE   $41A1
419F-   E6 01       INC   $01
41A1-   A5 00       LDA   $00
41A3-   C9 8C       CMP   #$8C
41A5-   D0 F0       BNE   $4197
41A7-   A5 01       LDA   $01
41A9-   C9 41       CMP   #$41
41AB-   D0 EA       BNE   $4197
; &c. &c. &c.
```

That's another version of The Badlands; it wipes most of main memory (except itself) and crashes.

But I'm positive I haven't seen any routines up to this point that would qualify as copy protection. I think this is just a self-integrity check to ensure that the program code is intact (and hasn't been tampered with).

Continuing...

```
4538-   20 F2 47    JSR   $47F2
453B-   F0 03       BEQ   $4540
453D-   20 A1 4C    JSR   $4CA1
```

Same pattern as before... I wonder what we're skipping over this time.

```
*4CA1L

; Holy paranoia, Batman! It's a *third*
; version of The Badlands, wiping most
; of main memory and crashing.
4CA1-   EA          NOP
4CA2-   A9 00       LDA   #$00
4CA4-   85 00       STA   $00
4CA6-   A9 04       LDA   #$04
4CA8-   85 01       STA   $01
4CAA-   A0 00       LDY   #$00
4CAC-   A9 00       LDA   #$00
4CAE-   91 00       STA   ($00),Y
4CB0-   E6 00       INC   $00
4CB2-   D0 02       BNE   $4CB6
4CB4-   E6 01       INC   $01
4CB6-   A5 00       LDA   $00
[...]
```

```
4CB8-   C9 A1        CMP   #$A1
4CBA-   D0 F0        BNE   $4CAC
4CBC-   A5 01        LDA   $01
4CBE-   C9 4C        CMP   #$4C
4CC0-   D0 EA        BNE   $4CAC
4CC2-   A9 C2        LDA   #$C2
4CC4-   85 00        STA   $00
4CC6-   A9 4C        LDA   #$4C
4CC8-   85 01        STA   $01
4CCA-   4C AC 4C     JMP   $4CAC
; &c. &c. &c.
```

OK, so what's at $47F2? Another self-integrity check? Two in a row? At this point, I'd believe anything. This guy *really* doesn't want anyone tampering with his code.

```
*47F2L

47F2-   A9 11        LDA   #$11
47F4-   8D 33 02     STA   $0233
47F7-   20 06 54     JSR   $5406
47FA-   AD 00 10     LDA   $1000
47FD-   C9 96        CMP   #$96
47FF-   60           RTS
```

I don't know what's going on here, but most of it is happening in $5406.

```
*5406L

5406-   EA           NOP
5407-   20 52 54     JSR   $5452
```

```
*5452L

; Seek (via RWTS call) to the track
; specified in $0233, which was set to
; $11 in the caller. Hey, that's the
; unreadable track!
5452-   20 E3 03    JSR   $03E3
5455-   84 00       STY   $00
5457-   85 01       STA   $01
5459-   AD 33 02    LDA   $0233  ; $11
545C-   A0 04       LDY   #$04
545E-   91 00       STA   ($00),Y
5460-   A9 00       LDA   #$00   ; seek
5462-   A0 0C       LDY   #$0C   ; cmd
5464-   91 00       STA   ($00),Y
5466-   A9 00       LDA   #$00
5468-   A0 03       LDY   #$03
546A-   91 00       STA   ($00),Y
546C-   20 E3 03    JSR   $03E3
546F-   20 D9 03    JSR   $03D9  ; go
5472-   A9 00       LDA   #$00
5474-   85 48       STA   $48
5476-   60          RTS

Popping the stack to $540A...

; get slot number (x16) from RWTS table
; (previous subroutine left pointer to
; the RWTS parameter table in $00/$01)
540A-   A0 01       LDY   #$01
540C-   B1 00       LDA   ($00),Y
540E-   AA          TAX

; turn on drive motor manually (never
; not suspicious)
540F-   BD 89 C0    LDA   $C089,X
5412-   BD 8E C0    LDA   $C08E,X

; sets ($00) to point to $1000
5415-   20 BE 52    JSR   $52BE
```

```
; skip over non-sync bytes
5418-   A0 00        LDY   #$00
541A-   EA           NOP
541B-   BD 8C C0     LDA   $C08C,X
541E-   10 FB        BPL   $541B
5420-   C9 FF        CMP   #$FF
5422-   D0 F7        BNE   $541B

; look for a sequence of sync bytes
; (at least $0F in a row)
5424-   A0 0F        LDY   #$0F
5426-   BD 8C C0     LDA   $C08C,X
5429-   10 FB        BPL   $5426
542B-   C9 FF        CMP   #$FF
542D-   D0 EB        BNE   $541A
542F-   88           DEY
5430-   D0 F4        BNE   $5426

; skip over any extra sync bytes
5432-   BD 8C C0     LDA   $C08C,X
5435-   10 FB        BPL   $5432
5437-   C9 FF        CMP   #$FF
5439-   F0 F7        BEQ   $5432

; now store the raw nibbles at ($00),
; which points to $1000
543B-   BD 8C C0     LDA   $C08C,X
543E-   10 FB        BPL   $543B
5440-   91 00        STA   ($00),Y
5442-   E6 00        INC   $00
5444-   D0 F5        BNE   $543B
5446-   E6 01        INC   $01
5448-   A5 01        LDA   $01

; looks like we're storing 16 pages of
; raw nibbles ($1000..$1FFF)
544A-   C9 20        CMP   #$20
544C-   90 ED        BCC   $543B
```

```
; turn off drive motor and exit
544E-   BD 88 C0    LDA   $C088,X
5451-   60          RTS
```

Let's revisit the caller in its entirety, filling in the pieces with what we've learned so far.

```
*47F2L

; set track number to seek
47F2-   A9 11       LDA   #$11
47F4-   8D 33 02    STA   $0233

; read raw nibbles from track $11 into
; $1000..$1FFF
47F7-   20 06 54    JSR   $5406

; look at the first one
47FA-   AD 00 10    LDA   $1000
47FD-   C9 96       CMP   #$96
47FF-   60          RTS
```

Popping the stack again, to $4538...

```
*4538L

; read unreadable track $11, compare
; the first non-sync byte to $96
4538-   20 F2 47    JSR   $47F2

; if equal, continue
453B-   F0 03       BEQ   $4540

; if not equal, fail catastrophically
453D-   20 A1 4C    JSR   $4CA1
```

And there it is. It reads $1000 nibbles from track $11 and compares the first one.

```
; skip the call to $5406 entirely
*47F7:2C

; patch the comparison check so it's
; always equal and the caller's BEQ
; branch is always taken
*47FA:EA A9 96
```

But how can I save this patch? All of this code is encrypted on disk and decrypted in memory by the loader. Oh hey, I have the decrypted version, because I saved it off after tracing the loader (and letting the decryption loop run). So let's write that back to disk and patch the loader so it doesn't decrypt the already-decrypted code.

The loader uses standard RWTS calls to read code from the disk. It reads in decreasing sector order but in increasing page order. That is, it starts at T03,S0F and address $2000 and works its way down (sectors) and up (addresses). So track $05, sector $0F is $4000..$40FF, sector $0E is $4100..$41FF, &c.

```
08C0-   A9 08        LDA   #$08
08C2-   A0 E8        LDY   #$E8
08C4-   20 D9 03     JSR   $03D9
08C7-   AC ED 08     LDY   $08ED
08CA-   88           DEY
08CB-   10 05        BPL   $08D2
08CD-   A0 0F        LDY   #$0F
08CF-   EE EC 08     INC   $08EC
08D2-   8C ED 08     STY   $08ED
08D5-   EE F1 08     INC   $08F1
08D8-   CE E1 08     DEC   $08E1
08DB-   D0 E3        BNE   $08C0
08DD-   60           RTS
```

```
          +-- sector count
          v
08E0- 00 10 00 00 00 00 00 00

08E8- 01 60 01 00 05 0F FB 08
                   ^    ^
           track --+    +-- sector

08F0- 00 40 00 00 02 00 FE 60
           ^
           +-- starting address

08F8- 01 00 00 00 01 EF D8 00

*BSAVE WRITE DECRYPTED CODE,A$8C0,L$40

[S6,D1=non-working copy]

*8C0G
...write write write...
```

Turning to my trusty Disk Fixer sector editor, I need to go back to the custom loader at $A291 and prevent it from decrypting this code, since it's now already decrypted on disk. That routine was at $A542, which is on T01,S04.

T01,S04,$42 change "EA" to "60"

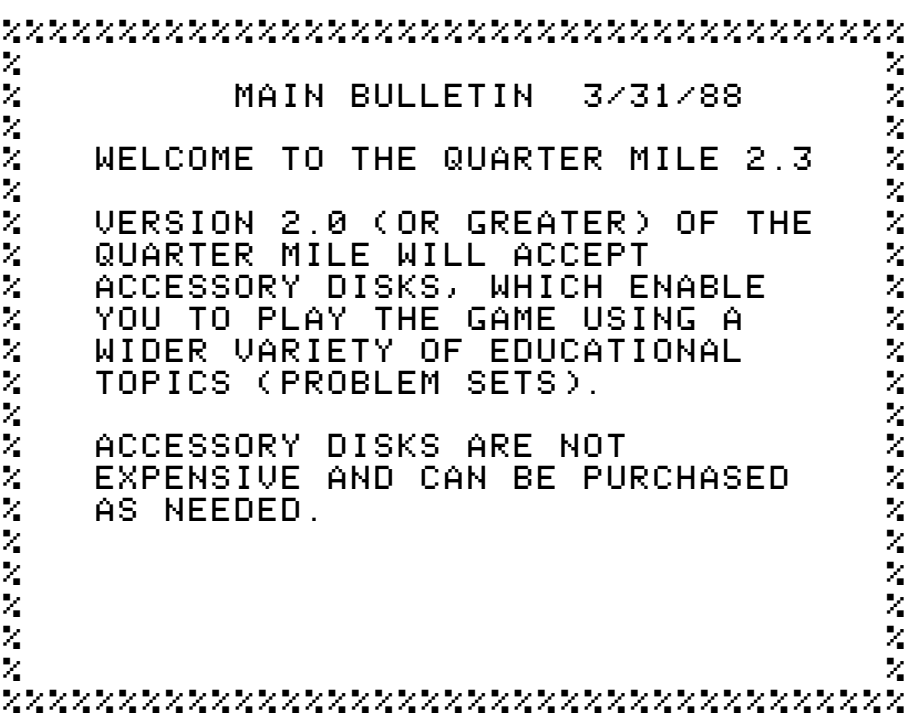
]PR#6

The game loads and runs without any further complaint. All options (including saving games and creating data disks) are fully functional.

Except one...

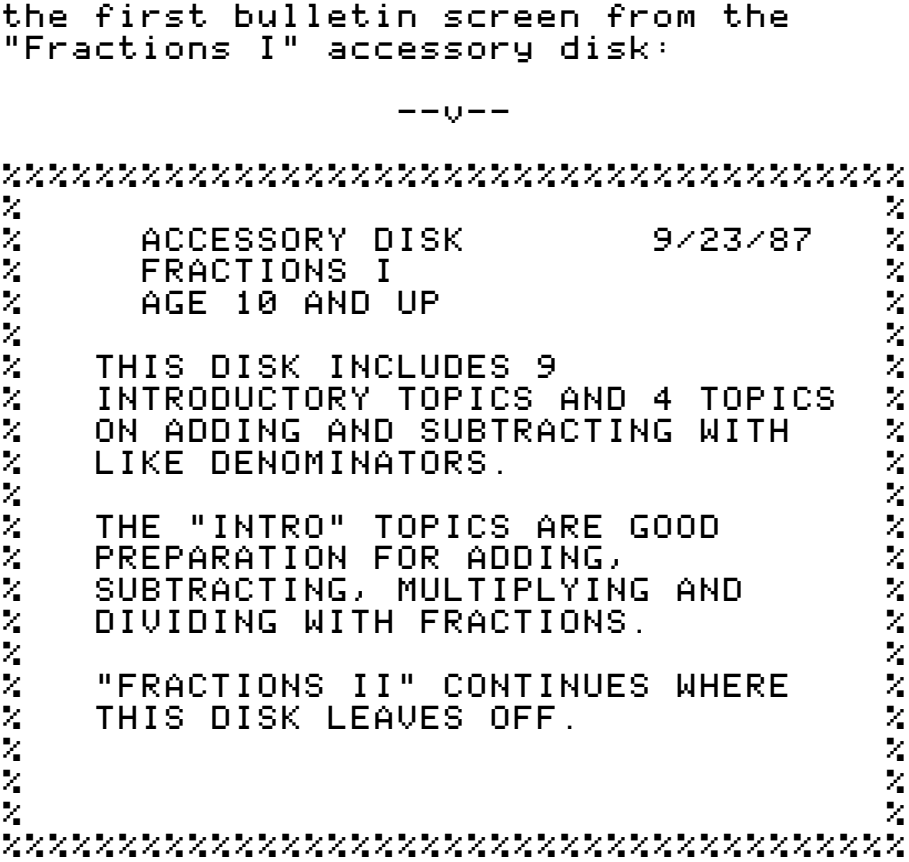
There is a main menu option called "BULLETIN". On the original disk, it displays several messages, one screen at a time, like so:

--v--



--^--

If you swap in an accessory disk and select "BULLETIN", it print a different set of messages. For example, here is the first bulletin screen from the "Fractions I" accessory disk:



But on my copy, I get no such messages (neither from the main program disk nor the accessory disks). When I select "BULLETIN", it reads the disk as if it's loading the bulletin text, but it just spews garbage within the border box of "%" characters.

This is not a critical bug. The game itself is still playable, even with the accessory disks. But it is definitely a bug, and a quite visible one at that. Somehow I have either triggered it or caused it or missed something somehow somewhere.

This is clearly unacceptable.

After minutes of intensive research, I can summarize the situation as follows:

* If I boot the original disk, leave it in, and select "BULLETIN", it reads the disk and displays the messages correctly.

* If I boot my copy, leave it in, and select "BULLETIN", it reads the disk and displays garbage.

* If I boot the original disk to the main menu, swap in my copy, and select "BULLETIN", it reads my copy and displays garbage.

* If I boot my copy to the main menu, swap in the original disk, and select "BULLETIN", it reads the (original) disk and displays the messages correctly.

I should turn that into a 2 x 2 chart, but I'm tired of making ASCII art. But that last finding is very significant. It tells me that I don't have a code problem; I have a data problem. The code on my copy is fine. All of the decryption and writing back decrypted code to the disk and disabling multiple routines during boot and skipping the nibble reading and hard-coding that comparison check... none of that prevents my copy from reading the bulletin data from the original disk and displaying it correctly.

My copy just doesn't have the data.

Because it's on track $11.

It must be. That's the only remaining difference between my copy and the original disk. Locksmith Fast Disk Backup couldn't read track $11, so it just wrote 16 sectors worth of null bytes and moved on. Now any disk (even the original) that tries to display a bulletin from my copy reads something from the disk and spews garbage.

I already know there's a routine to read track $11 -- it's at $5406. But I disabled that call, thinking it was purely copy protection. Could that why I'm seeing garbage in the bulletin? No, that doesn't make sense. When I boot my copy and swap in the original disk at the main menu, I see the correct text in the bulletin. That means it must be loading the bulletin text when I select "BULLETIN" from the main menu (at which point it's reading it from the original disk).

Since I wrote the decrypted code back to my copy, I can easily scan my copy for the hex sequence "20 06 54". Lo and behold, Disk Fixer finds a match on T06,S0D, at byte offset $AC. That's loaded into memory at $52AC.

```
]PR#5
...
]BLOAD QM.OBJ 2000-5FFF,A$2000
]BLOAD QM.OBJ 6000-9FFF,A$6000
]CALL -151
```

And there it is, right where I thought: another call to the nibble-reading routine at $5406. It's part of a larger subroutine that starts at $52A1 (based on the fact that $52A0 is an RTS).

```
*52A0L

52A0-   60          RTS
```

```
; again, setting up to seek to the
; unreadable track $11
52A1-   A9 11       LDA   #$11
52A3-   8D 33 02    STA   $0233

; these two subroutines just print a
; border around the text screen and
; some instructions at the bottom
52A6-   20 C7 52    JSR   $52C7
52A9-   20 E6 52    JSR   $52E6

; this reads raw nibbles from track $11
; into $1000..$1FFF
52AC-   20 06 54    JSR   $5406

; this is where things get interesting
52AF-   20 77 54    JSR   $5477

*5477L

5477-   EA          NOP

; reset ($00) pointer to $1000
5478-   20 BE 52    JSR   $52BE
547B-   A0 00       LDY   #$00

; get a raw nibble
547D-   B1 00       LDA   ($00),Y

; decrypt raw nibble into readable text
; (really)
547F-   20 90 54    JSR   $5490

; store it back
5482-   91 00       STA   ($00),Y
```

```
; and loop through all the nibbles
; (16 pages worth, $1000..$1FFF)
5484-   C8           INY
5485-   D0 F6        BNE   $547D
5487-   E6 01        INC   $01
5489-   A5 01        LDA   $01
548B-   C9 20        CMP   #$20
548D-   D0 EE        BNE   $547D
548F-   60           RTS

; This is the nibble-to-character
; conversion routine. It finds the
; index of the nibble in the standard
; nibble table at $BA29...
5490-   A2 3F        LDX   #$3F
5492-   DD 29 BA     CMP   $BA29,X
5495-   F0 08        BEQ   $549F
5497-   CA           DEX
5498-   10 F8        BPL   $5492
549A-   A9 BF        LDA   #$BF
549C-   4C A3 54     JMP   $54A3

; ...then converts the index to a
; printable character by offsetting it
; by $A0 (the space character).
549F-   8A           TXA
54A0-   18           CLC
54A1-   69 A0        ADC   #$A0
54A3-   60           RTS
```

It's so overt, it's covert. The unreadable track $11 isn't (just) for copy protection after all. It has actual data on it: a completely non-sector-based stream of nibbles that are converted to printable text and then printed.

Track $11 *is* the bulletin.

Now it all makes sense. Obviously my Locksmith Fast Disk Backup copies didn't preserve this data. Locksmith couldn't read the track at all, so it just skipped it. I originally thought nothing of it. Lots of disks dedicate an entire track to their copy protection scheme; once I bypass the protection, the empty track is just ignored. But this track isn't empty at all.

Popping the stack and continuing the listing at $52B2...

```
; reset the ($00) pointer again
52B2-   20 BE 52    JSR   $52BE

; this subroutine prints one message at
; a time and waits for a keypress
52B5-   20 46 53    JSR   $5346

; now branch back if there is another
; message to print and the user hasn't
; pressed ESC to cancel
52B8-   D0 FB       BNE   $52B5

; reset text borders and clear screen
52BA-   20 F3 53    JSR   $53F3
52BD-   60          RTS
```

There's nothing in this routine that clears the messages from memory after it's done. In fact, this entire routine is self-contained. That gives me a crazy idea.

```
]S6,D1=my work disk]

*C600G
...
]BLOAD QM.OBJ 2000-5FFF,A$2000
]BLOAD QM.OBJ 6000-9FFF,A$6000
]CALL -151

]S6,D1=original program disk]

*54A1G
...loads bulletin and displays it...
...ESC takes me back to the monitor...

*FC58G N 400<1000.13FFM
...screen fills with bulletin text...

]S6,D1=my work disk]

*BSAVE BULLETIN DECRYPTED,A$1000,L$1000
```

And just like that, I have the bulletin text from the unreadable track $11, converted to printable text and saved as a file.

(I repeated the procedure with each accessory disk. Remember those? They each have their own bulletin, which is stored in the same way on track $11. Which is brilliant, by the way. A blog distributed on floppy disks.)

I find myself faced with a wonderful confluence of coincidences:

* Each bulletin is exactly 16 pages worth of data ($1000..$1FFF)

* There are exactly 16 sectors free on each disk (track $11)

* There is already a subroutine (at $5406) dedicated to reading the bulletin text

The obvious solution is to write out the bulletin text to the main program disk and each accessory disk as standard sectors on track $11. Then I can modify the subroutine at $5406 to read those sectors into $1000..$1FFF via standard RWTS calls.

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

```
          +-- sector count
          v
08E0- 00 10 00 00 00 00 00 00
08E8- 01 60 01 00 11 0F FB 08
                  ^    ^
           track --+    +-- sector

08F0- 00 1F 00 00 02 00 FE 60
           ^
           +-- starting address

*BSAVE WRITE BULLETIN,A$8C0,L$40

*BLOAD BULLETIN DECRYPTED,A$1000

]S6,D1=program disk copy]

*8C0G
```

I repeated this procedure for each accessory disk. The files are stored on my work disk as

* FRACTIONS BULLETIN DECRYPTED
* INTEGERS BULLETIN DECRYPTED
* EQUATIONS BULLETIN DECRYPTED

Now that the decrypted text is stored in standard sectors, I need to rewrite the routine at $5406 (on T06,S0B) so it uses standard RWTS calls to read track $11 into $1000..$1FFF. I wrote this directly in a sector editor, so here is its code listing with my comments inline:

```
---------- DISASSEMBLY MODE ----------
0006:EA              NOP
```

```
; seek to track $11 (unchanged)
0007:20 52 54        JSR   $5452

; starting sector ($0F)
000A:A0 05           LDY   #$05
000C:A9 0F           LDA   #$0F
000E:91 00           STA   ($00),Y

; starting address ($1F00)
0010:A0 08           LDY   #$08
0012:A9 00           LDA   #$00
0014:91 00           STA   ($00),Y
0016:C8              INY
0017:A9 1F           LDA   #$1F
0019:91 00           STA   ($00),Y

; read command ($01)
001B:A0 0C           LDY   #$0C
001D:A9 01           LDA   #$01
001F:91 00           STA   ($00),Y

; any disk volume
0021:A0 03           LDY   #$03
0023:A9 00           LDA   #$00
0025:91 00           STA   ($00),Y

; read sector via RWTS
0027:20 E3 03        JSR   $03E3
002A:20 D9 03        JSR   $03D9

; decrement address
002D:A0 09           LDY   #$09
002F:B1 00           LDA   ($00),Y
0031:38              SEC
0032:E9 01           SBC   #$01
0034:91 00           STA   ($00),Y
```

```
; decrement sector
0036:A0 05           LDY   #$05
0038:B1 00           LDA   ($00),Y
003A:38              SEC
003B:E9 01           SBC   #$01
003D:91 00           STA   ($00),Y

; loop until we've read all 16 sectors
003F:10 E6           BPL   $0027
0041:A9 00           LDA   #$00
0043:85 48           STA   $48
0045:60              RTS

; now unused space
0046:00              BRK
0047:00              BRK
0048:00              BRK
0049:00              BRK
004A:00              BRK
004B:00              BRK
004C:00              BRK
004D:00              BRK
004E:00              BRK
004F:00              BRK
0050:00              BRK
0051:00              BRK
```

Last but not least, I need to disable the nibble-to-text conversion routine at $5477, which is no longer needed (since the bulletin text is already stored as printable characters).

```
T06,S0B,$77 change "EA" to "60"
```

Quod erat liberandum.

A 4am crack No. 182

----------EOF----------