15:06 Gumball

by 4am and Peter Ferrie (qkumba, san inc)

Name Gumball

Genre arcade

Year 1983

Credits by Robert Cook, concept by Doug Carlston

Publisher Broderbund Software

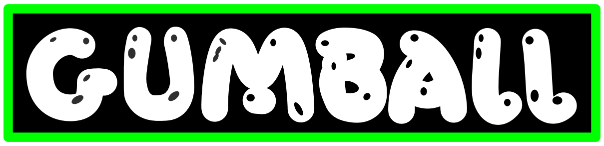
Platform Apple][+ or later (48K)

Media single-sided 5.25-inch floppy

OS custom

Other versions

- Mr. Krac-Man & The Disk Jockey
- several uncredited cracks



In Which Various Automated Tools Fail In Interesting Ways

COPYA immediate disk read error

Locksmith Fast Disk Backup unable to read any track

EDD 4 bit copy (no sync, no count) Disk seeks off track 0, then hangs with the drive motor on

Copy II+ nibble editor

- T00 has a modified address prologue (D5 AA B5) and modified epilogues
- T01+ appears to be 4-4 encoded data (2 nibbles on disk = 1 byte in memory) with a custom prologue/ delimiter. In any case, it's neither 13 nor 16 sectors.

Disk Fixer not much help

Why didn't COPYA work? not a 16-sector disk

Why didn't Locksmith FDB work? ditto

Why didn't my EDD copy work? I don't know. Early Broderbund games loved using half tracks and quarter tracks, not to mention the runtime protection checks, so it could be literally anything. Or, more likely, any combination of things.

This is decidedly not a single-load game. There is a classic crack that is a single binary, but it cuts out a lot of the introduction and some cut scenes later. All other cracks are whole-disk, multi-loaders.

Combined with the early indications of a custom bootloader and 4-4 encoded sectors, this is not going to be a straightforward crack by any definition of "straight" or "forward."

Let's start at the beginning.

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.

```
*9600<C600.C6FFM             copy drive firmware to $9600

*9600G                       and execute it
```

. . .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.

```
96F8      A0 00   LDY #$00        instead of jumping to on-disk
96FA   B9 00 08   LDA $0800,Y     code, copy boot sector to
96FD   99 00 28   STA $2800,Y     higher memory so it survives
9700         C8   INY             a reboot
9701      D0 F7   BNE $96FA

9703   AD E8 C0   LDA $C0E8       turn off slot 6 drive motor

9706   4C 00 C5   JMP $C500       reboot to my work disk in slot
*9600G                            5
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT0,A$2800,L$100
```

Now we get to[21] trace the boot process one sector, one page, one instruction at a time.

In Which We Get To Dip Our Toes Into An Ocean Of Raw Sewage

```
]CALL -151

*800<2800.28FFM                   copy code back to $0800
801L                              where it was originally loaded,
                                  to make it easier to follow

0801      A2 00   LDX #$00        immediately move this code
0803   BD 00 08   LDA $0800,X     to the input buffer at $0200
0806   9D 00 02   STA $0200,X
0809         E8   INX
080A      D0 F7   BNE $0803
080C   4C 0F 02   JMP $020F
```

OK, I can do that too. Well, mostly. The page at $0200 is the text input buffer, used by both Applesoft BASIC and the built-in monitor (which I'm in right now). But I can copy enough of it to examine this code in situ.

```
*20F<80F.8FFM
*20FL
```

```
020F      A0 AB   LDY #$AB        set up a nibble translation
0211         98   TYA             table at $0800
0212      85 3C   STA $3C
0214         4A   LSR
0215      05 3C   ORA $3C
0217      C9 FF   CMP #$FF
0219      D0 09   BNE $0224
021B      C0 D5   CPY #$D5
021D      F0 05   BEQ $0224
021F         8A   TXA
0220   99 00 08   STA $0800,Y
0223         E8   INX
0224         C8   INY
0225      D0 EA   BNE $0211
0227      84 3D   STY $3D

0229      84 26   STY $26         #$00 into zero page $26 and
022B      A9 03   LDA #$03        #$03 into $27 means we're
022D      85 27   STA $27         probably going to be loading
                                  data into $0300..$03FF later,
                                  because ($26) points to $0300.

022F      A6 2B   LDX $2B         zero page $2B holds the boot
0231   20 5D 02   JSR $025D       slot x16

*25DL

025D         18   CLC             read a sector from track $00
025E         08   PHP             (this is actually derived from
025F   BD 8C C0   LDA $C08C,X     the code in the disk controller
0262      10 FB   BPL $025F       ROM routine at $C65C, but
0264      49 D5   EOR #$D5        looking for an address
0266      D0 F7   BNE $025F       prologue of "D5 AA B5" instead
0268   BD 8C C0   LDA $C08C,X     of "D5 AA 96") and using the
026B      10 FB   BPL $0268       nibble translation table we set
026D      C9 AA   CMP #$AA        up earlier at $0800
026F      D0 F3   BNE $0264
0271         EA   NOP
0272   BD 8C C0   LDA $C08C,X
0275      10 FB   BPL $0272

0277      C9 B5   CMP #$B5        #$B5 for third prologue
0279      F0 09   BEQ $0284       nibble
027B         28   PLP
027C      90 DF   BCC $025D
027E      49 AD   EOR #$AD
0280      F0 1F   BEQ $02A1
0282      D0 D9   BNE $025D
0284      A0 03   LDY #$03
0286      84 2A   STY $2A
0288   BD 8C C0   LDA $C08C,X
028B      10 FB   BPL $0288
028D         2A   ROL
028E      85 3C   STA $3C
0290   BD 8C C0   LDA $C08C,X
0293      10 FB   BPL $0290
0295      25 3C   AND $3C
0297         88   DEY
0298      D0 EE   BNE $0288
029A         28   PLP
029B      C5 3D   CMP $3D
029D      D0 BE   BNE $025D
029F      B0 BD   BCS $025E
02A1      A0 9A   LDY #$9A
02A3      84 3C   STY $3C
02A5   BC 8C C0   LDY $C08C,X
02A8      10 FB   BPL $02A5
```

---

[21] If you replace the words "need to" with the words "get to," life becomes amazing.

```
02AA  59 00 08  EOR $0800,Y  use the nibble translation
02AD     A4 3C  LDY $3C      table we set up earlier to
02AF        88  DEY          convert nibbles on disk into
02B0  99 00 08  STA $0800,Y  bytes in memory
02B3     D0 EE  BNE $02A3
02B5     84 3C  STY $3C
02B7  BC 8C C0  LDY $C08C,X
02BA     10 FB  BPL $02B7
02BC  59 00 08  EOR $0800,Y
02BF     A4 3C  LDY $3C

02C1     91 26  STA ($26),Y  store the converted bytes at
02C3        C8  INY          $0300
02C4     D0 EF  BNE $02B5

02C6  BC 8C C0  LDY $C08C,X  verify the data with a
02C9     10 FB  BPL $02C6    one-nibble checksum
02CB  59 00 08  EOR $0800,Y
02CE     D0 8D  BNE $025D
02D0        60  RTS
```

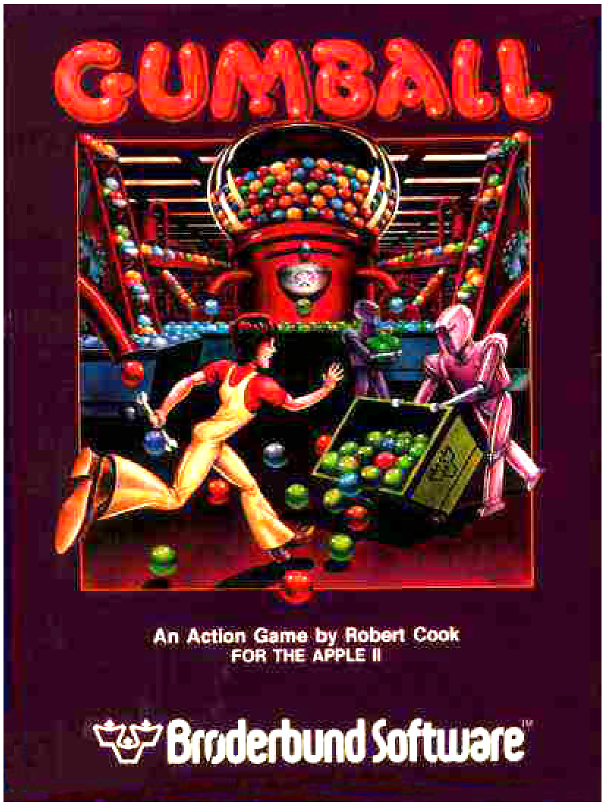
Continuing from $0234...

```
*234L
0234  20 D1 02  JSR $02D1
*2D1L
02D1        A8  TAY          finish decoding nibbles
02D2     A2 00  LDX #$00
02D4  B9 00 08  LDA $0800,Y
02D7        4A  LSR
02D8  3E CC 03  ROL $03CC,X
02DB        4A  LSR
02DC  3E 99 03  ROL $0399,X
02DF     85 3C  STA $3C
02E1     B1 26  LDA ($26),Y
02E3        0A  ASL
02E4        0A  ASL
02E5        0A  ASL
02E6     05 3C  ORA $3C
02E8     91 26  STA ($26),Y
02EA        C8  INY
02EB        E8  INX
02EC     E0 33  CPX #$33
02EE     D0 E4  BNE $02D4
02F0     C6 2A  DEC $2A
02F2     D0 DE  BNE $02D2

02F4  CC 00 03  CPY $0300    verify final checksum
02F7     D0 03  BNE $02FC

02F9        60  RTS          checksum passed, return to
                             caller and continue with the
                             boot process

02FC  4C 2D FF  JMP $FF2D    checksum failed, print "ERR"
                             and exit
```

Continuing from $0237...

```
0237  4C 01 03  JMP $0301    jump into the code we just
                             read
```

This is where I get to interrupt the boot, before it jumps to $0301.



In Which We Do A Bellyflop Into A Decrypted Stack And Discover That I Am Very Bad At Metaphors

```
*9600<C600.C6FFM

96F8     A9 05  LDA #$05     patch boot0 so it calls my
96FA  8D 38 08  STA $0838    routine instead of jumping to
96FD     A9 97  LDA #$97     $0301
96FF  8D 39 08  STA $0839

9702  4C 01 08  JMP $0801    start the boot

9705     A0 00  LDY #$00     (callback is here) copy the
9707  B9 00 03  LDA $0300,Y  code at $0300 to higher
970A  99 00 23  STA $2300,Y  memory so it survives a
970D        C8  INY          reboot
970E     D0 F7  BNE $9707

9710  AD E8 C0  LDA $C0E8    turn off slot 6 drive motor
9713  4C 00 C5  JMP $C500    and reboot to my work disk
*BSAVE TRACE,A$9600,L$116    in slot 5
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0300-03FF,A$2300,L$100
]CALL -151
*2301L
2301     84 48  STY $48
```

```
2303     A0 00    LDY #$00        clear hi-res graphics screen 2
2305        98    TYA
2306     A2 20    LDX #$20
2308  99 00 40    STA $4000,Y
230B        C8    INY
230C     D0 FA    BNE $2308
230E  EE 0A 03    INC $030A
2311        CA    DEX
2312     D0 F4    BNE $2308

2314  AD 57 C0    LDA $C057       and show it (appears blank)
2317  AD 52 C0    LDA $C052
231A  AD 55 C0    LDA $C055
231D  AD 50 C0    LDA $C050

2320  B9 00 03    LDA $0300,Y     decrypt the rest of this page
2323     45 48    EOR $48         to the stack page at $0100
2325  99 00 01    STA $0100,Y
2328        C8    INY
2329     D0 F5    BNE $2320

232B     A2 CF    LDX #$CF        set the stack pointer
232D        9A    TXS

232E        60    RTS             and exit via RTS


*9600<C600.C6FFM
96F8     A9 05    LDA #$05        patch boot0 so it calls my
96FA  8D 38 08    STA $0838       routine instead of jumping to
96FD     A9 97    LDA #$97        $0301
96FF  8D 39 08    STA $0839

9702  4C 01 08    JMP $0801       start the boot

9705     A0 00    LDY #$00        (callback is here) copy the
9707  B9 00 03    LDA $0300,Y     code at $0300 to higher
970A  99 00 23    STA $2300,Y     memory so it survives a
970D        C8    INY             reboot
970E     D0 F7    BNE $9707

9710  AD E8 C0    LDA $C0E8       turn off slot 6 drive motor
9713  4C 00 C5    JMP $C500       and reboot to my work disk
                                  in slot 5
*BSAVE TRACE,A$9600,L$116
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0300-03FF,A$2300,L$100
]CALL -151
*2301L
2301     84 48    STY $48

2303     A0 00    LDY #$00        clear hi-res graphics screen 2
2305        98    TYA
2306     A2 20    LDX #$20
2308  99 00 40    STA $4000,Y
230B        C8    INY
230C     D0 FA    BNE $2308
230E  EE 0A 03    INC $030A
2311        CA    DEX
2312     D0 F4    BNE $2308

2314  AD 57 C0    LDA $C057       and show it (appears blank)
2317  AD 52 C0    LDA $C052
231A  AD 55 C0    LDA $C055
231D  AD 50 C0    LDA $C050
2320  B9 00 03    LDA $0300,Y     decrypt the rest of this page
2323     45 48    EOR $48         to the stack page at $0100
2325  99 00 01    STA $0100,Y
2328        C8    INY
2329     D0 F5    BNE $2320

232B     A2 CF    LDX #$CF        set the stack pointer
232D        9A    TXS

232E        60    RTS             and exit via RTS
```

Oh joy, stack manipulation. The stack on an Apple II is just `$100` bytes in main memory (`$0100..$01FF`) and a single byte register that serves as an index into that page. This allows for all manner of mischief—overwriting the stack page (as we're doing here), manually changing the stack pointer (also doing that here), or even putting executable code directly on the stack.

The upshot is that I have no idea where execution continues next, because I don't know what ends up on the stack page. I get to interrupt the boot again to see the decrypted data that ends up at $0100.

Mischief Managed

```
*BLOAD TRACE
[first part is the same as the
 previous trace]
9705     84 48    STY $48         reproduce the decryption
9707     A0 00    LDY #$00        loop, but store the result at
9709  B9 00 03    LDA $0300,Y     $2100 so it survives a reboot
970C     45 48    EOR $48
970E  99 00 21    STA $2100,Y
9711        C8    INY
9712     D0 F5    BNE $9709

9714  AD E8 C0    LDA $C0E8       turn off drive motor and
9717  4C 00 C5    JMP $C500       reboot to my work disk

*BSAVE TRACE2,A$9600,L$11A
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0100-01FF,A$2100,L$100
]CALL -151
```

The original code at $0300 manually reset the stack pointer to #$CF and exited via RTS. The Apple II will increment the stack pointer before using it as an index into $0100 to get the next address. (For reasons I won't get into here, it also increments the address before passing execution to it.)

```
*21D0.
21D0  2F 01  FF 03 FF 04 4F 04
      next return address
```

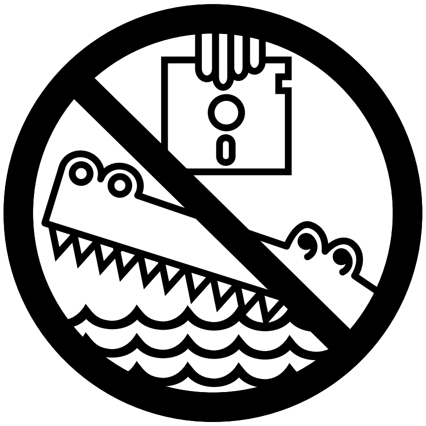
$012F + 1 = $0130, which is already in memory at $2130.

Oh joy. Code on the stack. (Remember, the "stack" is just a page in main memory. If you want to use that page for something else, it's up to you to ensure that it doesn't conflict with the stack functioning as a stack.)

```
*2130L
2130      A2 04    LDX #$04
2132      86 86    STX $86
2134      A0 00    LDY #$00
2136      84 83    STY $83
2138      86 84    STX $84
```

Now ($83) points to $0400.

```
213A      A6 2B    LDX $2B        get slot number (x16)

213C   BD 8C C0    LDA $C08C,X    find a 3-nibble prologue ("BF
213F      10 FB    BPL $213C      D7 D5")
2141      C9 BF    CMP #$BF
2143      D0 F7    BNE $213C
2145   BD 8C C0    LDA $C08C,X
2148      10 FB    BPL $2145
214A      C9 D7    CMP #$D7
214C      D0 F3    BNE $2141
214E   BD 8C C0    LDA $C08C,X
2151      10 FB    BPL $214E
2153      C9 D5    CMP #$D5
2155      D0 F3    BNE $214A

2157   BD 8C C0    LDA $C08C,X    read 4-4-encoded data
215A      10 FB    BPL $2157
215C         2A    ROL
215D      85 85    STA $85
215F   BD 8C C0    LDA $C08C,X
2162      10 FB    BPL $215F
2164      25 85    AND $85

2166      91 83    STA ($83),Y    store in $0400 (text page, but
2168         C8    INY            it's hidden right now because
2169      D0 EC    BNE $2157      we switched to hi-res graphics
                                  screen 2 at $0314)

216B   0E 00 C0    ASL $C000      find a 1-nibble epilogue ("D4")
216E   BD 8C C0    LDA $C08C,X
2171      10 FB    BPL $216E
2173      C9 D4    CMP #$D4
2175      D0 B9    BNE $2130

2177      E6 84    INC $84        increment target memory
                                  page

2179      C6 86    DEC $86        decrement sector count
217B      D0 DA    BNE $2157      (initialized at $0132)

217D         60    RTS            exit via RTS
```



Wait, what? Ah, we're using the same trick we used to call this routine—the stack has been prefilled with a series of "return" addresses. It's time to "return" to the next one.

```
*21D0.
21D0 2F 01  FF 03  FF 04 4F 04
           next return address
```

$03FF + 1 = $0400, and that's where I get to interrupt the boot.

Seek And Ye Shall Find

```
*BLOAD TRACE2
.
.  [same as previous trace]
.
9705      84 48    STY $48        reproduce the decryption loop
9707      A0 00    LDY #$00       that was originally at $0320
9709   B9 00 03    LDA $0300,Y
970C      45 48    EOR $48
970E   99 00 01    STA $0100,Y
9711         C8    INY
9712      D0 F5    BNE $9709

9714      A9 21    LDA #$21       now that the stack is in place
9716   8D D2 01    STA $01D2      at $0100, change the first
9719      A9 97    LDA #$97       return address so it points to
971B   8D D3 01    STA $01D3      a callback under my control
                                  (instead of continuing to
                                  $0400)

971E      A2 CF    LDX #$CF       continue the boot
9720         9A    TXS
9721         60    RTS

9722      A2 04    LDX #$04       (callback is here) copy the
9724      A0 00    LDY #$00       contents of the text page to
9726   B9 00 04    LDA $0400,Y    higher memory
9729   99 00 24    STA $2400,Y
972C         C8    INY
972D      D0 F7    BNE $9726
972F   EE 28 97    INC $9728
9732   EE 2B 97    INC $972B
9735         CA    DEX
9736      D0 EE    BNE $9726
```

```
9738   AD E8 C0   LDA $C0E8     turn off the drive and reboot
973B   4C 00 C5   JMP $C500     to my work disk
```

```
*BSAVE TRACE3,A$9600,L$13E
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0400-07FF,A$2400,L$400
]CALL -151
```

I'm going to leave this code at $2400, since I can't put it on the text page and examine it at the same time. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2400L
2400      A0 00   LDY #$00      copy three pages to the top of
2402   B9 00 05   LDA $0500,Y   main memory
2405   99 00 BD   STA $BD00,Y
2408   B9 00 06   LDA $0600,Y
240B   99 00 BE   STA $BE00,Y
240E   B9 00 07   LDA $0700,Y
2411   99 00 BF   STA $BF00,Y
2414         C8   INY
2415      D0 EB   BNE $2402
```

I can replicate that.

```
*FE89G FE93G ; disconnect DOS
*BD00<2500.27FFM ; simulate
copy loop
2417      A6 2B   LDX $2B
2419   8E 66 BF   STX $BF66
241C   20 48 BF   JSR $BF48

*BF48L
BF48   AD 81 C0   LDA $C081     zap contents of language card
BF4B   AD 81 C0   LDA $C081
BF4E      A0 00   LDY #$00
BF50      A9 D0   LDA #$D0
BF52      84 A0   STY $A0
BF54      85 A1   STA $A1
BF56      B1 A0   LDA ($A0),Y
BF58      91 A0   STA ($A0),Y
BF5A         C8   INY
BF5B      D0 F9   BNE $BF56
BF5D      E6 A1   INC $A1
BF5F      D0 F5   BNE $BF56
BF61   2C 80 C0   BIT $C080
BF64         60   RTS
```

Continuing from $041F...

```
241F   AD 83 C0   LDA $C083     set low-level reset vectors and
2422   AD 83 C0   LDA $C083     page 3 vectors to point to
2425      A0 00   LDY #$00      $BF00—presumably The
2427      A9 BF   LDA #$BF      Badlands (from which there is
2429   8C FC FF   STY $FFFC     no return)
242C   8D FD FF   STA $FFFD
242F   8C F2 03   STY $03F2
2432   8D F3 03   STA $03F3
2435      A0 03   LDY #$03
2437   8C F0 03   STY $03F0
243A   8D F1 03   STA $03F1
243D      84 38   STY $38
243F      85 39   STA $39
2441      49 A5   EOR #$A5
2443   8D F4 03   STA $03F4

*BF00L
```

```
BF00      A9 D2   LDA #$D2      There are multiple entry
BF02   2C A9 D0   BIT $D0A9     points here: $BF00, $BF03,
BF05   2C A9 CC   BIT $CCA9     $BF06, and $BF09 (hidden in
BF08   2C A9 A1   BIT $A1A9     this listing by the "BIT"
BF0B         48   PHA           opcodes).

BF0C   20 48 BF   JSR $BF48     zap the language card again

BF0F   20 2F FB   JSR $FB2F     TEXT/HOME/NORMAL
BF12   20 58 FC   JSR $FC58
BF15   20 84 FE   JSR $FE84

BF18         68   PLA           Depending on the initial entry
BF19   8D 00 04   STA $0400     point, this displays a different
                                character in the top left
                                corner of the screen

BF1C      A0 00   LDY #$00      now wipe all of main memory
BF1E         98   TYA
BF1F   99 00 BE   STA $BE00,Y
BF22         C8   INY
BF23      D0 FA   BNE $BF1F
BF25   CE 21 BF   DEC $BF21

BF28   2C 30 C0   BIT $C030     while playing a sound
BF2B   AD 21 BF   LDA $BF21
BF2E      C9 08   CMP #$08
BF30      B0 EA   BCS $BF1C

BF32   8D F3 03   STA $03F3     munge the reset vector
BF35   8D F4 03   STA $03F4

BF38   AD 66 BF   LDA $BF66     and reboot from whence we
BF3B         4A   LSR           came
BF3C         4A   LSR
BF3D         4A   LSR
BF3E         4A   LSR
BF3F      09 C0   ORA #$C0
BF41      E9 00   SBC #$00
BF43         48   PHA
BF44      A9 FF   LDA #$FF
BF46         48   PHA
BF47         60   RTS
```

Yeah, let's try not to end up there.

Continuing from $0446...

```
2446      A9 07   LDA #$07
2448   20 00 BE   JSR $BE00

*BE00L
BE00      A2 13   LDX #$13      entry point #1

BE02   2C A2 0A   BIT $0AA2     entry point #2 (hidden
                                behind a BIT opcode, but it's
                                "LDX #$0A")

BE05   8E 6E BE   STX $BE6E     (!) modify the code later
                                based on which entry point
                                we called
```

```
BE08  8D 90 BE   STA $BE90    The rest of this routine is a
BE0B  CD 65 BF   CMP $BF65    garden variety drive seek. The
BE0E     F0 59   BEQ $BE69    target phase (track x 2) is in
BE10     A9 00   LDA #$00     the accumulator on entry.
BE12  8D 91 BE   STA $BE91
BE15  AD 65 BF   LDA $BF65
BE18  8D 92 BE   STA $BE92
BE1B        38   SEC
BE1C  ED 90 BE   SBC $BE90
BE1F     F0 37   BEQ $BE58
BE21     B0 07   BCS $BE2A
BE23     49 FF   EOR #$FF
BE25  EE 65 BF   INC $BF65
BE28     90 05   BCC $BE2F
BE2A     69 FE   ADC #$FE
BE2C  CE 65 BF   DEC $BF65
BE2F  CD 91 BE   CMP $BE91
BE32     90 03   BCC $BE37
BE34  AD 91 BE   LDA $BE91
BE37     C9 0C   CMP #$0C
BE39     B0 01   BCS $BE3C
BE3B        A8   TAY
BE3C        38   SEC
BE3D  20 5C BE   JSR $BE5C
BE40  B9 78 BE   LDA $BE78,Y
BE43  20 6D BE   JSR $BE6D
BE46  AD 92 BE   LDA $BE92
BE49        18   CLC
BE4A  20 5F BE   JSR $BE5F
BE4D  B9 84 BE   LDA $BE84,Y
BE50  20 6D BE   JSR $BE6D
BE53  EE 91 BE   INC $BE91
BE56     D0 BD   BNE $BE15
BE58  20 6D BE   JSR $BE6D
BE5B        18   CLC
BE5C  AD 65 BF   LDA $BF65
BE5F     29 03   AND #$03
BE61        2A   ROL
BE62  0D 66 BF   ORA $BF66
BE65        AA   TAX
BE66  BD 80 C0   LDA $C080,X
BE69  AE 66 BF   LDX $BF66
BE6C        60   RTS

BE6D     A2 13   LDX #$13     (value of X may be modified
BE6F        CA   DEX          depending on which entry
BE70     D0 FD   BNE $BE6F    point was called)
BE72        38   SEC
BE73     E9 01   SBC #$01
BE75     D0 F6   BNE $BE6D
BE77        60   RTS
BE78 [01 30 28 24 20 1E 1D 1C]
BE80 [1C 1C 1C 1C 70 2C 26 22]
BE88 [1F 1E 1D 1C 1C 1C 1C 1C]
```

The fact that there are two entry points is interesting. Calling `$BE00` will set X to `#$13`, which will end up in `$BE6E`, so the wait routine at `$BE6D` will wait long enough to go to the next phase (a.k.a. half a track). Nothing unusual there; that's how all drive seek routines work. But calling `$BE03` instead of `$BE00` will set `X` to `#$0A`, which will make the wait routine burn fewer CPU cycles while the drive head is moving, so it will only move half a phase (a.k.a. a quarter track). That is potentially very interesting.

Continuing from `$044B`…

```
244B     A9 05   LDA #$05
244D     85 33   STA $33
244F     A2 03   LDX #$03
2451     86 36   STX $36
2453     A0 00   LDY #$00
2455     A5 33   LDA $33
2457     84 34   STY $34
2459     85 35   STA $35
```

Now (`$34`) points to `$0500`.

```
245B  AE 66 BF   LDX $BF66     find a 3-nibble prologue ("B5
245E  BD 8C C0   LDA $C08C,X   DE F7")
2461     10 FB   BPL $245E
2463     C9 B5   CMP #$B5
2465     D0 F7   BNE $245E
2467  BD 8C C0   LDA $C08C,X
246A     10 FB   BPL $2467
246C     C9 DE   CMP #$DE
246E     D0 F3   BNE $2463
2470  BD 8C C0   LDA $C08C,X
2473     10 FB   BPL $2470
2475     C9 F7   CMP #$F7
2477     D0 F3   BNE $246C

2479  BD 8C C0   LDA $C08C,X   read 4-4-encoded data into
247C     10 FB   BPL $2479     $0500+
247E        2A   ROL
247F     85 37   STA $37
2481  BD 8C C0   LDA $C08C,X
2484     10 FB   BPL $2481
2486     25 37   AND $37
2488     91 34   STA ($34),Y
248A        C8   INY
248B     D0 EC   BNE $2479
248B     D0 EC   BNE $2479
248D  0E FF FF   ASL $FFFF

2490  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue ("D5")
2493     10 FB   BPL $2490
2495     C9 D5   CMP #$D5
2497     D0 B6   BNE $244F
2499     E6 35   INC $35

249B     C6 36   DEC $36       3 sectors (initialized at $0451)
249D     D0 DA   BNE $2479

249F        60   RTS           and exit via RTS
```

We've read 3 more sectors into `$0500+`, overwriting the code we read earlier (but moved to `$BD00+`), and once again we simply exit and let the stack tell us where we're going next.

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
            next return address
```

$\texttt{\$04FF} + 1 = \texttt{\$0500}$, the code we just read.

And that's where I get to interrupt the boot.

Brøderbund

Return of the Jedi

```
*C500G
...
]CALL -151
*BLOAD TRACE3
.
.  [same as previous trace]
.
```

reboot because I disconnected and overwrote DOS to examine the previous code chunk at $BD00+

```
9714     A9 21   LDA #$21
9716  8D D4 01   STA $01D4
9719     A9 97   LDA #$97
971B  8D D5 01   STA $01D5
```

Patch the stack again, but slightly later, at $01D4. (The previous trace patched it at $01D2.)

```
971E     A2 CF   LDX #$CF
9720        9A   TXS
9721        60   RTS
```

continue the boot

```
9722     A2 04   LDX #$03
9724     A0 00   LDY #$00
9726  B9 00 05   LDA $0500,Y
9729  99 00 25   STA $2500,Y
972C        C8   INY
972D     D0 F7   BNE $9726
972F  EE 28 97   INC $9728
9732  EE 2B 97   INC $972B
9735        CA   DEX
9736     D0 EE   BNE $9726
```

(callback is here) We just executed all the code up to and including the "RTS" at $049F, so now let's copy the latest code at $0500..$07FF to higher memory so it survives a reboot.

```
9738  AD E8 C0   LDA $C0E8
973B  4C 00 C5   JMP $C500
```

reboot to my work disk

```
*BSAVE TRACE4,A$9600,L$13E
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT2
0500-07FF,A$2500,L$300
]CALL -151
```

Again, I'm going to leave this at $2500 because I can't examine code on the text page. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2500L
2500     A9 02   LDA #$02
2502  20 00 BE   JSR $BE00
```

seek to track 1

```
2505  AE 66 BF   LDX $BF66
2508     A0 00   LDY #$00
250A     A9 20   LDA #$20
250C     85 30   STA $30
250E        88   DEY
250F     D0 04   BNE $2515
2511     C6 30   DEC $30
2513     F0 3C   BEQ $2551
```

get slot number x16 (set a long time ago, at $0419)

```
2515  BD 8C C0   LDA $C08C,X
2518     10 FB   BPL $2515
251A     C9 D5   CMP #$D5
251C     D0 F0   BNE $250E
251E  BD 8C C0   LDA $C08C,X
2521     10 FB   BPL $251E
2523     C9 FF   CMP #$FF
2525     D0 F3   BNE $251A
2527  BD 8C C0   LDA $C08C,X
252A     10 FB   BPL $2527
252C     C9 DD   CMP #$DD
252E     D0 F3   BNE $2523
```

find a 3-nibble prologue ("D5 FF DD")

```
2530     A0 00   LDY #$00
2532  BD 8C C0   LDA $C08C,X
2535     10 FB   BPL $2532
2537        38   SEC
2538        2A   ROL
2539     85 30   STA $30
253B  BD 8C C0   LDA $C08C,X
253E     10 FB   BPL $253B
2540     25 30   AND $30
```

read 4-4-encoded data

```
2542  99 00 B0   STA $B000,Y
2545        C8   INY
2546     D0 EA   BNE $2532
```

into $B000 (hard-coded here, was not modified earlier unless I missed something)

```
2548  BD 8C C0   LDA $C08C,X
254B     10 FB   BPL $2548
254D     C9 D5   CMP #$D5
254F     F0 0B   BEQ $255C
```

find a 1-nibble epilogue ("D5")

```
2551     A0 00   LDY #$00
2553  B9 00 07   LDA $0700,Y
2556  99 00 B0   STA $B000,Y
2559        C8   INY
255A     D0 F7   BNE $2553
```

This is odd. If the epilogue doesn't match, it's not an error. Instead, it appears that we simply copy a page of data that we read earlier (at $0700).

```
255C  20 F0 05   JSR $05F0
```

execution continues here regardless

```
*25F0L
25F0     A0 56   LDY #$56
25F2     A9 BD   LDA #$BD
25F4        48   PHA
25F5     A9 FF   LDA #$FF
25F7        48   PHA
25F8     A9 07   LDA #$07
25FA        60   RTS
```

Weird, but OK. This ends up calling $BE00 with A=$07, which will seek to track 3.5.

And now we're on half tracks.

Continuing from $055F...

```
255F  BD 8C C0   LDA $C08C,X
2562     10 FB   BPL $255F
2564     C9 DD   CMP #$DD
2566     D0 F7   BNE $255F
2568  BD 8C C0   LDA $C08C,X
256B     10 FB   BPL $2568
256D     C9 EF   CMP #$EF
256F     D0 F3   BNE $2564
2571  BD 8C C0   LDA $C08C,X
2574     10 FB   BPL $2571
2576     C9 AD   CMP #$AD
2578     D0 F3   BNE $256D
```

find a 3-nibble prologue ("DD EF AD")

```
257A    A0 00   LDY #$00      read a 4-4 encoded byte (two
257C BD 8C C0   LDA $C08C,X   nibbles on disk = 1 byte in
257F    10 FB   BPL $257C     memory)
2581       38   SEC
2582       2A   ROL
2583    85 00   STA $00
2585 BD 8C C0   LDA $C08C,X
2588    10 FB   BPL $2585
258A    25 00   AND $00

258C       48   PHA           push the byte to the stack
                              (WTF?)

258D       88   DEY           repeat for $100 bytes
258E    D0 EC   BNE $257C

2590 BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue
2593    10 FB   BPL $2590     ("D5")
2595    C9 D5   CMP #$D5
2597    D0 C3   BNE $255C

2599 CE 9C 05   DEC $059C (!)
259C    61 00   ADC ($00,X)
```

(!) Self-modifying code alert! WOO WOO. I'll use this symbol whenever one instruction modifies the next instruction. When this happens, the disassembly listing is misleading because the opcode will be changed by the time the second instruction is executed.

In this case, the DEC at `$0599` modifies the opcode at `$059C`, so that's not really an "ADC." By the time we execute the instruction at `$059C`, it will have been decremented to `#$60`, a.k.a. "RTS."

One other thing: we've read `$100` bytes and pushed all of them to the stack. The stack is only `$100` bytes (`$0100..$01FF`), so this completely obliterates any previous values.

We haven't changed the stack pointer, though. That means the "RTS‘" at `$059C` will still look at `$01D6` to find the next "return" address. That used to be `‘‘4F 04’’`, but now it's been overwritten with new values, along with the rest of the stack. That's some serious Jedi mind trick stuff.

"These aren't the return addresses you're looking for."

"These aren't the return addresses we're looking for."

"He can go about his bootloader."

"You can go about your bootloader."

"Move along."

"Move along... move along."

In Which We Move Along

Luckily, there's plenty of room at `$0599`. I can insert a JMP to call back to code under my control, where I can save a copy of the stack. (And `$B000` as well, whatever that is.) I get to ensure I don't disturb the stack before I save it, so no JSR, PHA, PHP, or TXS. I think I can manage that. JMP doesn't disturb the stack, so that's safe for the callback.

```
*BLOAD TRACE4
.
.  [same as previous trace]
.
9722    A9 4C   LDA #$4C      set up a JMP $9734 at $0599
9724 8D 99 05   STA $0599
9727    A9 34   LDA #$34
9729 8D 9A 05   STA $059A
972C    A9 97   LDA #$97
972E 8D 9B 05   STA $059B

9731 4C 00 05   JMP $0500     continue the boot

9734    A0 00   LDY #$00      (callback is here) Copy $B000
9736 B9 00 B0   LDA $B000,Y   and $0100 to higher memory
9739 99 00 20   STA $2000,Y   so they survive a reboot
973C B9 00 01   LDA $0100,Y
973F 99 00 21   STA $2100,Y
9742       C8   INY
9743    D0 F1   BNE $9736

9745 AD E8 C0   LDA $C0E8     reboot to my work disk
9748 4C 00 C5   JMP $C500

*BSAVE TRACE5,A$9600,L$14B
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT2
B000-B0FF,A$2000,L$100
]BSAVE BOOT2
0100-01FF,A$2100,L$100
]CALL -151
```

Remember, the stack *pointer* hasn't changed. Now that I have the new stack *data*, I can just look at the right index in the captured stack page to see where the bootloader continues once it issues the "RTS" at `$059C`.

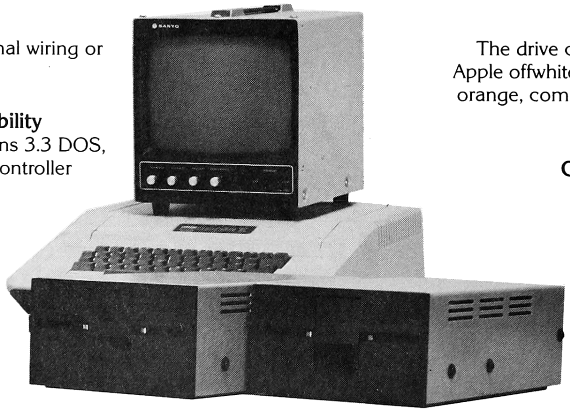
```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
                       next return address
```

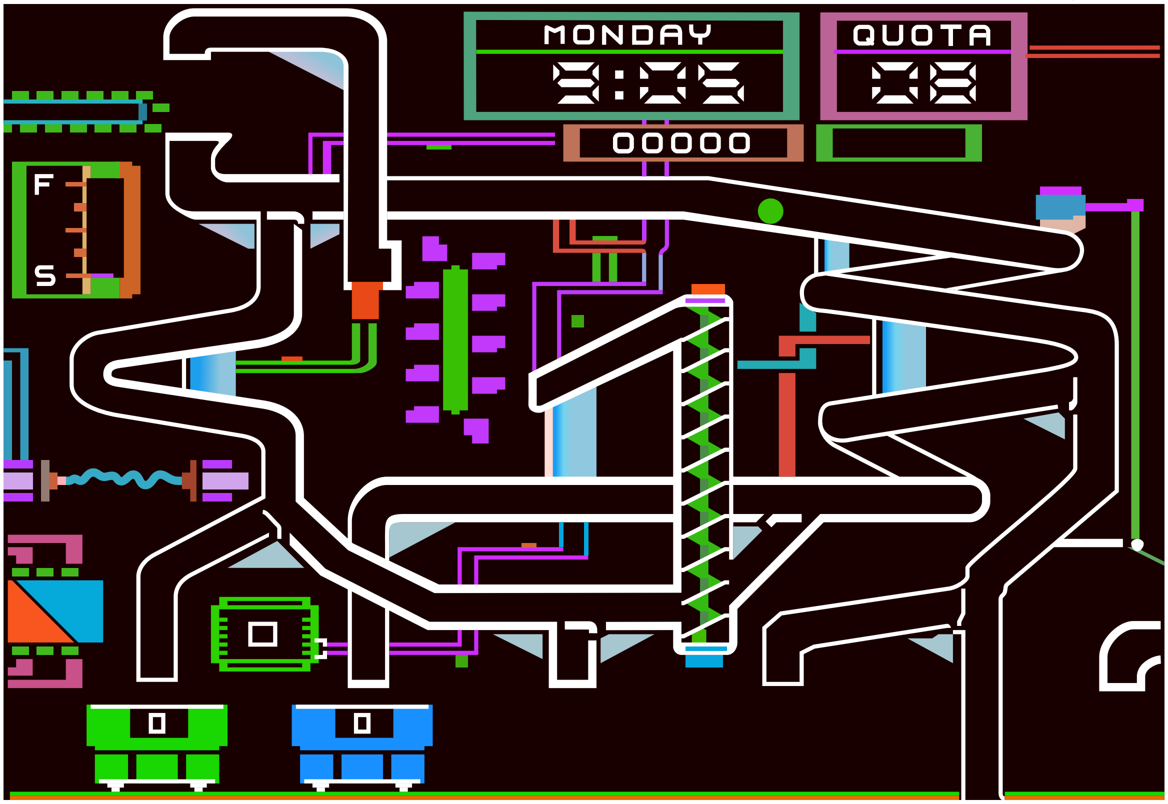
That's part of the stack page I just captured, so it's already in memory.

```
*2126L
```

Another disk read routine! The fourth? Fifth? I've truly lost count.

```
2126 BD 8C C0   LDA $C08C,X   find a 3-nibble prologue ("BF
2129    10 FB   BPL $2126     BE D4")
212B    C9 BF   CMP #$BF
212D    D0 F7   BNE $2126
212F BD 8C C0   LDA $C08C,X
2132    10 FB   BPL $212F
2134    C9 BE   CMP #$BE
2136    D0 F3   BNE $212B
2138 BD 8C C0   LDA $C08C,X
213B    10 FB   BPL $2138
213D    C9 D4   CMP #$D4
213F    D0 F3   BNE $2134
```




MONDAY
9:05
QUOTA
08
00000
F
S
0
0

```
2141    A0 00   LDY #$00       read 4-4-encoded data
2143 BD 8C C0   LDA $C08C,X
2146    10 FB   BPL $2143
2148       38   SEC
2149       2A   ROL
214A 8D 00 02   STA $0200
214D BD 8C C0   LDA $C08C,X
2150    10 FB   BPL $214D
2152 2D 00 02   AND $0200

2155 59 00 01   EOR $0100,Y    decrypt the data from disk by
                               using this entire page of code
                               (in the stack page) as the
                               decryption key (more on this
                               later)

2158 99 00 00   STA $0000,Y    and store it in zero page
215B       C8   INY
215C    D0 E5   BNE $2143

215E BD 8C C0   LDA $C08C,X    find a 1-nibble epilogue
2161    10 FB   BPL $215E      ("D5")
2163    C9 D5   CMP #$D5
2165    D0 BF   BNE $2126

2167       60   RTS            and exit via RTS
```

And we're back on the stack again.

```
*21D0.
21D0 F0 78 AD D8 02 85 25 01
21D8 57 FF 57 FF 57 FF 57 FF
21E0 57 FF 22 01 FF 05 B1 4C
```

The six `57 FF` words and the following `22 01` word are the next return addresses.

$FF57 + 1 = $FF58, which is a well-known address in ROM that is always an "RTS" instruction. So this will burn through several return addresses on the stack in short order, then finally arrive at $0123, in memory at $2123.

```
*2123L
2123   6C 28 00    JMP ($0028)
```

...which is in the new zero page that was just read from disk.

And to think, we've loaded basically nothing of consequence yet. The screen is still black. We have 3 pages of code at `$BD00..$BFFF`. There's still some code on the text screen, but who knows if we'll ever call it again. Now we're off to zero page for some reason.

Un. Be. Lievable.

By Perseverance The Snail Reached The Ark

I can't touch the code on the stack, because it's used as a decryption key. I mean, I could theoretically change a few bytes of it, then calculate the proper decrypted bytes on zero page by hand. But no.

Instead, I'm just going to copy this latest disk routine wholesale. It's short and has no external dependencies, so why not? Then I can capture the decrypted zero page and see where that `JMP ($0028)` is headed.

```
*BLOAD TRACE5
*9734<2126.2166M
```

Here's the entire disassembly listing of boot trace #6:

```
96F8    A9 05   LDA #$05       patch boot0 so it calls my
96FA 8D 38 08   STA $0838      routine instead of jumping to
96FD    A9 97   LDA #$97       $0301
96FF 8D 39 08   STA $0839

9702 4C 01 08   JMP $0801      start the boot

9705    84 48   STY $48        (callback #1 is here)
9707    A0 00   LDY #$00       reproduce the decryption loop
9709 B9 00 03   LDA $0300,Y    that was originally at $0320
970C    45 48   EOR $48
970E 99 00 01   STA $0100,Y
9711       C8   INY
9712    D0 F5   BNE $9709

9714    A9 21   LDA #$21       patch the stack so it jumps to
9716 8D D4 01   STA $01D4      my callback #2 instead of
9719    A9 97   LDA #$97       continuing to $0500
971B 8D D5 01   STA $01D5

971E    A2 CF   LDX #$CF       continue the boot
9720       9A   TXS
9721       60   RTS

9722    A9 4C   LDA #$4C       (callback #2) set up callback
9724 8D 99 05   STA $0599      #3 instead of passing control
9727    A9 34   LDA #$34       to the disk read routine at
9729 8D 9A 05   STA $059A      $0126
972C    A9 97   LDA #$97
972E 8D 9B 05   STA $059B

9731 4C 00 05   JMP $0500      continue the boot

9734 BD 8C C0   LDA $C08C,X    (callback #3) disk read
9737    10 FB   BPL $9734      routine copied wholesale from
9739    C9 BF   CMP #$BF       $0126..$0166 that reads a
973B    D0 F7   BNE $9734      sector and decrypts it into
973D BD 8C C0   LDA $C08C,X    zero page
9740    10 FB   BPL $973D
9742    C9 BE   CMP #$BE
9744    D0 F3   BNE $9739
9746 BD 8C C0   LDA $C08C,X
9749    10 FB   BPL $9746
974B    C9 D4   CMP #$D4
974D    D0 F3   BNE $9742
974F    A0 00   LDY #$00
9751 BD 8C C0   LDA $C08C,X
9754    10 FB   BPL $9751
9756       38   SEC
9757       2A   ROL
9758 8D 00 02   STA $0200
975B BD 8C C0   LDA $C08C,X
975E    10 FB   BPL $975B
9760 2D 00 02   AND $0200
9763 59 00 01   EOR $0100,Y
9766 99 00 00   STA $0000,Y
9769       C8   INY
976A    D0 E5   BNE $9751
976C BD 8C C0   LDA $C08C,X
976F    10 FB   BPL $976C
9771    C9 D5   CMP #$D5
9773    D0 BF   BNE $9734
```

```
                                execution falls through here
9775     A0 00   LDY #$00      now capture the decrypted
9777  B9 00 00   LDA $0000,Y   zero page
977A  99 00 20   STA $2000,Y
977D        C8   INY
977E     D0 F7   BNE $9777

9780  AD E8 C0   LDA $C0E8     turn off the slot 6 drive motor

9783  4C 00 C5   JMP $C500     reboot to my work disk

*BSAVE TRACE6,A$9600,L$186
                               Whew. Let's do it.
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT3
0000-00FF,A$2000,L$100
]CALL -151
*2028.2029
2028 D0 06
```

OK, the `JMP ($0028)` points to `$06D0`, which I captured earlier. It's part of the second chunk we read into the text page. (Not the first chunk—that was copied to `$BD00+` then overwritten.) So it's in the "`BOOT2 0500-07FF`" file, not the "`BOOT1 0400-07FF`" file.

```
*BLOAD BOOT2 0500-07FF,A$2500
*26D0L
26D0     A2 00   LDX #$00
26D2  EE D5 06   INC $06D5 (!)
26D5     C9 EE   CMP #$EE
```

Oh joy, more self-modifying code.

```
*26D5:CA
*26D5L
26D5        CA   DEX
26D6  EE D9 06   INC $06D9 (!)
26D9        0F   ???

*26D9:10
*26D9L
26D9     10 FB   BPL $26D6     branch is never taken,
26DB  CE DE 06   DEC $06DE (!) because we just DEX'd from
26DE     61 A0   ADC ($A0,X)   #$00 to #$FF

*26DE:60
*26DEL
26DE        60   RTS
```

And now we're back on the stack.

```
*BLOAD BOOT2 0100-01FF,A$2100
*21E0.
*21E0. 57 FF 22 01 FF 05 B1 4C
                   └─┬─┘
           next return address
```

`$05FF + 1 = $0600`, which is already in memory at `$2600`.

```
*2600L
2600     A0 00   LDY #$00      destroy stack by pushing the
2602        48   PHA           same value $100 times
2603        88   DEY
2604     D0 FC   BNE $2602
```

I guess we're done with all that code on the stack page. I mean, I hope we're done with it, since it all just disappeared.

```
2606     A2 FF   LDX #$FF      reset the stack pointer
2608        9A   TXS

2609  EE 0C 06   INC $060C (!)
260C        A8   TAY
```

Oh joy.

```
*260C:A9
*260CL
260C     A9 27   LDA #$27
260E  EE 11 06   INC $0611 (!)
2611        17   ???

*2611:18
*2611L
2611        18   CLC
2612  EE 15 06   INC $0615 (!)
2615        68   PLA

*2615:69
*2615L
2615     69 D9   ADC #$D9
2617  EE 1A 06   INC $061A (!)
261A        4B   ???

*261A:4C
*261AL
261A  4C 90 FD   JMP $FD90
```

Wait, what?

```
*FD90L
FD90     D0 5B   BNE $FDED
```

Despite the fact that the accumulator is `#$00` (because `#$27 + #$D9 = #$00`), the `INC` at `$0617` affects the Z register and causes this branch to be taken, because the final value of `$061A` was not zero.

```
*FDEDL
FDED  6C 36 00   JMP ($0036)
```

Of course, this is the standard output character routine, which routes through the output vector at `($0036)`. And we just set that vector, along with the rest of zero page. So what is it?

```
*2036.2037
2036 6F BF
```

Oh joy. Let's see, `$BD00..$BFFF` was copied earlier from `$0500..$07FF`, but from the first time we read into the text page, not the second time we read into text page. So it's in the "BOOT1 0400-07FF" file, not the "BOOT2 0500-07FF" file.

```
*BLOAD BOOT1 0400-07FF,A$2400
*FE89G FE93G                   disconnect DOS
```

```
*BD00<2500.27FFM                  move code into place
*BF6FL
BF6F     C9 07   CMP #$07
BF71     90 03   BCC $BF76
BF73  6C 3A 00   JMP ($003A)

*203A.203B
203A F0 FD
BF76     85 5F   STA $5F          save input value

BF78        A8   TAY              use value as an index into an
BF79  B9 68 BF   LDA $BF68,Y      array

BF7C  8D 82 BF   STA $BF82        (!)self-modifying code
BF7F     A9 00   LDA #$00         alert—this changes the
BF81  20 D0 BE   JSR $BED0        upcoming JSR at $BF81
```

Amazing. So this "output" vector does actually print characters through the standard `$FDF0` text print routine, but only if the character to be printed is at least `#$07`. If it's less than `#$07`, the "character" is treated as a command. Each command gets routed to a different routine somewhere in `$BExx`. The low byte of each routine is stored in the array at `$BF68`, and the "STA" at `$BF7C` modifies the "JSR" at `$BF81` to call the appropriate address.

```
*BF68.
BF68 D0 DF D0 D0 FD FD D0
```

Since `A = #$00` this time, the call is unchanged and we `JSR $BED0`. Other input values may call `$BEDF` or `$BEFD` instead.

```
*BED0L
BED0     A5 60   LDA $60          use the "value" of $C050 to
BED2  4D 50 C0   EOR $C050        produce a pseudo-random
BED5     85 60   STA $60          number between #$01 and
BED7     29 0F   AND #$0F         #$0E

BED9     F0 F5   BEQ $BED0        not #$00

BEDB     C9 0F   CMP #$0F         not #$0F
BEDD     F0 F1   BEQ $BED0

BEDF  20 66 F8   JSR $F866        set the lo-res plotting color
                                  (in zero page $30) to the
                                  random-ish value we just
                                  produced

BEE2     A9 17   LDA #$17         fill the lo-res graphics screen
BEE4        48   PHA              with blocks of that color

BEE5  20 47 F8   JSR $F847        calculates the base address for
BEE8     A0 27   LDY #$27         this line in memory and puts
BEEA     A5 30   LDA $30          it in $26/$27
BEEC     91 26   STA ($26),Y
BEEE        88   DEY
BEEF     10 FB   BPL $BEEC
BEF1        68   PLA

BEF2        38   SEC              do it for all 24 ($17) rows of
BEF3     E9 01   SBC #$01         the screen
BEF5     10 ED   BPL $BEE4

BEF7  AD 56 C0   LDA $C056        and switch to lo-res graphics
BEFA  AD 54 C0   LDA $C054        mode
BEFD        60   RTS
```

This explains why the original disk fills the screen with a different color every time it boots.

But wait, these commands do so much more than just fill the screen.

Continuing from `$BF84`...

```
BF84     A5 5F   LDA $5F
BF86     C9 04   CMP #$04
BF88     D0 03   BNE $BF8D
BF8A  4C 00 BD   JMP $BD00
```

If `A = #$04`, we exit via `$BD00`, which I'll investigate later.

```
BF8D     C9 05   CMP #$05
BF8F     D0 03   BNE $BF94
BF91  6C 82 BF   JMP ($BF82)
```

If `A = #$05`, we exit via `($BF82)`, which is the same thing we just called via the self-modified `JSR` at `$BF81`.

For all other values of A, we do this:

```
BF94  20 B0 BE   JSR $BEB0

*BEB0L
BEB0     A2 60   LDX #$60         another layer of encryption!
BEB2  BD 9F BF   LDA $BF9F,X
BEB5  5D 00 BE   EOR $BE00,X

BEB8  9D 9F BF   STA $BF9F,X      and it's decrypting the code
BEBB        CA   DEX              that we're about to run
BEBC     10 F4   BPL $BEB2
BEBE  AE 66 BF   LDX $BF66
BEC1        60   RTS
```

This is self-contained, so I can just run it right now and see what ends up at `$BF9F`.

```
*BEB0G
```

Continuing from `$BF97`...

```
BF97     A0 00   LDY #$00
BF99     A9 B2   LDA #$B2
BF9B     84 44   STY $44
BF9D     85 45   STA $45

BF9F  BD 89 C0   LDA $C089,X      everything beyond this point
                                  was encrypted, but we just
                                  decrypted it in $BEB0

BFA2  BD 8C C0   LDA $C08C,X      find a 3-nibble prologue
BFA5     10 FB   BPL $BFA2        (varies, based on whatever
BFA7     C5 40   CMP $40          the hell is in zero page
BFA9     D0 F7   BNE $BFA2        $40/$41/$42 at this point)
BFAB  BD 8C C0   LDA $C08C,X
BFAE     10 FB   BPL $BFAB
BFB0     C5 41   CMP $41
BFB2     D0 F3   BNE $BFA7
BFB4  BD 8C C0   LDA $C08C,X
BFB7     10 FB   BPL $BFB4
BFB9     C5 42   CMP $42
BFBB     D0 F3   BNE $BFB0
```

```
BFBD   BD 8C C0   LDA $C08C,X   read 4-4-encoded data
BFC0      10 FB   BPL $BFBD
BFC2         38   SEC
BFC3         2A   ROL
BFC4      85 46   STA $46
BFC6   BD 8C C0   LDA $C08C,X
BFC9      10 FB   BPL $BFC6
BFCB      25 46   AND $46

BFCD      91 44   STA ($44),Y   store in memory starting at
BFCF         C8   INY           $B200 (set at $BF9B)
BFD0      D0 EB   BNE $BFBD
BFD2      E6 45   INC $45
BFD4   BD 8C C0   LDA $C08C,X
BFD7      10 FB   BPL $BFD4
BFD9      C5 43   CMP $43
BFDB      D0 BA   BNE $BF97

BFDD      A5 45   LDA $45       read into $B200, $B300, and
BFDF      49 B5   EOR #$B5      $B400, then stop
BFE1      D0 DA   BNE $BFBD
BFE3         48   PHA ; A=00
BFE4      A5 45   LDA $45 ;
A=B5
BFE6      49 8E   EOR #$8E ;
A=3B
BFE8         48   PHA
BFE9         60   RTS
```

So we push #$00 and #$3B to the stack, then exit via RTS. That will "return" to $003C, which is in memory at $203C.

```
*203CL
203C   4C 00 B2   JMP $B200
```

And that's the code we just read from disk, which means I get to set up another boot trace to capture it.

In Which We Flutter For A Day And Think It Is Forever

I'll reboot my work disk again, since I disconnected DOS to examine the code at $BD00..$BFFF.

```
*C500G
...
]CALL -151
*BLOAD TRACE6
.
.  [same as previous trace, up
to and
.  including the inline disk
read
.  routine copied from $0126
that
.  decrypts a sector into zero
page]
.
9775      A9 80   LDA #$80      change the JMP address at
9777      85 3D   STA $3D       $003C so it points to my
9779      A9 97   LDA #$97      callback instead of continuing
977B      85 3E   STA $3E       to $B200

977D   4C 00 06   JMP $0600     continue the boot

9780      A2 03   LDX #$03      (callback is here) copy the
9782   B9 00 B2   LDA $B200,Y   new code to the graphics page
9785   99 00 22   STA $2200,Y   so it survives a reboot
9788         C8   INY
9789      D0 F7   BNE $9782
978B   EE 84 97   INC $9784
978E   EE 87 97   INC $9787
9791         CA   DEX
9792      D0 EE   BNE $9782

9794   AD E8 C0   LDA $C0E8     reboot to my work disk
9797   4C 00 C5   JMP $C500

*BSAVE TRACE7,A$9600,L$19A
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE
OBJ.B200-B4FF,A$2200,L$300
]CALL -151
*B200<2200.24FFM
*B200L
B200      A9 04   LDA #$04
B202   20 00 B4   JSR $B400
B205      A9 00   LDA #$00
B207      85 5A   STA $5A
B209   20 00 B3   JSR $B300
B20C   4C 00 B5   JMP $B500
```

$B400 is a disk seek routine, identical to the one at $BE00. (It even has the same dual entry points for seeking by half track and quarter track, at $B400 and $B403.) There's nothing at $B500 yet, so the routine at $B300 must be another disk read.

```
*B300L

B300      A0 00   LDY #$00      some zero page initialization
B302      A9 B5   LDA #$B5
B304      84 59   STY $59
B306         48   PHA
B307   20 30 B3   JSR $B330

*B330L

B330         48   PHA           more zero page initialization
B331      A5 5A   LDA $5A
B333      29 07   AND #$07
B335         A8   TAY
B336   B9 50 B3   LDA $B350,Y
B339      85 50   STA $50
B33B      A5 5A   LDA $5A
B33D         4A   LSR
B33E      09 AA   ORA #$AA
B340      85 51   STA $51
B342      A5 5A   LDA $5A
B344      09 AA   ORA #$AA
B346      85 52   STA $52
B348         68   PLA
B349      E6 5A   INC $5A
B34B   4C 60 B3   JMP $B360

*B350.
B350 D5 B5 B7 BC DF D4 B4 DB
```

That could be an array of nibbles. Maybe a rotating prologue? Or a decryption key?

Oh joy. Another disk read routine.

```
*B360L
B360      85 54   STA $54
B362      A2 02   LDX #$02
B364      86 57   STX $57
B366      A0 00   LDY #$00
B368      A5 54   LDA $54
B36A      84 55   STY $55
B36C      85 56   STA $56

B36E   AE 66 BF   LDX $BF66     find a 3-nibble prologue
B371   BD 8C C0   LDA $C08C,X   (varies, based on the zero
B374      10 FB   BPL $B371     page locations that were
B376      C5 50   CMP $50       initialized at $B330 based on
B378      D0 F7   BNE $B371     the array at $B350)
B37A   BD 8C C0   LDA $C08C,X
B37D      10 FB   BPL $B37A
B37F      C5 51   CMP $51
B381      D0 F3   BNE $B376
B383   BD 8C C0   LDA $C08C,X
B386      10 FB   BPL $B383
B388      C5 52   CMP $52
B38A      D0 F3   BNE $B37F

B38C   BD 8C C0   LDA $C08C,X   read a 4-4-encoded sector
B38F      10 FB   BPL $B38C
B391         2A   ROL
B392      85 58   STA $58
B394   BD 8C C0   LDA $C08C,X
B397      10 FB   BPL $B394
B399      25 58   AND $58

B39B      91 55   STA ($55),Y   store the data into ($55)
B39D         C8   INY
B39E      D0 EC   BNE $B38C

B3A0   0E FF FF   ASL $FFFF     find a 1-nibble epilogue
B3A3   BD 8C C0   LDA $C08C,X   ("D4")
B3A6      10 FB   BPL $B3A3
B3A8      C9 D4   CMP #$D4
B3AA      D0 B6   BNE $B362
B3AC      E6 56   INC $56
B3AE      C6 57   DEC $57
B3B0      D0 DA   BNE $B38C
B3B2         60   RTS
```

Let's see:

$57 is the sector count. Initially #$02 (set at $B364), decremented at $B3AE.

$56 is the target page in memory. Set at $B36C to the accumulator, which is set at $B368 to the value of address $54, which is set at $B360 to the accumulator, which is set at $B348 by the PLA, which was pushed to the stack at $B330, which was originally set at $B302 to a constant value of #$B5. Then $56 is incremented (at $B3AC) after reading and decoding $100 bytes worth of data from disk.

$55 is #$00, as set at $B36A.

So this reads two sectors into $B500..$B6FF and returns to the caller.

Backtracking to $B30A...

```
B30A      A4 59   LDY $59       $59 is initially #$00 (set at
B30C         18   CLC           $B304)

B30D   AD 65 BF   LDA $BF65     current phase (track x 2)
```

```
B310   79 28 B3   ADC $B328,Y   new phase

B313   20 03 B4   JSR $B403     move the drive head to the
                                new phase, but using the
                                second entry point, which
                                uses a reduced timing loop (!)

B316         68   PLA           this pulls the value that was
                                pushed to the stack at $B306,
                                which was the target memory
                                page to store the data being
                                read from disk by the routine
                                at $B360
B317         18   CLC           page += 2
B318      69 02   ADC #$02

B31A      A4 59   LDY $59       counter += 1
B31C         C8   INY

B31D      C0 04   CPY #$04      loop for 4 iterations
B31F      90 E3   BCC $B304
B321         60   RTS
```

So we're reading two sectors at a time, four times, into $B500+. `2 x 4 = 8`, so we're loading into $B500..$BCFF. That completely fills the gap in memory between the code at $B200..$B4FF (this chunk) and the code at $BD00..$BFFF (copied much earlier), which strongly suggests that my analysis is correct.

But what's going on with the weird drive seeking?

There is some definite weirdness here, and it's centered around the array at $B328. At $B200, we called the main entry point for the drive seek routine at $B400 to seek to track 2. Now, after reading two sectors, we're calling the secondary entry point (at $B403) to seek... where exactly?

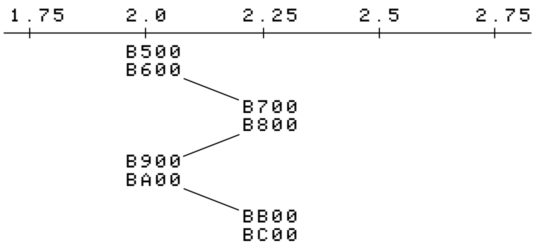
```
*B328.
B328 01 FF 01 00 00 00 00 00
```

Aha! This array is the differential to get the drive to seek forward or back. At $B200, we seeked to track 2. The first time through this loop at $B304, we read two sectors into $B500..$B6FF, then add 1 to the current phase, because `$B328 = #$01`. Normally this would seek forward a half track, to track 2.5, but because we're using the reduced timing loop, we only seek forward by a quarter track, to track 2.25.

The second time through the loop, we read two sectors into $B700..$B8FF, then subtract 1 from the phase (because `$B329 = #$FF`) and seek backwards by a quarter track. Now we're back on track 2.0.

The third time, we read two sectors from track 2.25 into $B900..$BAFF, then seek forward by a quarter track, because `$B32A = #$01`.

The fourth and final time, we read the final two sectors from track 2.25 into $BB00..$BCFF.



This explains the little “fluttering” noise the original disk makes during this phase of the boot. It’s flipping back and forth between adjacent quarter tracks, reading two sectors from each.

Boy am I glad I’m not trying to copy this disk with a generic bit copier. That would be nearly impossible, even if I knew exactly which tracks were split like this.

In Which The Floodgates Burst Open

```
*BLOAD TRACE7
.
.  [same as previous trace]
.
9780      A9 8D    LDA #$8D     interrupt the boot at $B20C
9782   8D 0D B2    STA $B20D    after it calls $B300 but before
9785      A9 97    LDA #$97     it jumps to the new code at
9787   8D 0E B2    STA $B20E    $B500

978A   4C 00 B2    JMP $B200    continue the boot

978D      A2 08    LDX #$08     (callback is here) capture the
978F      A0 00    LDY #$00     code at $B500..$BCFF so it
9791   B9 00 B5    LDA $B500,Y  survives a reboot
9794   99 00 25    STA $2500,Y
9797         C8    INY
9798      D0 F7    BNE $9791
979A   EE 93 97    INC $9793
979D   EE 96 97    INC $9796
97A0         CA    DEX
97A1      D0 EE    BNE $9791

97A3   AD E8 C0    LDA $C0E8    reboot to my work disk
97A6   4C 00 C5    JMP $C500

*BSAVE TRACE8,A$9600,L$1A9
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE
OBJ.B500-BCFF,A$2500,L$800
]CALL -151
*B500<2500.2CFFM
*B500L
B500   AE 5F 00    LDX $005F    same command ID (saved at
                                $BF76) that was "printed"
                                earlier (passed to the routine
                                at $BF6F via $FDED)

B503   BD 80 B5    LDA $B580,X  use command ID as an index
                                into this new array

B506   8D 0A B5    STA $B50A    (!)store the array value in the
                                middle of the next JSR
                                instruction

B509   20 50 B5    JSR $B550    and call it (modified based on
                                the previous lookup)

*B580.
B580 50 58 68 70 00 00 58
```

The high byte of the JSR address never changes, so depending on the command ID, we’re calling

- 00 => $B550
- 01 => $B558
- 02 => $B568
- 03 => $B570
- 06 => $B558 again

A nice, compact jump table.

```
*B550L
B550      A9 09    LDA #$09
B552      A0 00    LDY #$00
B554   4C 00 BA    JMP $BA00

*B558L
B558      A9 19    LDA #$19
B55A      A0 00    LDY #$00
B55C   20 00 BA    JSR $BA00
B55F      A9 29    LDA #$29
B561      A0 68    LDY #$68
B563   4C 00 BA    JMP $BA00

*B568L
B568      A9 31    LDA #$31
B56A      A0 00    LDY #$00
B56C   4C 00 BA    JMP $BA00

*B570L
B570      A9 41    LDA #$41
B572      A0 A0    LDY #$A0
B574   4C 00 BA    JMP $BA00
```

Those all look quite similar. Let’s see what’s at $BA00.

```
*BA00L
BA00         48    PHA          save the two input parameters
BA01      84 58    STY $58      (A & Y)

BA03   20 00 BE    JSR $BE00    seek the drive to a new phase
                                (given in A)

BA06      A2 00    LDX #$00     copy a number of bytes from
BA08      A4 58    LDY $58      $B900,Y (Y was passed in
BA0A   B9 00 B9    LDA $B900,Y  from the caller) to $BB00
BA0D   9D 00 BB    STA $BB00,X
BA10         C8    INY
BA11         E8    INX

BA12      E0 0C    CPX #$0C     $0C bytes. Always exactly
BA14      90 F4    BCC $BA0A    $0C bytes.
```

What’s at $B900? All kinds of fun[22] stuff.

---

[22] not guaranteed, actual fun may vary

```
*B900.
B900 08 09 0A 0B 0C 0D 0E 0F
B908 10 11 12 13 14 15 16 17
B910 18 19 1A 1B 1C 1D 1E 1F
B918 20 21 22 23 24 25 26 27
B920 28 29 2A 2B 2C 2D 2E 2F
B928 30 31 32 33 34 35 36 37
B930 38 39 3A 3B 3C 3D 3E 3F
B938 60 61 62 63 64 65 66 67
B940 68 69 6A 6B 6C 6D 6E 6F
B948 70 71 72 73 74 75 76 77
B950 78 79 7A 7B 7C 7D 7E 7F
B958 80 81 82 83 84 85 86 87
B960 00 00 00 00 00 00 00 00
```

That looks suspiciously like a set of high bytes for addresses in main memory. Note how it starts at `#$08` (immediately after the text page), then later jumps from `#$3F` to `#$60`, skipping over hi-res page 2.

Continuing from `$BA16`...

```
BA16  20 30 BA   JSR $BA30

*BA30L
BA30  AD 65 BF   LDA $BF65       current phase

BA33        4A   LSR             convert it to a track number
BA34     A2 03   LDX #$03

BA36     29 0F   AND #$0F        (track MOD $10)

BA38        A8   TAY             use that as the index into an
BA39  B9 10 BC   LDA $BC10,Y     array

BA3C     95 50   STA $50,X       and store it in zero page
BA3E        C8   INY
BA3F        98   TYA
BA40        CA   DEX
BA41     10 F3   BPL $BA36

*BC10.
BC10 F7 F5 EF EE DF DD D6 BE
BC18 BD BA B7 B6 AF AD AB AA
```

All of those are valid nibbles. Maybe this is setting up another rotating prologue for the next disk read routine?

Continuing from `$BA43`...

```
BA43  4C 0C BB   JMP $BB0C

*BB0CL
```

Oh joy. Another disk read routine.

```
BB0C     A2 0C   LDX #$0C        I think $54 is the sector count
BB0E     86 54   STX $54

BB10     A0 00   LDY #$00        and $55 is the logical sector
BB12  8C 54 BB   STY $BB54       number
BB15     84 55   STY $55

BB17  AE 66 BF   LDX $BF66       find a 3-nibble prologue
BB1A  BD 8C C0   LDA $C08C,X     (varies by track, set up at
BB1D     10 FB   BPL $BB1A       $BA39)
BB1F     C5 50   CMP $50
BB21     D0 F7   BNE $BB1A
BB23  BD 8C C0   LDA $C08C,X
BB26     10 FB   BPL $BB23
BB28     C5 51   CMP $51
BB2A     D0 EE   BNE $BB1A
BB2C  BD 8C C0   LDA $C08C,X
BB2F     10 FB   BPL $BB2C
BB31     C5 52   CMP $52
BB33     D0 E5   BNE $BB1A

BB35     A4 55   LDY $55         logical sector number
                                 (initialized to #$00 at $BB15)

BB37  B9 00 BB   LDA $BB00,Y     use the sector number as an
                                 index into the $0C-length
                                 page array we set up at $BA06)

BB3A  8D 55 BB   STA $BB55       and modify the upcoming
BB3D     E6 55   INC $55         code

BB3F  BC 8C C0   LDY $C08C,X     get the actual byte
BB42     10 FB   BPL $BB3F
BB44  B9 00 BC   LDA $BC00,Y
BB47        0A   ASL
BB48        0A   ASL
BB49        0A   ASL
BB4A        0A   ASL
BB4B  BC 8C C0   LDY $C08C,X
BB4E     10 FB   BPL $BB4B
BB50  19 00 BC   ORA $BC00,Y

BB53  8D 00 FF   STA $FF00       modified earlier (at $BB3A) to
BB56  EE 54 BB   INC $BB54       be the desired page in
BB59     D0 E4   BNE $BB3F       memory
BB5B  EE 55 BB   INC $BB55

BB5E  BD 8C C0   LDA $C08C,X     find a 1-nibble epilogue (also
BB61     10 FB   BPL $BB5E       varies by track)
BB63     C5 53   CMP $53
BB65     D0 A5   BNE $BB0C

BB67     C6 54   DEC $54         loop for all $0C sectors
BB69     D0 CA   BNE $BB35
BB6B        60   RTS
```

So we've read `$0C` sectors from the current track, which is the most you can fit on a track with this kind of "4-and-4" nibble encoding scheme.

Continuing from `$BA19`...

```
BA19     A5 58   LDA $58         increment the pointer to the
BA1B        18   CLC             next memory page
BA1C     69 0C   ADC #$0C
BA1E        A8   TAY

BA1F  B9 00 B9   LDA $B900,Y     if the next page is #$00,
BA22     F0 07   BEQ $BA2B       we're done

BA24        68   PLA             otherwise loop back, where
BA25        18   CLC             we'll move the drive head one
BA26     69 02   ADC #$02        full track forward and read
BA28     D0 D6   BNE $BA00       another $0C sectors

BA2B        68   PLA             execution continues here
BA2C        60   RTS             (from $BA22)
```

Now we have a whole bunch of new stuff in memory. In this case, `$B550` started on track 4.5 (`A = #$09` on entry to `$BA00`) and filled `$0800..$3FFF` and `$6000..$87FF`. If we "print" a different character, the routine at `$B500` will route through one of the other subroutines—`$B558`, `$B568`, or `$B570`. Each of them starts on a different track (`A`) and uses a different starting index (`Y`) into the page array at `$B900`. The underlying routine at `$BA00` doesn't know anything else; it just seeks and reads `$0C` sectors per track until the target page = `#$00`.

Continuing from `$B50C`...

```
B50C  20 00 B7   JSR $B700

*B700L
B700     A2 00   LDX #$00       oh joy, another decryption
B702  BD 00 B6   LDA $B600,X    loop
B705  5D 00 BE   EOR $BE00,X
B708  9D 00 03   STA $0300,X
B70B        E8   INX
B70C     E0 D0   CPX #$D0
B70E     90 F2   BCC $B702

B710  CE 13 B7   DEC $B713 (!)
B713  6D 09 B7   ADC $B709
B716        60   RTS
```

And more self-modifying code.

```
*B713:6C
*B713L
B713  6C 09 B7   JMP ($B709)
```

...which will jump to the newly decrypted code at `$0300`.

To recap: after 7 boot traces, the bootloader prints a null character via `$FD90`, which jumps to `$FDED`, which jumps to `($0036)`, which jumps to `$BF6F`, which calls `$BEB0`, which decrypts the code at `$BF9F` and returns just in time to execute it. `$BF9F` reads 3 sectors into `$B200-$B4FF`, pushes `#$00/#$3B` to the stack and exits via RTS, which returns to `$003C`, which jumps to `$B200`. `$B200` reads 8 sectors into `$B500-$BCFF` from tracks 2 and 2.5, shifting between the adjacent quarter tracks every two sectors, then jumps to `$B500`, which calls `$B5[50|58|68|70]`, which reads actual game code from multiple tracks starting at track 4.5, 9.5, 24.5, or 32.5. Then it calls `$B700`, which decrypts `$B600` into `$0300` (using `$BE00+` as the decryption key) and exits via a jump to `$0300`.

I'm sure[23] the code at `$0300` will be straightforward and easy to understand.

In Which We Go Completely Insane

The code at `$B600` is decrypted with the code at `$BE00` as the key. That was originally copied from the text page the first time, not the second time.

```
*BLOAD BOOT1 0400-07FF,A$2400
*BE00<2600.26FFM ; move key
into place
*B710:60 ; stop after loop
*B700G ; decrypt
*300L
0300     A0 00   LDY #$00       wipe almost everything we've
0302        98   TYA            already loaded at the top of
0303  99 00 B1   STA $B100,Y    main memory (!)
0306        C8   INY
0307     D0 F9   BNE $0302
0309  EE 05 03   INC $0305
030C  AE 05 03   LDX $0305

030F     E0 BD   CPX #$BD       stop at $BD00
0311     90 F0   BCC $0303
```

OK, so all we're left with in memory is the RWTS at `$BD00..$BFFF` (including the `$FDED` vector at `$BF6F`) and the single page at `$B000`. Oh, and the game, but who cares about that?

Moving on...

```
0313     A9 07   LDA #$07
0315  20 80 03   JSR $0380

*380L
0380  20 00 BE   JSR $BE00      drive seek (A = #$07, so
                                track 3.5)

0383     A2 03   LDX #$03       Pull 4 bytes from the stack,
0385        68   PLA            thus negating the JSR that
0386        CA   DEX            got us here (at $0315) and the
0387     10 FC   BPL $0385      JSR before that (at $B50C).

0389  4C 18 03   JMP $0318      continue by jumping directly
                                to the place we would have
                                returned to, if we hadn't just
                                popped the stack (which we
                                did)
```

What. The. Fahrvergnugen.

```
*318L
Oh joy.  Another disk routine.
0318  AE 66 BF   LDX $BF66

031B     A4 5F   LDY $5F        Y = command ID (a.k.a. the
                                character we "printed" way
                                back when)

031D  BD 8C C0   LDA $C08C,X    find a 3-nibble prologue ("D4
0320     10 FB   BPL $031D      D5 D7")
0322     C9 D4   CMP #$D4
0324     D0 F7   BNE $031D
0326  BD 8C C0   LDA $C08C,X
0329     10 FB   BPL $0326
032B     C9 D5   CMP #$D5
032D     D0 F3   BNE $0322
032F  BD 8C C0   LDA $C08C,X
0332     10 FB   BPL $032F
0334     C9 D7   CMP #$D7
0336     D0 F3   BNE $032B

0338        88   DEY            branch when Y goes negative
0339     30 08   BMI $0343
```

---

[23] not actually sure

```
033B  20 51 03  JSR $0351     read one byte from disk, store
                              it in $5E (not shown)

033E  20 51 03  JSR $0351     read 1 more byte from disk

0341     D0 F5  BNE $0338     loop back, unless the byte is
                              #$00
```

OK, I see it. It was hard to follow at first because the exit condition was checked before I knew it was a loop. But this is a loop. On track 3.5, there is a 3-nibble prologue (`"D4 D5 D7"`), then an array of values. Each value is two bytes. We're just finding the Nth value in the array. But to what end?

```
0343  20 51 03  JSR $0351     execution continues here
0346        48  PHA           (from $0339) read 2 more
0347  20 51 03  JSR $0351     bytes from disk and push
034A        48  PHA           them to the stack
```

Ah! A new "return" address!

Oh God. A new "return" address.

That's what this is: an array of addresses, indexed by the command ID. That's what we're looping through, and eventually pushing to the stack: the entry point for this block of the game.

But the entry point for each block is read directly from disk, so I have no idea what any of them are. Add that to the list of things I get to come back to later.

Onward. . .

```
034B  BD 88 C0  LDA $C088,X   turn off the drive motor
034E  4C 62 03  JMP $0362

*362L

0362     A0 00  LDY #$00      wipe this routine from
0364  99 00 03  STA $0300,Y   memory
0367        C8  INY
0368     C0 65  CPY #$65
036A     90 F8  BCC $0364

036C     A9 BE  LDA #$BE      push several values to the
036E        48  PHA           stack
036F     A9 AF  LDA #$AF
0371        48  PHA
0372     A9 34  LDA #$34
0374        48  PHA
0375  CE 78 03  DEC $0378 (!)
0378     29 CE  AND #$CE

More self-modifying code.
*378:28
*378L

0378        28  PLP           pop that #$34 off the stack,
0379  CE 7C 03  DEC $037C (!) but use it as status registers
037C     61 60  ADC ($60,X)   (weird, but legal—if it turns
                              out to matter, I can figure out
*37C:60                       exactly which status bits get
*37CL                         set and cleared)
037C        60  RTS
```

Now we "return" to `$BEB0` because we pushed `#$BE/#$AF/#$34` but then popped `#$34`. The routine at `$BEB0` re-encrypts the code at `$BF9F` (because now we've XOR'd it twice so it's back to its original form) and exits via RTS, which "returns" to the address we pushed to the stack at `$0346`, which we read from track 3.5—and varies based on the command we're still executing, which is really the character we "printed" via the output vector.

Which is all completely insane.

In Which We Are Restored To Sanity LOL, Just Kidding But Soon, Maybe

Since the "`JSR $B700`" at `$B50C` never returns (because of the crazy stack manipulation at `$0383`), that's the last chance I'll get to interrupt the boot and capture this chunk of game code in memory. I won't know what the entry point is (because it's read from disk), but one thing at a time.

```
*BLOAD TRACE8
.
.  [same as previous trace]
.
978D     A9 4C  LDA #$4C      unconditionally break after
978F  8D 0C B5  STA $B50C     loading the game code into
9792     A9 59  LDA #$59      main memory
9794  8D 0D B5  STA $B50D
9797     A9 FF  LDA #$FF
9799  8D 0E B5  STA $B50E


979C  4C 00 B5  JMP $B500     continue the boot

*BSAVE TRACE9,A$9600,L$19F
*9600G
...reboots slot 6...
...read read read...
<beep>
Success!
*C050 C054 C057 C052
[displays a very nice picture
of a
 gumball machine which is
featured in
 the game's introduction
sequence]
*C051
```

OK, let's save it. According to the table at `$B900`, we filled `$0800..$3FFF` and `$6000..$87FF`. `$0800+` is overwritten on reboot by the boot sector and later by the HELLO program on my work disk. `$8000+` is also overwritten by Diversi-DOS 64K, which is annoying but not insurmountable. So I'll save this in pieces.

```
*C500G
...
]BSAVE BLOCK
00.2000-3FFF,A$2000,L$2000
]BRUN TRACE9
...reboots slot 6...
<beep>
*2800<800.1FFFM
*C500G
...
]BSAVE BLOCK
00.0800-1FFF,A$2800,L$1800
]BRUN TRACE9
...reboots slot 6...
<beep>
*2000<6000.87FFM
*C500G
...
]BSAVE BLOCK
00.6000-87FF,A$2000,L$2800
```

Now what? Well this is only the first chunk of game code, loaded by printing a null character. By setting up another trace and changing the value of zero page `$5F`, I can route `$B500` through a different subroutine at `$B558` or `$B568` or `$B570` and load a different chunk of game code.

```
]CALL -151
*BLOAD OBJ.B500-BCFF,A$B500
According to the lookup table
at $B580,
$B500 routed through $B558 to
load the
game code.  Here is that
routine:
*B558L
B558      A9 19    LDA #$19
B55A      A0 00    LDY #$00
B55C   20 00 BA    JSR $BA00
B55F      A9 29    LDA #$29
B561      A0 68    LDY #$68
B563   4C 00 BA    JMP $BA00
```

The first call to `$BA00` will fill up the same parts of memory as we filled when the character (in `$5F`) was `#$00`—`$0800..$3FFF` and `$6000..$87FF`. But it starts reading from disk at phase `$19` (track `$0C 1/2`), so it's a completely different chunk of code.

The second call to `$BA00` starts reading at phase `$29` (track `$14 1/2`), and it looks at `$B900 + Y = $B968` to get the list of pages to fill in memory.

```
*B968.
B968 88 89 8A 8B 8C 8D 8E 8F
B970 90 91 92 93 94 95 96 97
B978 98 99 9A 9B 9C 9D 9E 9F
B980 A0 A1 A2 A3 A4 A5 A6 A7
B988 A8 A9 AA AB AC AD AE AF
B990 B2 B2 B2 B2 B2 B2 B2 B2
B998 00 00 00 00 00 00 00 00
```

The first call to `$BA00` stopped just shy of `$8800`, and that's exactly where we pick up in the second call. I'm guessing that `$B200` isn't really used, but the track read routine at `$BA00` is "dumb" in that it always reads exactly `$0C` sectors from each track. So we're filling up `$8800..$AFFF`, then reading the rest of the last track into `$B200` over and over.

Let's capture it.

```
*BLOAD TRACE9
.
.  [same as previous trace]
.

978D      A9 4C    LDA #$4C      again, break to the monitor at
978F   8D 0C B5    STA $B50C     $B50C instead of continuing to
9792      A9 59    LDA #$59      $B700
9794   8D 0D B5    STA $B50D
9797      A9 FF    LDA #$FF
9799   8D 0E B5    STA $B50E


979C      A9 01    LDA #$01      change the character being
979E      85 5F    STA $5F       "printed" to #$01 just before
                                 the bootloader uses it to load
                                 the appropriate chunk of
                                 game code


97A0   4C 00 B5    JMP $B500     continue the boot

*BSAVE TRACE10,A$9600,L$1A3
*9600G
...reboots slot 6...
...read read read...
<beep>
*C050 C054 C057 C052
[displays a very nice picture
of the
 main game screen]
*C051
*C500G
...
]BSAVE BLOCK
01.2000-3FFF,A$2000,L$2000
]BRUN TRACE10
...reboots slot 6...
<beep>
*2800<800.1FFFM
*C500G
...
]BSAVE BLOCK
01.0800-1FFF,A$2800,L$1800
]BRUN TRACE9
...reboots slot 6...
<beep>
*2000<6000.AFFFM
*C500G
...
]BSAVE BLOCK
01.6000-AFFF,A$2000,L$5000
```

And similarly with blocks 2 and 3. (These are not shown here, but you can look at TRACE11 and TRACE12 on my work disk.) Blocks 4 and 5 get special-cased earlier (at `$BF86` and `$BF8D`, respectively), so they never reach `$B500` to load anything from disk. Block 6 is the same as block 1.

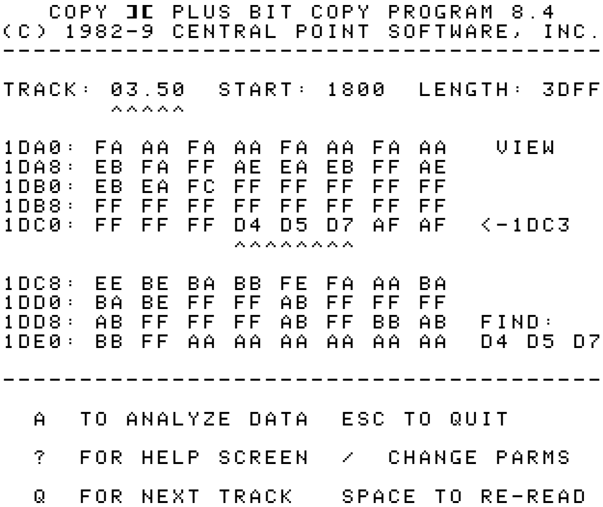
That's it. I've captured all the game code. Here's what the "game" looks like at this point:

```
]CATALOG
C1983 DSR^C#254
019 FREE
 A 002 HELLO
 B 003 BOOT0
*B 003 TRACE
 B 003 BOOT1 0300-03FF
*B 003 TRACE2
 B 003 BOOT1 0100-01FF
*B 003 TRACE3
 B 006 BOOT1 0400-07FF
*B 003 TRACE4
 B 005 BOOT2 0500-07FF
*B 003 TRACE5
 B 003 BOOT2 B000-B0FF
 B 003 BOOT2 0100-01FF
*B 003 TRACE6
 B 003 BOOT3 0000-00FF
*B 003 TRACE7
 B 005 OBJ.B200-B4FF
*B 003 TRACE8
 B 010 OBJ.B500-BCFF
*B 003 TRACE9
 B 026 BLOCK 00.0800-1FFF
 B 034 BLOCK 00.2000-3FFF
 B 042 BLOCK 00.6000-87FF
*B 003 TRACE10
 B 026 BLOCK 01.0800-1FFF
 B 034 BLOCK 01.2000-3FFF
 B 082 BLOCK 01.6000-AFFF
*B 003 TRACE11
 B 026 BLOCK 02.0800-1FFF
 B 034 BLOCK 02.2000-3FFF
 B 042 BLOCK 02.6000-87FF
*B 003 TRACE12
 B 034 BLOCK 03.2000-3FFF
```

It's... it's beautiful. *wipes tear*

In Which Every Exit Is An Entrance Somewhere Else

I've captured all the blocks of the game code (I think), but I still have no idea how to run it. The entry points for each block are read directly from disk, in the loop at `$031D`.



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

TRACK: 03.50  START: 1800  LENGTH: 3DFF
       ^^^^^

1DA0: FA AA FA AA FA AA FA AA   VIEW
1DA8: EB FA FF AE EA EB FF AE
1DB0: EB EA FC FF FF FF FF FF
1DB8: FF FF FF FF FF FF FF FF
1DC0: FF FF FF D4 D5 D7 AF AF  <-1DC3
               ^^^^^^^^

1DC8: EE BE BA BB FE FA AA BA
1DD0: BA BE FF FF AB FF FF FF
1DD8: AB FF FF FF AB FF BB AB  FIND:
1DE0: BB FF AA AA AA AA AA AA  D4 D5 D7
---------------------------------------

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ
```

Rather than try to boot-trace every possible block, I'm going to load up the original disk in a nibble editor and do the calculations myself. The array of entry points is on track 3.5. Firing up Copy II Plus nibble editor, I searched for the same 3-nibble prologue ("D4 D5 D7") that the code at `$031D` searches for, and lo and behold!

After the "D4 D5 D7" prologue, I find an array of 4-and-4-encoded nibbles starting at offset `$1DC6`. Breaking them down into pairs and decoding them with the 4-4 encoding scheme, I get this list of bytes:

nibbles	byte
AF AF	#$0F
EE BE	#$9C
BA BB	#$31
FE FA	#$F8
AA BA	#$10
BA BE	#$34
FF FF	#$FF
AB FF	#$57
FF FF	#$FF
AB FF	#$57
FF FF	#$FF
AB FF	#$57
BB AB	#$23
BB FF	#$77

And now—maybe!—I have my list of entry points for each block of the game code.

```
Only one way to know for
sure...
]PR#5
...
]CALL -151
*800:0 N 801<800.BEFEM           clear main memory so I'm not
                                 accidentally relying on
                                 random stuff left over from all
                                 my other testing

*BLOAD BLOCK                     load all of block 0 into place
00.0800-1FFF,A$800
*BLOAD BLOCK
00.2000-3FFF,A$2000
*BLOAD BLOCK
00.6000-87FF,A$6000

*F9DG                            jump to the entry point I
[displays the game intro         found on track 3.5 (+1, since
sequence]                        the original code pushes it to
*does a little happy dance in    the stack and "returns" to it)
my chair*
```

We have no further use for the original disk. Now would be an excellent time to take it out of the drive and store it in a cool, dry place.

In Which Two Wrongs Don't Make A—Oh God I Can't Even—With This Pun

Remember when I said I'd look at `$BD00` later? The time has come. Later is now.

The output vector at `$BF6F` has special case handling if `A = #$04`. Instead of continuing to `$0300` and `$B500`, it jumps directly to `$BD00`. What's so special about `$BD00`?

The code at `$BD00` was moved there very early in the boot process, from page `$0500` on the text screen. (The first time we loaded code into the text screen, not the second time.) So it's in "BOOT1 0400-07FF" on my work disk.

```
]PR#5
...
]BLOAD BOOT1 0400-07FF,A$2400
]CALL -151
*BD00<2500.25FFM
*BD00L
BD00  AE 66 BF   LDX $BF66     turn on drive motor
BD03  BD 89 C0   LDA $C089,X

BD06     A9 64   LDA #$64      wait for drive to settle
BD08  20 A8 FC   JSR $FCA8

BD0B     A9 10   LDA #$10      seek to phase $10 (track 8)
BD0D  20 00 BE   JSR $BE00

BD10     A9 02   LDA #$02      seek to phase $02 (track 1)
BD12  20 00 BE   JSR $BE00

BD15     A0 FF   LDY #$FF      initialize data latches
BD17  BD 8D C0   LDA $C08D,X
BD1A  BD 8E C0   LDA $C08E,X
BD1D  9D 8F C0   STA $C08F,X
BD20  1D 8C C0   ORA $C08C,X

BD23     A9 80   LDA #$80      wait
BD25  20 A8 FC   JSR $FCA8
BD28  20 A8 FC   JSR $FCA8

BD2B  BD 8D C0   LDA $C08D,X   Oh God
BD2E  BD 8E C0   LDA $C08E,X
BD31        98   TYA
BD32  9D 8F C0   STA $C08F,X
BD35  1D 8C C0   ORA $C08C,X
BD38        48   PHA
BD39        68   PLA
BD3A     C1 00   CMP ($00,X)
BD3C     C1 00   CMP ($00,X)
BD3E        EA   NOP
BD3F        C8   INY

BD40  9D 8D C0   STA $C08D,X   Oh God
BD43  1D 8C C0   ORA $C08C,X
BD46  B9 8F BD   LDA $BD8F,Y
BD49     D0 EF   BNE $BD3A
BD4B        A8   TAY
BD4C        EA   NOP
BD4D        EA   NOP

BD4E  B9 00 B0   LDA $B000,Y   ← !
BD51        48   PHA
BD52        4A   LSR
BD53     09 AA   ORA #$AA
BD55  9D 8D C0   STA $C08D,X   Oh God Oh God Oh God
BD58  DD 8C C0   CMP $C08C,X
BD5B     C1 00   CMP ($00,X)
BD5D        EA   NOP
BD5E        EA   NOP
BD5F        48   PHA
BD60        68   PLA
BD61        68   PLA
BD62     09 AA   ORA #$AA
BD64  9D 8D C0   STA $C08D,X
BD67  DD 8C C0   CMP $C08C,X
BD6A        48   PHA
BD6B        68   PLA
BD6C        C8   INY
BD6D     D0 DF   BNE $BD4E
BD6F     A9 D5   LDA #$D5
BD71     C1 00   CMP ($00,X)
BD73        EA   NOP
BD74        EA   NOP
BD75  9D 8D C0   STA $C08D,X
BD78  1D 8C C0   ORA $C08C,X
BD7B     A9 08   LDA #$08
BD7D  20 A8 FC   JSR $FCA8
BD80  BD 8E C0   LDA $C08E,X
BD83  BD 8C C0   LDA $C08C,X

BD86     A9 07   LDA #$07      seek back to track 3.5
BD88  20 00 BE   JSR $BE00

BD8B  BD 88 C0   LDA $C088,X   turn off drive motor and exit
BD8E        60   RTS           gracefully
```

This is a disk write routine. It's taking the data at `$B000` (that mystery sector that was loaded even earlier in the boot) and writing it to track 1.

Because high scores.

That's what's at `$B000`. High scores. [Edit from the future: also some persistent joystick options.]

Why is this so distressing? Because it means I'll get to include a full read/write RWTS on my crack (which I haven't even starting building yet, but soon!) so it can save high scores like the original game. Because anything less is obviously unacceptable.

The Right Ones In The Right Order

Let's step back from the low-level code for a moment and talk about how this game interacts with the disk at a high level.

- There is no runtime protection check. All the "protection" is structural—data is stored on whole tracks, half tracks, and even some consecutive quarter tracks. Once the game code is in memory, there are no nibble checks or secondary protections.
- The game code itself contains no disk code. They're completely isolated. I proved this by loading the game code from my work disk and

jumping to the entry point. (I tested the animated introduction, but you can also run the game itself by loading the block `$01` files into memory and jumping to `$31F9`. The game runs until you finish the level and it tries to load the first cut scene from disk.)

- The game code communicates with the disk subsystem through the output vector, i.e. by printing `#$00..#$06` to `$FDED`. The disk code handles filling the screen with a pseudo-random color, reading the right chunks from the right places on disk and putting them into the right places in memory, then jumping to the right address to continue. (In the case of printing `#$04`, it handles writing the right data in memory to the right place on disk.)
- Game code lives at `$0800..$AFFF`, zero page, and one page at `$B000` for high scores. The disk subsystem clobbers the text screen at `$0400` using lo-res graphics for the color fills. All memory above `$B100` is available; in fact, most of it is wiped (at `$0300`) after every disk command.

This is great news. It gives us total flexibility to recreate the game from its constituent pieces.

A Man, A Plan, A Canal, *&c.*

Here's the plan:

1. Write the game code to a standard 16-sector disk
2. Write a bootloader and RWTS that can read the game code into memory
3. Write some glue code to mimic the original output vector at `$BF6F` (A = command ID from `#$00-#$06`, all other values actually print) so I don't need to change any game code
4. Declare victory[24]

Looking at the length of each block and dividing by 16, I can space everything out on separate tracks and still have plenty of room. This means each block can start on its own track, which saves a few bytes by being able to hard-code the starting sector for each block.

The disk map will look like this:

tr	memory range	notes
00	\$BD00..\$BFFF	Gumboot
01	\$B000..\$B3FF	scores/zpage/glue
02	\$0800..\$17FF	block 0
03	\$1800..\$27FF	block 0
04	\$2800..\$37FF	block 0
05	\$3800..\$3FFF	block 0
06	\$6000..\$67FF	block 0
07	\$6800..\$77FF	block 0
08	\$7000..\$87FF	block 0
09	\$0800..\$17FF	block 1
0A	\$1800..\$27FF	block 1
0B	\$2800..\$37FF	block 1
0C	\$3800..\$3FFF	block 1
0D	\$6000..\$6FFF	block 1
0E	\$7000..\$7FFF	block 1
0F	\$8000..\$8FFF	block 1
10	\$9000..\$9FFF	block 1
11	\$A000..\$AFFF	block 1
12	\$0800..\$17FF	block 2
13	\$1800..\$27FF	block 2
14	\$2800..\$37FF	block 2
15	\$3800..\$3FFF	block 2
16	\$6000..\$6FFF	block 2
17	\$7000..\$7FFF	block 2
18	\$8000..\$87FF	block 2
19	\$2000..\$2FFF	block 3
1A	\$3000..\$3FFF	block 3

I wrote a build script to take all the chunks of game code I captured way back on page 43. And by "script", I mean "BASIC program."

```
]PR#5
...
 10 REM MAKE GUMBALL
 11 REM S6,D1=BLANK DISK
 12 REM S5,D1=WORK DISK
 20 D$ = CHR$ (4)
```

Load the first part of block 0:

```
 30 PRINT D$"BLOAD BLOCK
00.0800-1FFF,
 A$1000"
 40 PRINT D$"BLOAD BLOCK
00.2000-3FFF,
 A$2800"
```

Write it to tracks \$02-\$05:

```
 50 PAGE = 16:COUNT = 56:TRK =
2:
 SEC = 0:  GOSUB 1000
```

Load the second part of block 0:

```
 60 PRINT D$"BLOAD BLOCK
00.6000-87FF,
 A$6000"
```

Write it to tracks \$06-\$08:

```
 70 PAGE = 96:COUNT = 40:TRK =
6:
 SEC = 0:  GOSUB 1000
```

---

[24] take a nap

And so on, for all the other blocks:

```
 80 PRINT D$"BLOAD BLOCK
01.0800-1FFF,
 A$1000"
 90 PRINT D$"BLOAD BLOCK
01.2000-3FFF,
 A$2800"
 100 PAGE = 16:COUNT = 56:TRK
= 9:
 SEC = 0:  GOSUB 1000
 110 PRINT D$"BLOAD BLOCK
01.6000-AFFF,
 A$6000"
 120 PAGE = 96:COUNT = 80:TRK
= 13:
 SEC = 0:  GOSUB 1000
 130 PRINT D$"BLOAD BLOCK
02.0800-1FFF,
 A$1000"
 140 PRINT D$"BLOAD BLOCK
02.2000-3FFF,
 A$2800"
 150 PAGE = 16:COUNT = 56:TRK
= 18:
 SEC = 0:  GOSUB 1000
 160 PRINT D$"BLOAD BLOCK
02.6000-87FF,
 A$6000"
 170 PAGE = 96:COUNT = 40:TRK
= 22:
 SEC = 0:  GOSUB 1000
 180 PRINT D$"BLOAD BLOCK
03.2000-3FFF,
 A$2000"
 190 PAGE = 32:COUNT = 32:TRK
= 25:
 SEC = 0:  GOSUB 1000
 200 PRINT D$"BLOAD BOOT2
0500-07FF,
 A$2500"
 210 PAGE = 39:COUNT = 1:TRK =
1:
 SEC = 0:  GOSUB 1000
 220 PRINT D$"BLOAD BOOT3
0000-00FF,
 A$1000"
 230 POKE 4150,0:  POKE
4151,178:  REM
 SET ($36) TO $B200
 240 PAGE = 16:COUNT = 1:TRK =
1:
 SEC = 7:  GOSUB 1000
 999 END
 1000 REM WRITE TO DISK
 1010 PRINT D$"BLOAD WRITE"
 1020 POKE 908,TRK
 1030 POKE 909,SEC
 1040 POKE 913,PAGE
 1050 POKE 769,COUNT
 1060 CALL 768
 1070 RETURN
]SAVE MAKE
```

The BASIC program relies on a short assembly language routine to do the actual writing to disk. Here is that routine (loaded on line 1010):

```
]CALL -151

0300      A9 D1   LDA #$D1 ☺    page count (set from BASIC)
0302      85 FF   STA $FF

0304      A9 00   LDA #$00      logical sector (incremented)
0306      85 FE   STA $FE

0308      A9 03   LDA #$03      call RWTS to write sector
030A      A0 88   LDY #$88
030C   20 D9 03   JSR $03D9

030F      E6 FE   INC $FE       increment logical sector, wrap
0311      A4 FE   LDY $FE       around from $0F to $00 and
0313      C0 10   CPY #$10      increment track
0315      D0 07   BNE $031E
0317      A0 00   LDY #$00
0319      84 FE   STY $FE
031B   EE 8C 03   INC $038C

031E   B9 40 03   LDA $0340,Y   convert logical to physical
0321   8D 8D 03   STA $038D     sector

0324   EE 91 03   INC $0391     increment page to write

0327      C6 FF   DEC $FF       loop until done with all
0329      D0 DD   BNE $0308     sectors
032B         60   RTS

*340.34F

0340 00 07 0E 06 0D 05 0C 04    logical to physical sector
0348 0B 03 0A 02 09 01 08 0F    mapping
*388.397

0388 01 60 01 00  D1 D1  FB F7
                  \___/
              track/sector
            (set from BASIC)

0390 00  D1  00 00 02 00 00 60
          ↑
       address
   (set from BASIC)
                                RWTS parameter table,
                                pre-initialized with slot
                                (#$06), drive (#$01), and
                                RWTS write command (#$02)

*BSAVE WRITE,A$300,L$98
[S6,D1=blank disk]
]RUN MAKE
```

…write write write…

Boom! The entire game is on tracks `$02-$1A` of a standard 16-sector disk.

Now we get to write an RWTS.

Introducing Gumboot

Gumboot is a fast bootloader and full read/write RWTS. It fits in 4 sectors on track 0, including a boot sector. It uses only 6 pages of memory for all its code + data + scratch space. It uses no zero page addresses after boot. It can start the game from a cold boot in 3 seconds. That's twice as fast as the original disk.

GUMBOOT

qkumba wrote it from scratch, because of course he did. I, um, mostly just cheered.

After boot-time initialization, Gumboot is dead simple and always ready to use:

entry	command	parameters
$BD00	read	A = first track Y = first page X = sector count
$BE00	write	A = sector Y = page
$BF00	seek	A = track

That's it. It's so small, there's `$80` unused bytes at `$BF80`. You could fit a cute message in there! (We didn't.)

Some important notes:

- The read routine reads consecutive tracks in physical sector order into consecutive pages in memory. There is no translation from physical to logical sectors.
- The write routine writes one sector, and also assumes a physical sector number.
- The seek routine can seek forward or back to any whole track. (I mention this because some fastloaders can only seek forward.)

I said Gumboot takes 6 pages in memory, but I've only mentioned 3. The other 3 are for data:

`$BA00..$BB55` scratch space for write (technically available as long as you don't mind them being clobbered during disk write)

`$BB00..$BCFF` data tables (initialized once during boot)

Gumboot Boot0

Gumboot starts, as all disks start, on track `$00`. Sector `$00` (boot0) reuses the disk controller ROM routine to read sector `$0E`, `$0D`, and `$0C` (boot1). Boot0 creates a few data tables, modifies the boot1 code to accommodate booting from any slot, and jumps to it.

Boot0 is loaded at `$0800` by the disk controller ROM routine.

```
0800 [01]                      tell the ROM to load only
                               this sector (we'll do the rest
                               manually)

0801        4A    LSR          The accumulator is #$01 after
                               loading sector $00, #$03 after
                               loading sector $0E, #$05 after
                               loading sector $0D, and #$07
                               after loading sector $0C. We
                               shift it right to divide by 2,
                               then use that to calculate the
                               load address of the next
                               sector.

0802     69 BC    ADC #$BC     Sector $0E → $BD00
                               Sector $0D → $BE00
                               Sector $0C → $BF00

0804     85 27    STA $27      store the load address

0806        0A    ASL          shift the accumulator again
0807        0A    ASL          (now that we've stored the
                               load address)

0808        8A    TXA          transfer X (boot slot x16) to
                               the accumulator, which will
                               be useful later but doesn't
                               affect the carry flag we may
                               have just tripped with the
                               two "ASL" instructions

0809     B0 0D    BCS $0818    if the two "ASL" instructions
                               set the carry flag, it means
                               the load address was at least
                               #$C0, which means we've
                               loaded all the sectors we
                               wanted to load and we should
                               exit this loop

080B     E6 3D    INC $3D      Set up next sector number to
                               read. The disk controller
                               ROM does this once already,
                               but due to quirks of timing,
                               it's much faster to increment
                               it twice so the next sector you
                               want to load is actually the
                               next sector under the drive
                               head. Otherwise you end up
                               waiting for the disk to spin an
                               entire revolution, which is
                               quite slow.

080D        4A    LSR          Set up the "return" address to
080E        4A    LSR          jump to the "read sector"
080F        4A    LSR          entry point of the disk
0810        4A    LSR          controller ROM. This could
0811     09 C0    ORA #$C0     be anywhere in $Cx00
                               depending on the slot we
                               booted from, which is why we
                               put the boot slot in the
                               accumulator at $0808.
```

```
0813       48   PHA          push the entry point on the
0814    A9 5B   LDA #$5B     stack
0816       48   PHA

0817       60   RTS          "Return" to the entry point
                             via RTS. The disk controller
                             ROM always jumps to $0801
                             (remember, that's why we
                             had to move it and patch it to
                             trace the boot all the way
                             back on page 25), so this
                             entire thing is a loop that
                             only exits via the "BCS"
                             branch at $0809.

0818    09 8C   ORA #$8C     Execution continues here
081A    A2 00   LDX #$00     (from $0809) after three
081C BC AF 08   LDY $08AF,X  sectors have been loaded into
081F    84 26   STY $26      memory at $BD00..$BFFF.
0821 BC B0 08   LDY $08B0,X  There are a number of places
0824    F0 0A   BEQ $0830    in boot1 that hit a
0826    84 27   STY $27      slot-specific soft switch (read
0828    A0 00   LDY #$00     a nibble from disk, turn off
082A    91 26   STA ($26),Y  the drive, &c.). Rather than
082C       E8   INX          the usual form of "LDA
082D       E8   INX          $C08C,X", we will use "LDA
082E    D0 EC   BNE $081C    $C0EC" and modify the $EC
                             byte in advance, based on the
                             boot slot. $08A4 is an array of
                             all the places in the Gumboot
                             code that get this adjustment.

0830    29 F8   AND #$F8     munge $EC → $E8 (used later
0832 8D FC BD   STA $BDFC    to turn off the drive motor)

0835    09 01   ORA #$01     munge $E8 → $E9 (used later
0837 8D 0B BD   STA $BD0B    to turn on the drive motor)
083A 8D 07 BE   STA $BE07

083D    49 09   EOR #$09     munge $E9 → $E0 (used later
083F 8D 54 BF   STA $BF54    to move the drive head via
                             the stepper motor)

0842    29 70   AND #$70     munge $E0 → $60 (boot slot
0844 8D 37 BE   STA $BE37    x16, used during seek and
0847 8D 69 BE   STA $BE69    write routines)
084A 8D 7F BE   STA $BE7F
084D 8D AC BE   STA $BEAC
```

6 + 2

Before I dive into the next chunk of code, I get to pause and explain a little bit of theory. As you probably know if you're the sort of person who's read this far already, 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 is 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 such that `%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.

Gumboot also uses “6-and-2” encoding. The first `$56` nibbles in the data field are still split into pairs of bits that will 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 Gumboot gets to the data field checksum, it has already stored all `$100` 8-bit bytes in their final resting place in memory. This means that we can read all 16 sectors on a track in one revolution of the disk. That’s what makes it crazy fast.

To make it possible to twiddle the bits and not miss nibbles as the disk spins[25], we do some of the work in advance. We multiply each of the 64 possible decoded values by 4 and store those values. (Since this is done 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 bit math 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 first table, at `$BC00..$BCFF`, 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. 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 `$BB96..$BBFF`, is the “pre-shift” table. This contains all the possible 6-bit bytes, in order, each multiplied by 4. (They are 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.

---

[25] 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.

addr	raw	decoded 6-bit	pre-shift
$BB96	$96	0 = %00000000	%00000000
$BB97	$97	1 = %00000001	%00000100
$BB98	$98	[invalid raw nibble]	
$BB99	$99	[invalid raw nibble]	
$BB9A	$9A	2 = %00000010	%00001000
$BB9B	$9B	3 = %00000011	%00001100
$BB9C	$9C	[invalid raw nibble]	
$BB9D	$9D	4 = %00000100	%00010000
.			
.			
.			
$BBFE	$FE	62 = %00111110	%11111000
$BBFF	$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. This wasn't an accident; I mean, that sort of magic doesn't just happen. But the table of 2-bit bytes is arranged in such a way that we can take one of the raw nibbles to be decoded and split apart (from the first `$56` raw nibbles in the data field), use each 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.

Back to Gumboot

This is the loop that creates the pre-shift table at `$BB96`. As a special bonus, it also creates the inverse table that is used during disk write operations, converting in the other direction.

```
0850      A2 3F    LDX #$3F
0852      86 FF    STX $FF
0854         E8    INX
0855      A0 7F    LDY #$7F
0857      84 FE    STY $FE
0859         98    TYA
085A         0A    ASL
085B      24 FE    BIT $FE
085D      F0 18    BEQ $0877
085F      05 FE    ORA $FE
0861      49 FF    EOR #$FF
0863      29 7E    AND #$7E
0865      B0 10    BCS $0877
0867         4A    LSR
0868      D0 FB    BNE $0865
086A         CA    DEX
086B         8A    TXA
086C         0A    ASL
086D         0A    ASL
086E   99 80 BB    STA $BB80,Y
0871         98    TYA
0872      09 80    ORA #$80
0874   9D 56 BB    STA $BB56,X
0877         88    DEY
0878      D0 DD    BNE $0857
```

And this is the result, where "`..`" means that the address is uninitialized and unused.

```
BB90                   00 04
BB98 .. .. 08 0C .. 10 14 18
BBA0 .. .. .. .. .. .. 1C 20
BBA8 .. .. .. 24 28 2C 30 34
BBB0 .. .. 38 3C 40 44 48 4C
BBB8 .. 50 54 58 5C 60 64 68
BBC0 .. .. .. .. .. .. .. ..
BBC8 .. .. .. 6C .. 70 74 78
BBD0 .. .. .. 7C .. .. 80 84
BBD8 .. 88 8C 90 94 98 9C A0
BBE0 .. .. .. .. .. A4 A8 AC
BBE8 .. B0 B4 B8 BC C0 C4 C8
BBF0 .. .. CC D0 D4 D8 DC E0
BBF8 .. E4 E8 EC F0 F4 F8 FC
```

Next up: a loop to create the table of 2-bit values at `$BC00`, magically arranged to enable easy lookups later.

```
087A      84 FD    STY $FD
087C      46 FF    LSR $FF
087E      46 FF    LSR $FF
0880   BD BD 08    LDA $08BD,X
0883   99 00 BC    STA $BC00,Y
0886      E6 FD    INC $FD
0888      A5 FD    LDA $FD
088A      25 FF    AND $FF
088C      D0 05    BNE $0893
088E         E8    INX
088F         8A    TXA
0890      29 03    AND #$03
0892         AA    TAX
0893         C8    INY
0894         C8    INY
0895         C8    INY
0896         C8    INY
0897      C0 03    CPY #$03
0899      B0 E5    BCS $0880
089B         C8    INY
089C      C0 03    CPY #$03
089E      90 DC    BCC $087C
```

And this is the result:

```
BC00 00 00 00 .. 00 00 02 ..
BC08 00 00 01 .. 00 00 03 ..
BC10 00 02 00 .. 00 02 02 ..
BC18 00 02 01 .. 00 02 03 ..
BC20 00 01 00 .. 00 01 02 ..
BC28 00 01 01 .. 00 01 03 ..
BC30 00 03 00 .. 00 03 02 ..
BC38 00 03 01 .. 00 03 03 ..
BC40 02 00 00 .. 02 00 02 ..
BC48 02 00 01 .. 02 00 03 ..
BC50 02 02 00 .. 02 02 02 ..
BC58 02 02 01 .. 02 02 03 ..
BC60 02 01 00 .. 02 01 02 ..
BC68 02 01 01 .. 02 01 03 ..
BC70 02 03 00 .. 02 03 02 ..
BC78 02 03 01 .. 02 03 03 ..
BC80 01 00 00 .. 01 00 02 ..
BC88 01 00 01 .. 01 00 03 ..
BC90 01 02 00 .. 01 02 02 ..
BC98 01 02 01 .. 01 02 03 ..
BCA0 01 01 00 .. 01 01 02 ..
BCA8 01 01 01 .. 01 01 03 ..
BCB0 01 03 00 .. 01 03 02 ..
BCB8 01 03 01 .. 01 03 03 ..
BCC0 03 00 00 .. 03 00 02 ..
BCC8 03 00 01 .. 03 00 03 ..
BCD0 03 02 00 .. 03 02 02 ..
BCD8 03 02 01 .. 03 02 03 ..
BCE0 03 01 00 .. 03 01 02 ..
BCE8 03 01 01 .. 03 01 03 ..
BCF0 03 03 00 .. 03 03 02 ..
BCF8 03 03 01 .. 03 03 03 ..
```

And with that, Gumboot is fully armed and operational.

```
08A0      A9 B2   LDA #$B2     Push a "return" address on
08A2         48   PHA          the stack. We'll come back to
08A3      A9 F0   LDA #$F0     this later. (Ha ha, get it,
08A5         48   PHA          come back to it? OK, let's
                               pretend that never happened.)

08A6      A9 01   LDA #$01     Set up an initial read of 3
08A8      A2 03   LDX #$03     sectors from track 1 into
08AA      A0 B0   LDY #$B0     $B000..$B2FF. This contains
                               the high scores data, zero
                               page, and a new output vector
                               that interfaces with Gumboot.
08AC   4C 00 BD   JMP $BD00    Read all that from disk and
                               exit via the "return" address
                               we just pushed on the stack
                               at $0895.
```

Execution will continue at `$B2F1`, once we read that from disk. `$B2F1` is new code I wrote, and I promise to show it to you. But first, I get to finish showing you how the disk read routine works.

Read & Go Seek

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 boot from any slot. Gumboot also supports booting and reading from any slot, but instead of using an index, most fetch instructions are set up in advance 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. (We take full advantage of this freedom.) I've marked each pre-set softswitch with ☺.

There are several other instances of addresses and constants that get modified while Gumboot is executing. I've left these with a bogus value `$D1` and marked them with ☺.

Gumboot's source code should be available from the same place you found this write-up. If you're looking to modify this code for your own purposes, I suggest you "use the source, Luke."

```
*BD00L
BD00         0A   ASL            A = the track number to seek
BD01   8D 10 BF   STA $BF10      to. We multiply it by 2 to
                                 convert it to a phase, then
                                 store it inside the seek routine
                                 which we will call shortly.

BD04   8E EF BD   STX $BDEF      X = the number of sectors to
                                 read

BD07   8C 24 BD   STY $BD24      Y = the starting address in
                                 memory

BD0A   AD E9 C0   LDA $C0E9 ☺    turn on the drive motor


BD0D   20 75 BF   JSR $BF75      poll for real nibbles (#$FF
                                 followed by non-#$FF) as a
                                 way to ensure the drive has
                                 spun up fully

BD10      A9 10   LDA #$10       are we reading this entire
BD12   CD EF BD   CMP $BDEF      track?

BD15      B0 01   BCS $BD18      yes -> branch

BD17         AA   TAX            no
BD18   8E 94 BF   STX $BF94

BD1B   20 04 BF   JSR $BF04      seek to the track we want

BD1E   AE 94 BF   LDX $BF94      Initialize an array of which
BD21      A0 00   LDY #$00       sectors we've read from the
BD23      A9 D1   LDA #$D1 ☺     current track. The array is in
BD25   99 84 BF   STA $BF84,Y    physical sector order, thus the
BD28   EE 24 BD   INC $BD24      RWTS assumes data is stored
BD2B         C8   INY            in physical sector order on
BD2C         CA   DEX            each track. (This saves 18
BD2D      D0 F4   BNE $BD23      bytes: 16 for the table and 2
                                 for the lookup command!)
BD2F   20 D5 BE   JSR $BED5      Values are the actual pages in
                                 memory where that sector
*BED5L                           should go, and they get
                                 zeroed once the sector is read
                                 (so we don't waste time
                                 decoding the same sector
                                 twice).

BED5   20 E4 BE   JSR $BEE4      This routine reads nibbles
BED8      C9 D5   CMP #$D5       from disk until it finds the
BEDA      D0 F9   BNE $BED5      sequence "D5 AA", then it
BEDC   20 E4 BE   JSR $BEE4      reads one more nibble and
BEDF      C9 AA   CMP #$AA       returns it in the accumulator.
BEE1      D0 F5   BNE $BED8      We reuse this routine to find
BEE3         A8   TAY            both the address and data
BEE4   AD EC C0   LDA $C0EC ☺    field prologues.
BEE7      10 FB   BPL $BEE4
BEE9         60   RTS
```

Continuing from `$BD32`…

BD32	49 AD	EOR #$AD	If that third nibble is not #$AD, we assume it's the end of the address prologue.
BD34	F0 35	BEQ $BD6B	
BD36	20 C2 BE	JSR $BEC2	(#$96 would be the third nibble of a standard address prologue, but we don't actually check.) We fall through and start decoding the 4-4 encoded values in the address field.

```
*BEC2L
```

BEC2	A0 03	LDY #$03	This routine parses the 4-4-encoded values in the address field. The first time through this loop, we'll read the disk volume number. The second time, we'll read the track number. The third time, we'll read the physical sector number. We don't actually care about the disk volume or the track number, and once we get the sector number, we don't verify the address field checksum.
BEC4	20 E4 BE	JSR $BEE4	
BEC7	2A	ROL	
BEC8	8D E0 BD	STA $BDE0	
BECB	20 E4 BE	JSR $BEE4	
BECE	2D E0 BD	AND $BDE0	
BED1	88	DEY	
BED2	D0 F0	BNE $BEC4	
BED4	60	RTS	On exit, the accumulator contains the physical sector number.

Continuing from $BD39…

BD39	A8	TAY	use physical sector number as an index into the sector address array
BD3A	BE 84 BF	LDX $BF84,Y	get the target page (where we want to store this sector in memory)
BD3D	F0 F0	BEQ $BD2F	if the target page is #$00, it means we've already read this sector, so loop back to find the next address prologue
BD3F	8D E0 BD	STA $BDE0	store the physical sector number later in this routine
BD42	8E 64 BD	STX $BD64	store the target page in several places throughout this routine
BD45	8E C4 BD	STX $BDC4	
BD48	8E 7C BD	STX $BD7C	
BD4B	8E 8E BD	STX $BD8E	
BD4E	8E A6 BD	STX $BDA6	
BD51	8E BE BD	STX $BDBE	
BD54	E8	INX	
BD55	8E D9 BD	STX $BDD9	
BD58	CA	DEX	
BD59	CA	DEX	
BD5A	8E 94 BD	STX $BD94	
BD5D	8E AC BD	STX $BDAC	
BD60	A0 FE	LDY #$FE	Save the two bytes immediately after the target page, because we're going to use them for temporary storage. (We'll restore them later.)
BD62	B9 02 D1	LDA $D102,Y	
BD65	48	PHA	
BD66	C8	INY	
BD67	D0 F9	BNE $BD62	
BD69	B0 C4	BCS $BD2F	this is an unconditional branch
BD6B	E0 00	CPX #$00	execution continues here (from $BD34) after matching the data prologue
BD6D	F0 C0	BEQ $BD2F	If X is still #$00, it means we found a data prologue before we found an address prologue. In that case, we have to skip this sector, because we don't know which sector it is and we wouldn't know where to put it. Sad!

Nibble loop #1 reads nibbles `$00..$55`, looks up the corresponding offset in the preshift table at `$BB96`, and stores that offset in the temporary two-byte buffer after the target page.

BD6F	8D 7E BD	STA $BD7E	initialize rolling checksum to #$00, or update it with the results from the calculations below
BD72	AE EC C0	LDX $C0EC ☺	read one nibble from disk
BD75	10 FB	BPL $BD72	
BD77	BD 00 BB	LDA $BB00,X	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 $BB96, we index off $BB00 + X. Math!
BD7A ☺	99 02 D1	STA $D102,Y	Now the accumulator has the offset into the table of individual 2-bit combinations ($BC00..$BCFF). Store that offset in a temporary buffer towards the end of the target page. (It will eventually get overwritten by full 8-bit bytes, but in the meantime it's a useful $56-byte scratch space.)
BD7D	49 D1	EOR #$D1 ☺	The EOR value is set at $BD6F each time through loop #1.
BD7F	C8	INY	The `Y` register started at #$AA (set by the "TAY" instruction at $BD39), so this loop reads a total of #$56 nibbles.
BD80	D0 ED	BNE $BD6F	

Here endeth nibble loop #1.

Nibble 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.

BD82	A0 AA	LDY #$AA	
BD84	AE EC C0	LDX $C0EC ☺	
BD87	10 FB	BPL $BD84	
BD89	5D 00 BB	EOR $BB00,X	
BD8C ☺	BE 02 D1	LDX $D102,Y	
BD8F	5D 02 BC	EOR $BC02,X	
BD92 ☺	99 56 D1	STA $D156,Y	This address was set at $BD5A based on the target page (minus 1 so we can add Y from #$AA..#$FF).
BD95	C8	INY	
BD96	D0 EC	BNE $BD84	

Here endeth nibble loop #2.

Nibble 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.

```
BD98      29 FC   AND #$FC
BD9A      A0 AA   LDY #$AA
BD9C   AE EC C0   LDX $C0EC o_O
BD9F      10 FB   BPL $BD9C
BDA1   5D 00 BB   EOR $BB00,X
BDA4   BE 02 D1   LDX $D102,Y
o_O
BDA7   5D 01 BC   EOR $BC01,X

BDAA   99 AC D1   STA $D1AC,Y   This address was set at $BD5D
o_O                             based on the target page
BDAD         C8   INY           (minus 1 so we can add Y
BDAE      D0 EC   BNE $BD9C     from #$AA..#$FF).
```

Here endeth nibble loop #3.

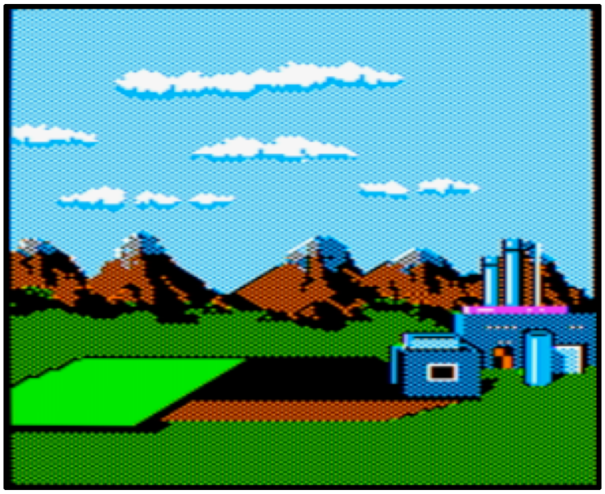
Loop #4 reads nibbles `$102..$155`, combines them with bits 4-5 of the appropriate nibble from the first `$56`, and stores them in bytes `$AC..$101` of the target page in memory. (This overwrites two bytes after the end of the target page, but we'll restore then later from the stack.)

```
BDB0      29 FC   AND #$FC
BDB2      A2 AC   LDX #$AC
BDB4   AC EC C0   LDY $C0EC o_O
BDB7      10 FB   BPL $BDB4
BDB9   59 00 BB   EOR $BB00,Y
BDBC   BC 00 D1   LDY $D100,X
o_O
BDBF   59 00 BC   EOR $BC00,Y

BDC2   9D 00 D1   STA $D100,X   This address was set at $BD45
o_O                             based on the target page.
BDC5         E8   INX
BDC6      D0 EC   BNE $BDB4
```

Here endeth nibble loop #4.

```
BDC8      29 FC   AND #$FC      Finally, get the last nibble
BDCA   AC EC C0   LDY $C0EC o_O and convert it to a byte. This
BDCD      10 FB   BPL $BDCA     should equal all the previous
BDCF   59 00 BB   EOR $BB00,Y   bytes XOR'd together. (This
                                is the standard checksum
                                algorithm shared by all
                                16-sector disks.)

BDD2      C9 01   CMP #$01      set carry if value is anything
                                but 0

BDD4      A0 01   LDY #$01      Restore the original data in
BDD6         68   PLA           the two bytes after the target
BDD7   99 00 D1   STA $D100,Y   page. (This does not affect
o_O                             the carry flag, which we will
BDDA         88   DEY           check in a moment, but we
BDDB      10 F9   BPL $BDD6     need to restore these bytes
                                now to balance out the
                                pushing to the stack we did at
                                $BD65.)

BDDD      B0 8A   BCS $BD69     if data checksum failed at
                                $BDD2, start over

BDDF      A0 D1   LDY #$D1 o_O  This was set to the physical
BDE1         8A   TXA           sector number (at $BD3F), so
                                this is a index into the
                                16-byte array at $BF84.

BDE2   99 84 BF   STA $BF84,Y   store #$00 at this location in
                                the sector array to indicate
                                that we've read this sector

BDE5   CE EF BD   DEC $BDEF     decrement sector count
BDE8   CE 94 BF   DEC $BF94
BDEB         38   SEC

BDEC      D0 EF   BNE $BDDD     If the sectors-left-in-this-track
                                count (in $BF94) isn't zero
                                yet, loop back to read more
                                sectors.

BDEE      A2 D1   LDX #$D1 o_O  If the total sector count (in
BDF0      F0 09   BEQ $BDFB     $BDEF, set at $BD04 and
                                decremented at $BDE5) is zero,
                                we're done—no need to read
                                the rest of the track. (This
                                lets us have sector counts that
                                are not multiples of 16, i.e.
                                reading just a few sectors
                                from the last track of a
                                multi-track block.)

BDF2   EE 10 BF   INC $BF10     increment phase (twice, so it
BDF5   EE 10 BF   INC $BF10     points to the next whole
                                block)

BDF8   4C 10 BD   JMP $BD10     jump back to seek and read
                                from the next track

BDFB   AD E8 C0   LDA $C0E8 o_O Execution continues here
BDFE         60   RTS           (from $BDEF). We're all done,
                                so turn off drive motor and
                                exit.
```

And that's all she wrote`^H^H^H^H`read.



I Make My Verse For The Universe

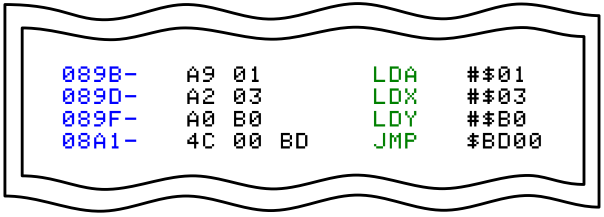
How's our master plan from page 47 going? Pretty darn well, I'd say.

Step 1) write all the game code to a standard disk. Done.

Step 2) write an RWTS. Done.

Step 3) make them talk to each other.

The "glue code" for this final step lives on track 1. It was loaded into memory at the very end of the boot sector:



```
089B-    A9 01          LDA    #$01
089D-    A2 03          LDX    #$03
089F-    A0 B0          LDY    #$B0
08A1-    4C 00 BD       JMP    $BD00
```

That loads 3 sectors from track 1 into `$B000..$B2FF`. `$B000` is the high scores, which stays at `$B000`. `$B100` is moved to zero page. `$B200` is the output vector and final initialization code. This page is never used by the game. (It was used by the original RWTS, but that has been greatly simplified by stripping out the copy protection. I love when that happens!)

Here is my output vector, replacing the code that originally lived at `$BF6F`:

```
*B200L
B200      C9 07    CMP #$07        command or regular
                                   character?
B202      90 03    BCC $B207       command -> branch

B204   6C 3A 00    JMP ($003A)     regular character -> print to
                                   screen
B207      85 5F    STA $5F         store command in zero page

B209         A8    TAY             set up the call to the screen
B20A   B9 97 B2    LDA $B297,Y     fill
B20D   8D 19 B2    STA $B219

B210   B9 9E B2    LDA $B29E,Y     set up the call to Gumboot
B213   8D 1C B2    STA $B21C

B216      A9 00    LDA #$00        call the appropriate screen fill
B218   20 69 B2    JSR $B269 ☺

B21B   20 2B B2    JSR $B22B ☺     call Gumboot

B21E      A5 5F    LDA $5F         find the entry point for this
B220         0A    ASL             block
B221         A8    TAY

B222   B9 A6 B2    LDA $B2A6,Y     push the entry point to the
B225         48    PHA             stack
B226   B9 A5 B2    LDA $B2A5,Y
B229         48    PHA

B22A         60    RTS             and exit via "RTS"
```

This is the routine that calls Gumboot to load the appropriate blocks of game code from the disk, according to the disk map on page 47. Here is the summary of which sectors are loaded by each block:

cmd	track (A)	count (X)	page (Y)
$00	$02	$38	$08
	$06	$28	$60
$01	$09	$38	$08
	$0D	$50	$60
$02	$12	$38	$08
	$16	$28	$60
$03	$19	$20	$20

(The parameters for command `#$06` are the same as command `#$01`.)

The lookup at `$B210` modified the "JSR" instruction at `$B21B`, so each command starts in a different place:

```
B22B      A9 02    LDA #$02        command #$00
B22D   20 56 B2    JSR $B256
B230      A9 06    LDA #$06
B232      D0 1C    BNE $B250

B234      A9 09    LDA #$09        command #$01
B236   20 56 B2    JSR $B256
B239      A9 0D    LDA #$0D
B23B      A2 50    LDX #$50
B23D      D0 13    BNE $B252

B23F      A9 12    LDA #$12        command #$02
B241   20 56 B2    JSR $B256
B244      A9 16    LDA #$16
B246      D0 08    BNE $B250

B248      A9 19    LDA #$19        command #$03
B24A      A2 20    LDX #$20
B24C      A0 20    LDY #$20
B24E      D0 0A    BNE $B25A
B250      A2 28    LDX #$28
B252      A0 60    LDY #$60
B254      D0 04    BNE $B25A
B256      A2 38    LDX #$38
B258      A0 08    LDY #$08
B25A   4C 00 BD    JMP $BD00

B25D      A9 01    LDA #$01        command #$04: seek to track
B25F   20 00 BF    JSR $BF00       1 and write $B000..$B0FF to
B262      A9 00    LDA #$00        sector 0
B264      A0 B0    LDY #$B0
B266   4C 00 BE    JMP $BE00
```

```
B269     A5 60    LDA $60         exact replica of the screen fill
B26B  4D 50 C0    EOR $C050       code that was originally at
B26E     85 60    STA $60         $BEB0
B270     29 0F    AND #$0F
B272     F0 F5    BEQ $B269
B274     C9 0F    CMP #$0F
B276     F0 F1    BEQ $B269
B278  20 66 F8    JSR $F866
B27B     A9 17    LDA #$17
B27D        48    PHA
B27E  20 47 F8    JSR $F847
B281     A0 27    LDY #$27
B283     A5 30    LDA $30
B285     91 26    STA ($26),Y
B287        88    DEY
B288     10 FB    BPL $B285
B28A        68    PLA
B28B        38    SEC
B28C     E9 01    SBC #$01
B28E     10 ED    BPL $B27D
B290  AD 56 C0    LDA $C056
B293  AD 54 C0    LDA $C054
B296        60    RTS
```

```
B297 [69 7B 69 69 96 96 69]    lookup table for screen fills

B29E [2B 34 3F 48 2A 2A 34]    lookup table for Gumboot
                               calls

B2A5 [9C 0F]                   lookup table for entry points
B2A7 [F8 31]
B2A9 [34 10]
B2AB [57 FF]
B2AD [5C B2]
B2AF [95 B2]
B2B1 [77 23]
```

Last but not least, a short routine at `$B2F1` to move zero page into place and start the game. (This is called because we pushed `#$B2`/`#$F0` to the stack in our boot sector, at `$0895`.)

```
*B2F1L
B2F1     A2 00    LDX #$00        copy $B100 to zero page
B2F3  BD 00 B1    LDA $B100,X
B2F6     95 00    STA $00,X
B2F8        E8    INX
B2F9     D0 F8    BNE $B2F3

B2FB     A9 00    LDA #$00        print a null character to start
B2FD  4C ED FD    JMP $FDED       the game
```

Quod erat liberand one more thing…

Oops

Heeeeey there. Remember this code?

```
0372     A9 34    LDA #$34
0374        48    PHA
...
0378        28    PLP
```

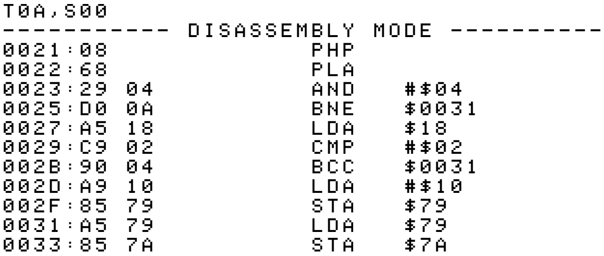
Here's what I said about it when I first saw it:

> pop that #$34 off the stack, but use it as status registers (weird, but legal—if it turns out to matter, I can figure out exactly which status bits get set and cleared)

Yeah, so that turned out to be more important than I thought. After extensive play testing, we[26] discovered the game becomes unplayable on level 3.

How unplayable? Gates that are open won't close; balls pass through gates that are already closed; bins won't move more than a few pixels.

So, not a crash, and (contrary to our first guess) not an incompatibility with modern emulators. It affects real hardware too, and it was intentional. Deep within the game code, there are several instances of code like this:



```
T0A,S00
----------- DISASSEMBLY MODE ----------
0021:08                PHP
0022:68                PLA
0023:29 04             AND    #$04
0025:D0 0A             BNE    $0031
0027:A5 18             LDA    $18
0029:C9 02             CMP    #$02
002B:90 04             BCC    $0031
002D:A9 10             LDA    #$10
002F:85 79             STA    $79
0031:A5 79             LDA    $79
0033:85 7A             STA    $7A
```

"PHP" pushes the status registers on the stack, but "PLA" pulls a value from the stack and stores it as a byte, in the accumulator. That's… weird. Also, it's the reverse of the weird code we saw at `$0372`, which took a byte in the accumulator and blitted it into the status registers. Then "`AND #$04`" isolates one status bit in particular: the interrupt flag. The rest of the code is the game-specific way of making the game unplayable.

This is a very convoluted, obfuscated, sneaky way to ensure that the game was loaded through its original bootloader. Which, of course, it wasn't.

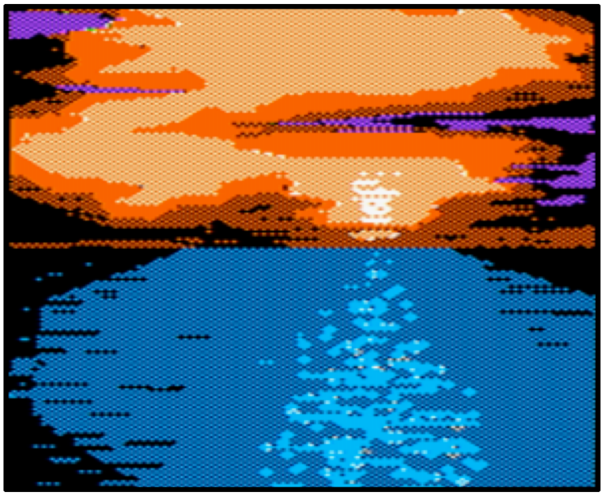
The solution: after loading each block of game code and pushing the new entry point to the stack, set the interrupt flag.

```
B222  B9 A6 B2    LDA $B2A6,Y     pop that #$34 off the stack,
B225        48    PHA             but use it as status registers
B226  B9 A5 B2    LDA $B2A5,Y     (weird, but legal—if it turns
B229        48    PHA             out to matter, I can figure out
                                  exactly which status bits get
                                  set and cleared) push the
                                  entry point to the stack

B22A        78    SEI             set the interrupt flag (new!)


B22B        60    RTS             and exit via "RTS"
```

Many thanks to Marco V. for reporting this and helping reproduce it; qkumba for digging into it to find the check within the game code; Tom G. for making the connection between the interrupt flag and the weird "LDA/PHA/PLP" code at `$0372`.

---

[26] not me, and not qkumba either, who beat the entire game twice. It was Marco V. Thanks, Marco!



This Is Not The End, Though

This game holds one more secret, but it's not related to the copy protection, thank goodness. As far as I can tell, this secret has not been revealed in 33 years. qkumba found it because of course he did.

Once the game starts, press `Ctrl-J` to switch to joystick mode. Press and hold button 2 to activate "targeting" mode, then move your joystick to the bottom-left corner of the screen and also press button 1. The screen will be replaced by this message:

PRESS CTRL-Z DURING THE CARTOONS

Now, the game has 5 levels. After you complete a level, your character gets promoted: worker, foreman, supervisor, manager, and finally vice president. Each of these is a little cartoon—what kids today would call a cut scene. When you complete the entire game, it shows a final screen and your character retires.

Pressing `Ctrl-Z` during each cartoon reveals four ciphers.

After level 1:

RBJRY JSYRR

After level 2:

VRJJRY ZIAR

After level 3:

ESRB

After level 4:

FIG YRJMYR

Taken together, they form a simple substitution cipher:

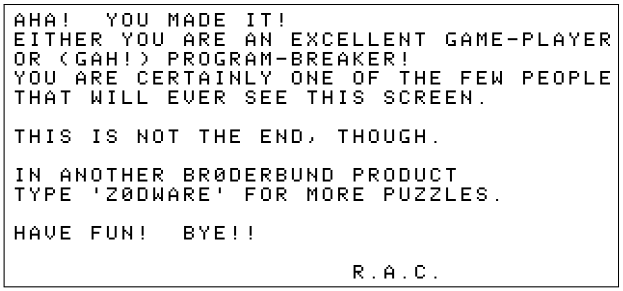
- ENTER THREE
- LETTER CODE
- WHEN
- YOU RETIRE

But what is the code?

It turns out that pressing `Ctrl-Z` *again*, while any of the pieces of the cipher are on screen, reveals another clue:

DOUBLE HELIX

Entering the three-letter code `DNA` at the "retirement" screen reveals the final secret message:



```
AHA!  YOU MADE IT!
EITHER YOU ARE AN EXCELLENT GAME-PLAYER
OR (GAH!) PROGRAM-BREAKER!
YOU ARE CERTAINLY ONE OF THE FEW PEOPLE
THAT WILL EVER SEE THIS SCREEN.

THIS IS NOT THE END, THOUGH.

IN ANOTHER BR0DERBUND PRODUCT
TYPE 'Z0DWARE' FOR MORE PUZZLES.

HAVE FUN!  BYE!!

                           R.A.C.
```

At time of writing, no one has found the "Z0DWARE" puzzle. You could be the first!

Keys and Controls

The game can be played with a joystick or keyboard.

`Ctrl-J` switch to joystick mode
`Ctrl-K` switch to keyboard mode

When using a keyboard:

`S` move bins left
`D` stop bins
`F` move bins right
`Space` switch in-tube gates
`E` increase speed
`C` decrease speed

`Return` toggle target sighting

`U I O` move the target sight

`J K L` (for when the bombs
`M , .` start dropping)

When using a joystick:

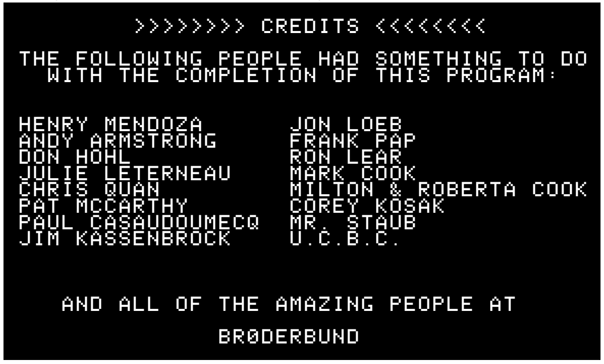
`buttons 0+1` toggle target sighting

`Ctrl-X` flip joystick X axis
`Ctrl-Y` flip joystick Y axis

Other keys:

`Ctrl-S` toggle sound on/off
`Ctrl-R` restart level
`Ctrl-Q` restart game
`Ctrl-H` view high scores
`Esc` pause/resume game

After the game starts, press `Ctrl-U Ctrl-C Ctrl-B` in sequence to see a secret credits page that lists most of the people involved in making the game. Sadly, the author of the copy protection is not listed.



```
         >>>>>>>> CREDITS <<<<<<<<

THE FOLLOWING PEOPLE HAD SOMETHING TO DO
  WITH THE COMPLETION OF THIS PROGRAM:

HENRY MENDOZA          JON LOEB
ANDY ARMSTRONG         FRANK PAP
DON HOHL               RON LEAR
JULIE LETERNEAU        MARK COOK
CHRIS QUAN             MILTON & ROBERTA COOK
PAT MCCARTHY           COREY KOSAK
PAUL CASAUDOUMECQ      MR. STAUB
JIM KASSENBROCK        U.C.B.C.


   AND ALL OF THE AMAZING PEOPLE AT

              BR0DERBUND
```

Cheats

I have not enabled any cheats on our release, but I have verified that they work. You can use any or all of them:

Stop the clock
`T09,S0A,$B1`
change `01` to `00`

Start on level 2-5
`T09,S0C,$53`
change `00` to <level-1>

Acknowledgements

Thanks to Alex, Andrew, John, Martin, Paul, Quinn, and Richard for reviewing drafts of this write-up.

And finally, many thanks to qkumba: Shifter of Bits, Master of the Stack, author of Gumboot, and my friend.