# BEER RUN

2015-05-24

4am
to deprotect
and preserve

# Contents

```
----------------Beer Run----------------
A 4am crack                   2015-05-24
-------------------. updated 2015-09-06
                   |____________________
```

Name: Beer Run
Genre: arcade
Year: 1981
Author: Mark Turmell
Publisher: Sirius Software
Media: single-sided 5.25-inch floppy
OS: custom
Other versions: The Chief Surgeon /
  Black Bag crack

This game is a single-load... almost. It initially boots to an animated title screen, and game play follows without any disk access. But once the game is over, it reads several tracks from disk before returning to the title screen.

The original disk is write-protected (un-notched), so it's not saving high scores. The post-game disk access could be purely copy protection (like Sneakers), or there could be code and data that is only used during the title screen which is reloaded from disk as needed (like Repton and Plasmania).

There are two classic file-based cracks of this game. One shows the animated title screen the first time through; the other strips out the title screen altogether.

The original disk only boots on an Apple II+ or an unenhanced Apple ][e. Later models appear to load the entire game into memory, then hang.

# Chapter 0
# In Which Various Automated Tools Fail In Interesting Ways

COPYA
immediate disk read error

Locksmith Fast Disk Backup
unable to read any track

EDD 4 bit copy (no sync, no count)
hangs during boot

Copy ][+ nibble editor
track 0 has some 4-4 encoded data
other tracks are unreadable

Disk Fixer
nope (can't read 4-4 encoded tracks)

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. Could be a nibble check during boot. Could be that the data is loaded from half tracks. Could be both, or neither.

Next steps:

1. Trace the boot
2. Capture the game in memory
3. See what's going on with the post-game disk access

# Chapter 1
# In Which We Find A Very Unfriendly "Do Not Disturb" Sign

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

]PR#5
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0

]BLOAD BOOT0,A$800
]CALL -151

*801L

; display hi-res graphics page
; (uninitialized)
0801-   8D 50 C0    STA   $C050
0804-   8D 52 C0    STA   $C052
0807-   8D 54 C0    STA   $C054
080A-   8D 57 C0    STA   $C057

; get slot (x16)
080D-   A6 2B       LDX   $2B

; a counter? or an address?
080F-   A9 04       LDA   #$04
0811-   85 11       STA   $11
0813-   A0 00       LDY   #$00
0815-   84 10       STY   $10
```

```
; look for custom prologue ("DD AD DA")
0817-   BD 8C C0    LDA   $C08C,X
081A-   10 FB       BPL   $0817
081C-   C9 DD       CMP   #$DD
081E-   D0 F7       BNE   $0817
0820-   BD 8C C0    LDA   $C08C,X
0823-   10 FB       BPL   $0820
0825-   C9 AD       CMP   #$AD
0827-   D0 F3       BNE   $081C
0829-   BD 8C C0    LDA   $C08C,X
082C-   10 FB       BPL   $0829
082E-   C9 DA       CMP   #$DA
0830-   D0 EA       BNE   $081C

; read 4-4 encoded data immediately
; (no address field, no sector numbers)
0832-   BD 8C C0    LDA   $C08C,X
0835-   10 FB       BPL   $0832
0837-   38          SEC
0838-   2A          ROL
0839-   85 0E       STA   $0E
083B-   BD 8C C0    LDA   $C08C,X
083E-   10 FB       BPL   $083B
0840-   25 0E       AND   $0E

; ($10) is an address, initialized at
; $080F as $0400 (yes, the text page)
0842-   91 10       STA   ($10),Y
0844-   C8          INY
0845-   D0 EB       BNE   $0832
0847-   E6 11       INC   $11
0849-   A5 11       LDA   $11

; loop until we hit page 8 (i.e. we're
; filling $0400..$07FF)
084B-   C9 08       CMP   #$08
084D-   D0 E3       BNE   $0832
084F-   BD 80 C0    LDA   $C080,X
```

```
; clear $0900..$BFFF in main memory
0852-   A9 09       LDA   #$09
0854-   85 01       STA   $01
0856-   A9 00       LDA   #$00
0858-   85 00       STA   $00
085A-   A8          TAY
085B-   A2 B7       LDX   #$B7
085D-   91 00       STA   ($00),Y
085F-   C8          INY
0860-   D0 FB       BNE   $085D
0862-   E6 01       INC   $01
0864-   CA          DEX
0865-   D0 F6       BNE   $085D

; calculate a checksum of page 8 (this
; code right here)
0867-   8A          TXA
0868-   E8          INX
0869-   F0 06       BEQ   $0871
086B-   5D 00 08    EOR   $0800,X
086E-   4C 68 08    JMP   $0868

; use the stack pointer (!) to keep a
; copy of that checksum
0871-   AA          TAX
0872-   9A          TXS

; calculate another checksum of zero
; page
0873-   A2 00       LDX   #$00
0875-   8A          TXA
0876-   55 00       EOR   $00,X
0878-   E8          INX
0879-   D0 FB       BNE   $0876

; get slot (x16) again
087B-   A6 2B       LDX   $2B
```

```
; jump to the code we just read into
; the text page
087D-   4C 00 04    JMP   $0400
```

Well that's lovely. I need to interrupt the boot at $087D, but if I do, it will modify the checksum that ends up in the stack pointer (which is a great place to stash a checksum as long as you never use PHA, PLA, PHP, PLP, JSR, RTS, or RTI).

It's also wiping main memory, including the place I usually put my boot trace callbacks (around $9700).

So, a three-pronged attack:

1. Relocate the code to $0900. Most of it uses relative branching already, except for one JMP at $086E, which I can patch. The code will still run, but I'll be able to patch it without altering the checksum.

2. Disable the memory wipe at $095D.

3. Patch the code at $097D to jump to a routine under my control.

# Chapter 2
# In Which Nothing Happens, Inhospitably

```
*9600<C600.C6FFM

; relocate the code from $0800 to $0900
96F8-    A0 00          LDY    #$00
96FA-    B9 00 08       LDA    $0800,Y
96FD-    99 00 09       STA    $0900,Y
9700-    C8             INY
9701-    D0 F7          BNE    $96FA

; disable the memory wipe by changing
; STA to BIT
9703-    A9 24          LDA    #$24
9705-    8D 5D 09       STA    $095D

; fix the absolute JMP address
9708-    A9 09          LDA    #$09
970A-    8D 70 09       STA    $0970

; set up the callback
970D-    A9 1A          LDA    #$1A
970F-    8D 7E 09       STA    $097E
9712-    A9 97          LDA    #$97
9714-    8D 7F 09       STA    $097F

; start the boot
9717-    4C 01 09       JMP    $0901
```

```
; callback is here
; copy the code on the text page to
; higher memory so it will survive a
; reboot
971A-   A2 04       LDX   #$04
971C-   A0 00       LDY   #$00
971E-   B9 00 04    LDA   $0400,Y
9721-   99 00 24    STA   $2400,Y
9724-   C8          INY
9725-   D0 F7       BNE   $971E
9727-   EE 20 97    INC   $9720
972A-   EE 23 97    INC   $9723
972D-   CA          DEX
972E-   D0 EE       BNE   $971E

; turn off slot 6 drive motor and
; reboot to my work disk in slot 5
9730-   AD E8 C0    LDA   $C0E8
9733-   4C 00 C5    JMP   $C500

*BSAVE TRACE,A$9600,L$136
*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. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2400L

; calculate another checksum of zero
; page, starting with the value of the
; previous checksum (at $0873)
2400-   A0 00       LDY   #$00
2402-   59 00 00    EOR   $0000,Y
2405-   C8          INY
2406-   D0 FA       BNE   $2402
2408-   A8          TAY

; if equal, nothing has changed (we've
; EOR'd everything twice, so we're back
; to zero)
2409-   F0 03       BEQ   $240E

; if checksums don't match, jump to
; (what I presume is) The Badlands
240B-   4C 40 05    JMP   $0540

*2540L

; clear most of main memory
2540-   A0 00       LDY   #$00
2542-   84 00       STY   $00
2544-   A9 0C       LDA   #$0C
2546-   85 01       STA   $01
2548-   A2 B4       LDX   #$B4
254A-   98          TYA
254B-   91 00       STA   ($00),Y
254D-   C8          INY
254E-   D0 FB       BNE   $254B
2550-   E6 01       INC   $01
2552-   CA          DEX
2553-   D0 F6       BNE   $254B
```

```
; play a cute sound
2555-   A9 C0       LDA   #$C0
2557-   85 00       STA   $00
2559-   A0 C0       LDY   #$C0
255B-   AD 30 C0    LDA   $C030
255E-   A6 00       LDX   $00
2560-   CA          DEX
2561-   D0 FD       BNE   $2560
2563-   88          DEY
2564-   D0 F5       BNE   $255B
2566-   46 00       LSR   $00
2568-   D0 EF       BNE   $2559

; and reboot from whence we came
256A-   A6 2B       LDX   $2B
256C-   CA          DEX
256D-   8A          TXA
256E-   4A          LSR
256F-   4A          LSR
2570-   4A          LSR
2571-   4A          LSR
2572-   09 C0       ORA   #$C0
2574-   48          PHA
2575-   A9 FF       LDA   #$FF
2577-   48          PHA
2578-   60          RTS
```

Continuing at $040E...

```
*240EL

; set reset vector to The Badlands
240E-   A9 40       LDA   #$40
2410-   8D F2 03    STA   $03F2
2413-   A9 05       LDA   #$05
2415-   8D F3 03    STA   $03F3
2418-   49 A5       EOR   #$A5
241A-   8D F4 03    STA   $03F4
241D-   86 2B       STX   $2B
241F-   EA          NOP

; read from ROM but write to RAM bank 2
2420-   AD 81 C0    LDA   $C081
2423-   AD 81 C0    LDA   $C081

; wipe RAM bank 2 by copying ROM
2426-   A0 00       LDY   #$00
2428-   84 00       STY   $00
242A-   A9 D0       LDA   #$D0
242C-   85 01       STA   $01
242E-   B1 00       LDA   ($00),Y
2430-   91 00       STA   ($00),Y
2432-   C8          INY
2433-   D0 F9       BNE   $242E
2435-   E6 01       INC   $01
2437-   D0 F5       BNE   $242E

; set low-level reset vector while the
; language card RAM is writeable
2439-   A9 40       LDA   #$40
243B-   8D FC FF    STA   $FFFC
243E-   A9 05       LDA   #$05
2440-   8D FD FF    STA   $FFFD

; switch back to ROM
2443-   AD 80 C0    LDA   $C080
```

```
; set input and output vectors to
; something unpleasant
2446-   A9 A2       LDA   #$A2
2448-   85 36       STA   $36
244A-   85 38       STA   $38
244C-   A9 05       LDA   #$05
244E-   85 37       STA   $37
2450-   85 39       STA   $39

; take the checksum from boot0 (that we
; stashed in the stack pointer) and put
; it in zero page $0B
2452-   A9 00       LDA   #$00
2454-   BA          TSX
2455-   86 0B       STX   $0B
2457-   85 0C       STA   $0C
2459-   85 0D       STA   $0D
245B-   85 0E       STA   $0E

; use that checksum (now in zero page
; $0B) as the starting value of ANOTHER
; checksum of all the code on the text
; page (including this code right here)
245D-   A5 0B       LDA   $0B
245F-   A2 00       LDX   #$00
2461-   5D 00 04    EOR   $0400,X
2464-   5D 00 05    EOR   $0500,X
2467-   5D 00 06    EOR   $0600,X
246A-   5D 00 07    EOR   $0700,X
246D-   E8          INX
246E-   D0 F1       BNE   $2461
2470-   AA          TAX

; and put the new checksum back into
; the stack pointer
2471-   9A          TXS
```

# Chapter 3
# You're Very Clever, Young Man, But It's Checksums All The Way Down

```
*9600<C600.C6FFM

; move boot0 to $0900 and patch it up
96F8-   A0 00       LDY   #$00
96FA-   B9 00 08    LDA   $0800,Y
96FD-   99 00 09    STA   $0900,Y
9700-   C8          INY
9701-   D0 F7       BNE   $96FA
9703-   A9 24       LDA   #$24
9705-   8D 5D 09    STA   $095D
9708-   A9 09       LDA   #$09
970A-   8D 70 09    STA   $0970

; set up callback after first checksum
; is calculated
970D-   A9 4C       LDA   #$4C
970F-   8D 71 09    STA   $0971
9712-   A9 1F       LDA   #$1F
9714-   8D 72 09    STA   $0972
9717-   A9 97       LDA   #$97
9719-   8D 73 09    STA   $0973

; start the boot
971C-   4C 01 09    JMP   $0901

; callback is here
; save the checksum and unconditionally
; break to the monitor
971F-   8D 25 97    STA   $9725
9722-   4C 59 FF    JMP   $FF59
9725-   00          BRK

*BSAVE TRACE CHECKSUM,A$9600,L$126
*9600G
...reboots slot 6...
<beep>
```

```
*9725

9725- 20
```

The initial checksum of boot0 is $20.

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

*9600<C600.C6FFM

; move boot0 to $0900 and patch it up
96F8-   A0 00       LDY   #$00
96FA-   B9 00 08    LDA   $0800,Y
96FD-   99 00 09    STA   $0900,Y
9700-   C8          INY
9701-   D0 F7       BNE   $96FA
9703-   A9 24       LDA   #$24
9705-   8D 5D 09    STA   $095D
9708-   A9 09       LDA   #$09
970A-   8D 70 09    STA   $0970

; set up callback instead of jumping to
; boot1 at $0400
970D-   A9 1A       LDA   #$1A
970F-   8D 7E 09    STA   $097E
9712-   A9 97       LDA   #$97
9714-   8D 7F 09    STA   $097F

; start the boot
9717-   4C 01 09    JMP   $0901
```

```
; callback is here
; hard-code the initial checksum value
; ($20), then reproduce the checksum on
; the boot1 code before we start
; patching it to high heaven
971A-   A9 20       LDA   #$20
971C-   A2 00       LDX   #$00
971E-   5D 00 04    EOR   $0400,X
9721-   5D 00 05    EOR   $0500,X
9724-   5D 00 06    EOR   $0600,X
9727-   5D 00 07    EOR   $0700,X
972A-   E8          INX
972B-   D0 F1       BNE   $971E

; store the new checksum and break
972D-   8D 33 97    STA   $9733
9730-   4C 59 FF    JMP   $FF59
9733-   00          BRK

*BSAVE TRACE CHECKSUM 2,A$9600,L$134
*9600G
...reboots slot 6...
<beep>

*9733

9733- 01
```

The second checksum, moved to the stack pointer at $0471, is $01.

# Chapter 4
# Half A Track Is Better Than None

Continuing the boot trace at $0472...

```
*C500G
...
]BLOAD BOOT1 0400-07FF,A$2400
]CALL -151

*2472L

2472-   A0 03       LDY   #$03
2474-   20 DC 04    JSR   $04DC

*24DCL

; advance drive head by one phase
; (a.k.a. a half track)
24DC-   E6 0C       INC   $0C
24DE-   A5 0C       LDA   $0C
24E0-   29 03       AND   #$03
24E2-   0A          ASL
24E3-   05 2B       ORA   $2B
24E5-   AA          TAX
24E6-   BD 81 C0    LDA   $C081,X
24E9-   20 F8 04    JSR   $04F8
24EC-   BD 80 C0    LDA   $C080,X
24EF-   20 F8 04    JSR   $04F8

; loop a number of times (given in the
; Y register on entry)
24F2-   88          DEY
24F3-   D0 E7       BNE   $24DC
24F5-   A6 2B       LDX   $2B
24F7-   60          RTS
24F8-   A9 40       LDA   #$40
24FA-   8D 50 C0    STA   $C050
24FD-   4C A8 FC    JMP   $FCA8
```

We started on track 0 and advanced the drive head by 3 phases, so now we're on track 1.5.

```
; get target page (given in an array at
; $05D0)
2477-   A4 0E       LDY   $0E
2479-   B9 D0 05    LDA   $05D0,Y
247C-   D0 03       BNE   $2481

; if target page = 0, we're done, so
; continue at $0500 (will get to that
; shortly)
247E-   4C 00 05    JMP   $0500

2481-   20 90 04    JSR   $0490

*2490L

; set up target page
2490-   85 05       STA   $05
2492-   18          CLC

; sector count (4-4 encoded tracks can
; only hold $0C pages worth of data)
2493-   A9 0C       LDA   #$0C
2495-   85 06       STA   $06
2497-   A0 00       LDY   #$00
2499-   84 04       STY   $04
```

```
; find custom prologue "DD AD DA"
249B-   BD 8C C0    LDA   $C08C,X
249E-   10 FB       BPL   $249B
24A0-   C9 DD       CMP   #$DD
24A2-   D0 F7       BNE   $249B
24A4-   BD 8C C0    LDA   $C08C,X
24A7-   10 FB       BPL   $24A4
24A9-   C9 AD       CMP   #$AD
24AB-   D0 F3       BNE   $24A0
24AD-   BD 8C C0    LDA   $C08C,X
24B0-   10 FB       BPL   $24AD
24B2-   C9 DA       CMP   #$DA
24B4-   D0 EA       BNE   $24A0

; now read 4-4 encoded data into ($04)
24B6-   BD 8C C0    LDA   $C08C,X
24B9-   10 FB       BPL   $24B6
24BB-   38          SEC
24BC-   2A          ROL
24BD-   85 0F       STA   $0F
24BF-   8D 50 C0    STA   $C050
24C2-   BD 8C C0    LDA   $C08C,X
24C5-   10 FB       BPL   $24C2
24C7-   25 0F       AND   $0F
24C9-   91 04       STA   ($04),Y
24CB-   C8          INY
24CC-   D0 E8       BNE   $24B6

; increment target page
24CE-   E6 05       INC   $05

; decrement sector count
24D0-   C6 06       DEC   $06
```

```
; Loop back to read more. Note: this
; goes directly to data read routine,
; not the prologue match routine. There
; is only one prologue per track.
24D2-   D0 E2         BNE   $24B6
24D4-   60            RTS
```

Continuing at $0484...

```
*2484L

; sets Y=2 and falls through to drive
; head advance routine, so this will
; skip ahead 2 phases = 1 whole track,
; so we're still on half tracks but now
; 2.5, 3.5, 4.5, &c.
2484-   20 D8 04      JSR   $04D8

; This routine literally does nothing.
; I'm assuming this boot routine was
; repurposed from other Sirius titles
; that have animated load screens, but
; this disk does not.
2487-   20 00 06      JSR   $0600

; increment page index
248A-   E6 0E         INC   $0E

; and branch back (exits via $0500 when
; the target page = 0)
248C-   4C 77 04      JMP   $0477
```

Here is the target page table (accessed at $0479):

```
*25D0.25DF

25D0- 08 14 60 6C 78 84 90 9C
25D8- A8 B4 00 00 00 00 00 00
```

To sum up:

- We're reading data from consecutive half tracks (1.5, 2.5, 3.5, &c.)

- Each track has $0C pages of data in a custom (non-sector-based) format

- We're filling main memory ($0800.. $BFFF), except the two hi-res graphics pages ($2000..$5FFF)

- Nothing in this read loop relies on the checksum in the stack pointer

- $047E exits via $0500

Let's capture it.

# Chapter 5
# In Which The End Is Nigh

```
*9600<C600.C6FFM

; move boot0 to $0900 and patch it up
96F8-   A0 00       LDY   #$00
96FA-   B9 00 08    LDA   $0800,Y
96FD-   99 00 09    STA   $0900,Y
9700-   C8          INY
9701-   D0 F7       BNE   $96FA
9703-   A9 24       LDA   #$24
9705-   8D 5D 09    STA   $095D
9708-   A9 09       LDA   #$09
970A-   8D 70 09    STA   $0970

; set up callback before jumping to
; $0400
970D-   A9 1A       LDA   #$1A
970F-   8D 7E 09    STA   $097E
9712-   A9 97       LDA   #$97
9714-   8D 7F 09    STA   $097F

; start the boot
9717-   4C 01 09    JMP   $0901

; break to the monitor at $047E instead
; of continuing at $0500
971A-   A9 59       LDA   #$59
971C-   8D 7F 04    STA   $047F
971F-   A9 FF       LDA   #$FF
9721-   8D 80 04    STA   $0480

; initialize zero page (copied verbatim
; from $0457)
9724-   A9 00       LDA   #$00
9726-   85 0C       STA   $0C
9728-   85 0D       STA   $0D
972A-   85 0E       STA   $0E
```

```
; continue the boot (skipping over all
; the checksums and other stuff I don't
; care about)
972C-   4C 72 04    JMP   $0472

*BSAVE TRACE2,A$9600,L$12F
*9600G
...reboots slot 6...
<beep>

*2800<800.1FFFM
*C500G
...
]BSAVE BOOT2 0800-1FFF,A$2800,L$1800
]BRUN TRACE2
...reboots slot 6...
<beep>

*2000<6000.9FFFM
*C500G
...
]BSAVE BOOT2 6000-9FFF,A$2000,L$4000
]BRUN TRACE2
...reboots slot 6...
<beep>

*2000<A000.BFFFM
*C500G
...
]BSAVE BOOT2 A000-BFFF,A$2000,L$2000
```

I have the entire game in three files. Victory is in sight. The end is nigh.

# Chapter 6
# In Which The End Is Not Nigh

Continuing from $0500 so I can find the main entry point...

```
]BLOAD BOOT1 0400-07FF,A$2400
]CALL -151

*2500L

; turn off drive motor
2500-   BD 88 C0    LDA   $C088,X

; checksum all of main memory
2503-   A9 08       LDA   #$08
2505-   85 81       STA   $81
2507-   A9 00       LDA   #$00
2509-   85 80       STA   $80
250B-   A8          TAY
250C-   A2 B8       LDX   #$B8
250E-   51 80       EOR   ($80),Y
2510-   C8          INY
2511-   D0 FB       BNE   $250E
2513-   E6 81       INC   $81
2515-   CA          DEX
2516-   D0 F6       BNE   $250E
2518-   A8          TAY

; if anything has been modified, jump
; to The Badlands
2519-   D0 25       BNE   $2540

; initialize some zero page
251B-   A2 00       LDX   #$00
251D-   A0 00       LDY   #$00
251F-   A9 00       LDA   #$00
2521-   85 00       STA   $00
2523-   85 01       STA   $01
```

```
; wait, what?
2525-   FF              ???
2526-   00              BRK
2527-   00              BRK
```

Did I miss a memo? That shouldn't work at all. That should just... crash.

But it doesn't... at least, not on a 6502. The original 6502 (used by the Apple II+ and first revision Apple ][e) had a number of illegal, undocumented instructions. These opcodes actually "worked," in the sense that they did things repeatably and reliably. Many provide unusual addressing modes or weird combinations of two or more other instructions.

Oh, and few, if any, work on a 65C02, so any disk that relied on these undocumented instructions will crash and burn.

Which brings us back to this opcode:

```
2525-   FF              ???
2526-   00              BRK
2527-   00              BRK
```

According to <http://www.ataripreservation.org/websites/freddy.offenga/illopc31.txt>, the $FF opcode is "ISC" a.k.a "ISB" a.k.a. "INS". It functions as a combination of INC and SBC. The $FF variant in particular is a 3-byte instruction that accesses an absolute address + X, like so:

```
2525-   FF 00 00    ISC   $0000,X
```

which, when executed, does this:

```
                    INC   $0000,X
                    SBC   $0000,X
```

Since the accumulator, the X register, and zero page $00 have just been set to zero, this will increment zero page $00 to $01 and subtract that from A. So zero page $00 ends up as $FF, zero page $01 stays at $00, X stays at $00, and A ends up as $FF.

Continuing...

```
; store the result in $00
2528-   85 00       STA   $00

; take the checksum we stashed in the
; stack pointer (at $0471)
252A-   BA          TSX
252B-   8A          TXA

; XOR that with the result of this
; illegal operation
252C-   45 00       EOR   $00
```

```
; and put that somewhere in the middle
; of who knows where
252E-   8D 47 0C    STA   $0C47

; then XOR that with $FF and put that
; in the middle of God knows what
2531-   49 FF       EOR   #$FF
2533-   8D 69 0C    STA   $0C69

; then start the game
2536-   4C 00 BB    JMP   $BB00
```

The checksum in the stack pointer was $01, and zero page $00 is $FF.

```
$01 EOR $FF = $FE --> $0C47
$FE EOR $FF = $01 --> $0C69
```

# Chapter 7
# This Isn't Even My Final Form

So $BB00 starts the game, right? Wrong.

```
*BLOAD BOOT2 A000-BFFF,A$2000
*FE89G FE93G
*A000<2000.3FFFM
*BB00L

BB00-   A6 2B       LDX   $2B

; copies $6200 page to $BE00
BB02-   20 92 BC    JSR   $BC92
BB05-   A9 60       LDA   #$60

; turns on drive motor and waits
BB07-   20 E0 BC    JSR   $BCE0

; advance drive head to phase $17
; (that's a half track -- even phases
; are whole tracks, odd phases are half
; tracks)
BB0A-   A9 17       LDA   #$17
BB0C-   20 48 BB    JSR   $BB48

; Reads a track of data ($0C pages)
; using the same RWTS as the routine
; at $0490. Stores it starting at
; $4000 (based on the accumulator on
; entry).
BB0F-   A9 40       LDA   #$40
BB11-   20 F0 BB    JSR   $BBF0

; advance drive head to phase $19
BB14-   A9 19       LDA   #$19
BB16-   20 48 BB    JSR   $BB48

; read another track, into $4C00..$57FF
BB19-   A9 4C       LDA   #$4C
BB1B-   20 F0 BB    JSR   $BBF0
```

```
; advance drive head to phase $1B
BB1E-   A9 1B       LDA   #$1B
BB20-   20 48 BB    JSR   $BB48

; read another track, into $5800..$63FF
BB23-   A9 58       LDA   #$58
BB25-   20 F0 BB    JSR   $BBF0

; turn off drive motor
BB28-   BD 88 C0    LDA   $C088,X

; checksum everything we just read
BB2B-   A0 00       LDY   #$00
BB2D-   84 00       STY   $00
BB2F-   A9 40       LDA   #$40
BB31-   85 01       STA   $01
BB33-   A2 24       LDX   #$24
BB35-   98          TYA
BB36-   51 00       EOR   ($00),Y
BB38-   C8          INY
BB39-   D0 FB       BNE   $BB36
BB3B-   E6 01       INC   $01
BB3D-   CA          DEX
BB3E-   D0 F6       BNE   $BB36
BB40-   A8          TAY

; branch back to try again if the
; checksum fails
BB41-   D0 C2       BNE   $BB05

BB43-   4C A1 BC    JMP   $BCA1
```

```
*BCA1L

; copy a page back to $6200 (was copied
; there at $BB02)
BCA1-   A0 00       LDY   #$00
BCA3-   B9 00 BE    LDA   $BE00,Y
BCA6-   99 00 62    STA   $6200,Y
BCA9-   88          DEY
BCAA-   D0 F7       BNE   $BCA3

; start the game at the title screen
BCAC-   4C 00 40    JMP   $4000
```

If I replace the illegal instruction at $0525 with an equivalent sequence of instructions, maybe I can get this game to boot on my enhanced //e. Then I can interrupt it at $BCAC and capture the final form of the game code, including the final chunk at $4000.

One small speed bump... the game loader overwrites my boot tracer at $9600, so I'll need to copy the last stage of my trace into the text page instead of jumping back to my code.

```
*9600<C600.C6FFM

; move boot0 to $0900 and patch it up
96F8-   A0 00       LDY   #$00
96FA-   B9 00 08    LDA   $0800,Y
96FD-   99 00 09    STA   $0900,Y
9700-   C8          INY
9701-   D0 F7       BNE   $96FA
9703-   A9 24       LDA   #$24
9705-   8D 5D 09    STA   $095D
9708-   A9 09       LDA   #$09
970A-   8D 70 09    STA   $0970
```

```
; set up callback before jumping to
; $0400
970D-   A9 1A       LDA   #$1A
970F-   8D 7E 09    STA   $097E
9712-   A9 97       LDA   #$97
9714-   8D 7F 09    STA   $097F

; start the boot
9717-   4C 01 09    JMP   $0901

; callback is here
; change a JMP from $0500 to $051B to
; skip over the checksum loop at $0503
971A-   A9 1B       LDA   #$1B
971C-   8D 7F 04    STA   $047F

; set up zero page for initial load
971F-   A9 00       LDA   #$00
9721-   85 0C       STA   $0C
9723-   85 0D       STA   $0D
9725-   85 0E       STA   $0E

; copy the next stage of the trace to
; $0525, where it will be executed
; after the first load is complete but
; before the jump to $BB00
9727-   A0 24       LDY   #$24
9729-   B9 35 97    LDA   $9735,Y
972C-   99 25 05    STA   $0525,Y
972F-   88          DEY
9730-   10 F7       BPL   $9729

; continue the boot (skipping over all
; the checksums and other stuff I don't
; care about)
9732-   4C 72 04    JMP   $0472
```

```
; this code ends up at $0525
; reproduce the illegal opcode with two
; legal ones
9735-   E6 00        INC   $00
9737-   E5 00        SBC   $00

; store the result and do the other
; bit twiddling
9739-   85 00        STA   $00
973B-   A9 01        LDA   #$01
973D-   45 00        EOR   $00
973F-   8D 47 0C     STA   $0C47
9742-   49 FF        EOR   #$FF
9744-   8D 69 0C     STA   $0C69

; now change the JMP at $BCAC to break
; to the monitor instead of starting
; the title screen
9747-   A9 4C        LDA   #$4C
9749-   8D AC BC     STA   $BCAC
974C-   A9 59        LDA   #$59
974E-   8D AD BC     STA   $BCAD
9751-   A9 FF        LDA   #$FF
9753-   8D AE BC     STA   $BCAE

; continue the boot
9756-   4C 00 BB     JMP   $BB00

*BSAVE TRACE4,A$9600,L$159
*9600G
...reboots slot 6...
<beep>
```

(I'm going to re-save all the chunks, since the disk routines that were called from $BB00 modified some of the values in $C00 page and copied some other pages back and forth and I kind of lost track of it all. Anyway, I have the "final" version of the game in memory, so let's just save it all.)

```
*2000<800.1FFFM
*C500G
...
]BSAVE OBJ.0800-1FFF,A$2000,L$1800
]BRUN TRACE4
...reboots slot 6...
<beep>

*C500G
...
]BSAVE OBJ.4000-5FFF,A$4000,L$2000
]BRUN TRACE4
...reboots slot 6...
<beep>

*2000<6000.9FFFM
*C500G
...
]BSAVE OBJ.6000-9FFF,A$2000,L$4000
]BRUN TRACE4
...reboots slot 6...
<beep>

*2000<A000.BFFFM
*C500G
...
]BSAVE OBJ.A000-BFFF,A$2000,L$2000
```

# Chapter 8
# In Which Some Things Are Not Where They Belong

I've finally captured the entire game in memory, in its final form. Let's see if it actually works.

```
]CALL -151

*800:0 N 801<800.BEFEM
*BLOAD OBJ.0800-1FFF,A$800
*BLOAD OBJ.4000-5FFF,A$4000
*BLOAD OBJ.6000-9FFF,A$6000
*BLOAD OBJ.A000-BFFF,A$2000
*FE89G FE93G
*A000<2000.3FFFM
*4000G
```

...shows title screen, I press a key to start the game, it shows "Paddles or Keyboard" screen, then it crashes...

```
057D-    A=01 X=43 Y=23 P=F1 S=E6
```

Oh no. It's calling back to the RWTS that was on the text page (but isn't anymore). But the original disk doesn't do any more disk access at this point, so I'm guessing this is pure copy protection. But who knows.

Let's try the old "fill the text page with RTS" trick.

```
*C500G
...
]CALL -151
*BLOAD OBJ.0800-1FFF,A$800
*BLOAD OBJ.4000-5FFF,A$4000
*BLOAD OBJ.6000-9FFF,A$6000
*BLOAD OBJ.A000-BFFF,A$2000
*FE89G FE93G
*A000<2000.3FFFM
*400:60 N 401<400.7FEM N 4000G
...works...
```

Aha! It is pure copy protection. But where exactly is it calling? I mean, I could just fill the text page with $60, but then I'd never know why it worked and whether I'd missed something important.

To narrow it down, let's go back to the RWTS on the text page and see what code is near $057D.

```
]PR#5
...
]BLOAD BOOT1 0400-07FF,A$2400
]CALL -151
...
```

```
*256AL

256A-   A6 2B       LDX   $2B
256C-   CA          DEX
256D-   8A          TXA
256E-   4A          LSR
256F-   4A          LSR
2570-   4A          LSR
2571-   4A          LSR
2572-   09 C0       ORA   #$C0
2574-   48          PHA
2575-   A9 FF       LDA   #$FF
2577-   48          PHA
2578-   60          RTS
2579-   00          BRK
257A-   00          BRK
257B-   00          BRK
257C-   00          BRK
257D-   00          BRK
257E-   00          BRK
257F-   00          BRK
2580-   00          BRK
```

This is the final part of The Badlands. It reboots by pushing the boot slot (minus 1) on the stack. But that's not the part we're calling now. And there's nothing beyond $0578 except BRK, but it never set a BRK handler, so it's not calling that.

I bet it's calling the RTS. At $0578. It doesn't want to *do* anything. It just wants to make sure the bootloader is still on the text page where it belongs.

Let's test that theory.

```
*C500G
...
]CALL -151
*BLOAD OBJ.0800-1FFF,A$800
*BLOAD OBJ.4000-5FFF,A$4000
*BLOAD OBJ.6000-9FFF,A$6000
*BLOAD OBJ.A000-BFFF,A$2000
*FE89G FE93G
*A000<2000.3FFFM
*400:0 N 401<400.7FEM N 578:60 N 4000G
...works...
```

Turning to my trusty Copy ][+ sector editor (version 5.5, the last version that can "follow" files and scan for byte sequences within them), I scanned each of the OBJ.* files for the byte sequence "4C 78 05" (JMP $0578). No matches. Then I scanned for "20 78 05" (JSR $0578) and hit the jackpot.

For posterity, here are all the places within the game that rely on a single "RTS" being at $0578:

```
]PR#5
...
]CALL -151
*BLOAD OBJ.0800-1FFF,A$800
*BLOAD OBJ.4000-5FFF,A$4000
*BLOAD OBJ.6000-9FFF,A$6000
*BLOAD OBJ.A000-BFFF,A$2000
*FE89G FE93G
*A000<2000.3FFFM
```

```
*1FEFL

1FEF-   91 6B       STA   ($6B),Y
1FF1-   88          DEY
1FF2-   88          DEY
1FF3-   E8          INX
1FF4-   C6 6A       DEC   $6A
1FF6-   D0 E2       BNE   $1FDA
1FF8-   20 78 05    JSR   $0578   <-- !
1FFB-   60          RTS

*66BDL

66BD-   A9 EA       LDA   #$EA
66BF-   8D E4 62    STA   $62E4
66C2-   20 78 05    JSR   $0578   <-- !
66C5-   4C 00 BB    JMP   $BB00
```

(I think this one is called just after the game ends, since it's jumping back to $BB00.)

```
*BC83L

BC83-   20 78 05    JSR   $0578   <-- !
BC86-   AD 00 C0    LDA   $C000
BC89-   60          RTS
```

Rather than patch each of these, I'm going to set $0578 to $60 during boot, so I don't need to patch the game code at all.

# Chapter 9
# If You Wish To Play A Game, You Must First Create The Universe

The original disk starts with an animated title screen, then returns to it after the game ends. In between, it throws away those assets to make room for game assets, then re-reads several tracks from disk (at $BB00) when it wants to return to the title screen. Classic cracks either didn't include the title screen at all or let it run once then never returned to it.

It's 2015. We can do better.

When in doubt, assume more RAM. In this case, I'm going to assume 64K instead of 48K, a.k.a. a language card with an additional 16K in two banks at $D000. This will allow me to load the entire game in one shot, then stash the assets for the title screen and bring them back as needed -- without re-reading them from disk.

To this end, I added two routines to replace the disk access at $BB00. The first is at $BB40; it's called once during boot, after everything is read from disk. (We'll get to the custom bootloader in a minute.)

```
; turn off drive motor (X register will
; still be the boot slot x16 here)
BB40-   BD 88 C0    LDA   $C088,X

; write to RAM bank 2 in language card
BB43-   AD 81 C0    LDA   $C081
BB46-   AD 81 C0    LDA   $C081
```

```
; copy $4000..$5FFF to RAM bank 2
; (this is only run once, so I can use
; self-modifying code without re-
; initializing the addresses)
BB49-   A2 20       LDX   #$20
BB4B-   A0 00       LDY   #$00
BB4D-   B9 00 40    LDA   $4000,Y
BB50-   99 00 D0    STA   $D000,Y
BB53-   C8          INY
BB54-   D0 F7       BNE   $BB4D
BB56-   EE 4F BB    INC   $BB4F
BB59-   EE 52 BB    INC   $BB52
BB5C-   CA          DEX
BB5D-   D0 EE       BNE   $BB4D

; switch back to ROM
BB5F-   AD 82 C0    LDA   $C082

; set up the magical "RTS" in the text
; page (called from several places in
; the game code)
BB62-   A9 60       LDA   #$60
BB64-   8D 78 05    STA   $0578

; set up a friendly reset vector (will
; restart the game at the title screen)
BB67-   A9 00       LDA   #$00
BB69-   8D F2 03    STA   $03F2
BB6C-   A9 BB       LDA   #$BB
BB6E-   8D F3 03    STA   $03F3
BB71-   49 A5       EOR   #$A5
BB73-   8D F4 03    STA   $03F4

; start the game at the title screen
BB76-   4C 00 40    JMP   $4000
```

The other custom routine is at $BB00, which is called after the game displays the "GAME OVER" screen. On the original disk, this routine would reload the title screen assets from disk, but now I'll copy them from RAM bank 2 instead.

```
*BB00L

; read from RAM bank 2
BB00-   AD 80 C0    LDA   $C080

; initialize addresses in the following
; loop
BB03-   A9 D0       LDA   #$D0
BB05-   8D 13 BB    STA   $BB13
BB08-   A9 40       LDA   #$40
BB0A-   8D 16 BB    STA   $BB16

; copy $D000..$FFFF to $4000..$5FFF
BB0D-   A2 20       LDX   #$20
BB0F-   A0 00       LDY   #$00
BB11-   B9 00 D0    LDA   $D000,Y
BB14-   99 00 40    STA   $4000,Y
BB17-   C8          INY
BB18-   D0 F7       BNE   $BB11
BB1A-   EE 13 BB    INC   $BB13
BB1D-   EE 16 BB    INC   $BB16
BB20-   CA          DEX
BB21-   D0 EE       BNE   $BB11

; switch back to ROM
BB23-   AD 82 C0    LDA   $C082
```

```
; Wait. The game is still displaying
; the "GAME OVER" screen here. The
; original disk only displayed it as
; long as it took to re-read the title
; screen assets from disk, but copying
; from memory is obviously much faster.
BB26-    A9 00          LDA   #$00
BB28-    20 A8 FC       JSR   $FCA8
BB2B-    20 A8 FC       JSR   $FCA8
BB2E-    20 A8 FC       JSR   $FCA8
BB31-    20 A8 FC       JSR   $FCA8
BB34-    20 A8 FC       JSR   $FCA8

; start the game at the title screen
BB37-    4C 00 40       JMP   $4000
```

With those custom routines in place, I can turn to recreating the game disk itself. The original disk loads code into $0800..$1FFF, $6000..$BFFF, and eventually $4000..$5FFF. There's a hole in there, in hi-res graphics page 1 ($2000..$3FFF), which is never loaded or initialized.

My custom bootloader works best with loading entire tracks (16 pages) at a time, so here's the plan:

| track | data |
|---|---|
| 01 | $0800..$17FF |
| 02 | $1800..$1FFF + 8 unused sectors |
| 03 | $4000..$4FFF |
| 04 | $5000..$5FFF |
| 05 | $6000..$6FFF |
| 06 | $7000..$7FFF |
| 07 | $8000..$8FFF |
| 08 | $9000..$9FFF |
| 09 | $A000..$AFFF |
| 0A | $B000..$BFFF |

Thus:

```
]PR#5
...
]BLOAD OBJ.0800-1FFF,A$800
]BLOAD OBJ.4000-5FFF,A$2800
]BLOAD OBJ.6000-9FFF,A$4800
]BLOAD OBJ.A000-BFFF,A$8800
```

Now I have all the game assets in memory, starting at $0800, aligned with the tracks they will occupy on disk.

```
[S6,D1=blank formatted disk]
[S5,D1=my work disk]

]PR#5
```

```
]CALL -151

; page count (decremented)
0300-    A9 A0          LDA    #$A0
0302-    85 FF          STA    $FF

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

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

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

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

; increment page to write
0324-    EE 91 03       INC    $0391

; loop until done with all $90 pages
0327-    C6 FF          DEC    $FF
0329-    D0 DD          BNE    $0308
032B-    60             RTS
```

```
; logical to physical sector mapping
*340.34F

0340- 00 07 0E 06 0D 05 0C 04
0348- 0B 03 0A 02 09 01 08 0F

; RWTS parameter table, pre-initialized
; with slot 6, drive 1, track $01,
; sector $00, address $0800, and RWTS
; write command ($02)
*388.397

0388- 01 60 01 00 01 00 FB F7
0390- 00 08 00 00 02 00 00 60

*BSAVE MAKE,A$300,L$98

*300G
```

Now I have the entire game on tracks $01-$0A of a standard format disk.

# Chapter 10
# 4boot for the win

The bootloader, which I've named 4boot, lives on track 0. Sector 0 is boot0; it reuses the disk controller ROM routine to load boot1 (sectors $0C-$0E) into $3D00..$3FFF. The game takes up $0800..$1FFF and $4000..$BFFF, so boot1 needs to go in the hole in the middle.

Boot0 looks like this:

```
; decrement sector count
0801-   CE 19 08     DEC   $0819

; branch once we've read enough sectors
0804-   30 12        BMI   $0818

; increment physical sector to read
0806-   E6 3D        INC   $3D

; set page to save sector data
0808-   A9 3F        LDA   #$3F
080A-   85 27        STA   $27

; decrement page
080C-   CE 09 08     DEC   $0809

; $0880 is a sparse table of $C1..$C6,
; so this sets up the proper jump to
; the disk controller ROM based on the
; slot number
080F-   BD 80 08     LDA   $0880,X
0812-   8D 17 08     STA   $0817

; read a sector (exits via $0801)
0815-   4C 5C 00     JMP   $005C
```

```
; sector read loop exits to here (from
; $0804) -- note: by the time execution
; reaches here, $0819 is $FF, so this
; just resets the stack
0818-   A2 03       LDX   #$03
081A-   9A          TXS

; set up zero page (used by RWTS) and
; push an array of addresses to the
; stack at the same time
081B-   A2 0F       LDX   #$0F
081D-   BD 80 08    LDA   $0880,X
0820-   95 F0       STA   $F0,X
0822-   48          PHA
0823-   CA          DEX
0824-   D0 F7       BNE   $081D

; display the hi-res graphics page
; (uninitialized, like the original)
0826-   2C 54 C0    BIT   $C054
0829-   2C 57 C0    BIT   $C057
082C-   2C 52 C0    BIT   $C052
082F-   2C 50 C0    BIT   $C050
0832-   60          RTS
```

Here are the addresses that are pushed to the stack:

```
*881.88F

0880-    88 FE 92 FE FF 3C 7B
0888- 3E 3F BB 08 02 00 00 00
```

These bytes are pushed on the stack in reverse order, starting with $088F. When we hit the "RTS" at $0832, it pops the stack and jumps to $FE89, $FE93, $3D00, $3E7C, and $BB40. Each of these routines exits via RTS and "returns" to the next address (+1) that we pushed on the stack.

- $FE89 and $FE93 are in ROM (IN#0 and PR#0)
- $3D00 is the RWTS entry point. It reads tracks $01..$02 into $0800..$27FF. These values are stored in zero page, which we just set. Since we set up the entire stack chain in advance, we can safely overwrite the boot0 code at $0800 with data from disk. We're never going back to boot0.
- $3E7C is a small routine that sets up the second multi-track read, tracks $03..$0A into $4000..$BFFF, and jumps to $3D00 to execute it.
- $BB40 is the game initialization routine I wrote (see below). It never returns, so the other values on the stack are irrelevant.

The RWTS at $3D00 is derived from the ProDOS RWTS. It uses in-place nibble decoding to avoid extra memory copying, and it uses "scatter reads" to read whatever sector is under the drive head when it's ready to load something.

```
*3D00L

; set up some places later in the RWTS
; where we need to read from a slot-
; specific data latch
3D00-   A6 2B       LDX   $2B
3D02-   8A          TXA
3D03-   09 8C       ORA   #$8C
3D05-   8D 96 3D    STA   $3D96
3D08-   8D AD 3D    STA   $3DAD
3D0B-   8D C3 3D    STA   $3DC3
3D0E-   8D D7 3D    STA   $3DD7
3D11-   8D EC 3D    STA   $3DEC

; advance drive head to next track
3D14-   20 53 3E    JSR   $3E53

; sectors-left-to-read-on-this-track
; counter
3D17-   A0 0F       LDY   #$0F
3D19-   84 F8       STY   $F8

; Initialize array at $0100 that tracks
; which sectors we've read from the
; current track. The array is in
; physical sector order, thus the RWTS
; assumes data is stored in physical
; sector order on each track. Values
; are the actual pages in memory where
; that sector should go, and they get
; zeroed once the sector is read.
3D1B-   98          TYA
3D1C-   18          CLC
3D1D-   65 FB       ADC   $FB
3D1F-   99 00 01    STA   $0100,Y
3D22-   88          DEY
3D23-   10 F6       BPL   $3D1B
```

```
; find the next address prologue and
; store the address field in $2C..$2F,
; like DOS 3.3
3D25-   20 0F 3E    JSR   $3E0F

; check if this sector has been read
3D28-   A4 2D       LDY   $2D
3D2A-   B9 00 01    LDA   $0100,Y

; if 0, we've read this sector already,
; so loop back and look for another
3D2D-   F0 F6       BEQ   $3D25

; if not 0, use the target page and set
; up some STA instructions in the RWTS
; so we write this sector directly to
; its intended page in memory
3D2F-   A8          TAY
3D30-   84 FF       STY   $FF
3D32-   8C EA 3D    STY   $3DEA
3D35-   A5 FE       LDA   $FE
3D37-   8D E9 3D    STA   $3DE9
3D3A-   38          SEC
3D3B-   E9 54       SBC   #$54
3D3D-   8D D1 3D    STA   $3DD1
3D40-   B0 02       BCS   $3D44
3D42-   88          DEY
3D43-   38          SEC
3D44-   8C D2 3D    STY   $3DD2
3D47-   E9 57       SBC   #$57
3D49-   8D AA 3D    STA   $3DAA
3D4C-   B0 01       BCS   $3D4F
3D4E-   88          DEY
3D4F-   8C AB 3D    STY   $3DAB

; read the sector into memory
3D52-   20 6D 3D    JSR   $3D6D
```

```
; if that failed, just loop back and
; look for another sector
3D55-   B0 CE        BCS   $3D25

; mark this sector as read
3D57-   A4 2D        LDY   $2D
3D59-   A9 00        LDA   #$00
3D5B-   99 00 01     STA   $0100,Y
3D5E-   E6 FB        INC   $FB

; decrement sectors-left-to-read-on-
; this-track counter
3D60-   C6 F8        DEC   $F8

; loop until we've read all the sectors
; on this track
3D62-   10 C1        BPL   $3D25

; decrement tracks-left-to-read counter
; (set in boot0)
3D64-   C6 FC        DEC   $FC

; loop until we've read all the tracks
3D66-   D0 AC        BNE   $3D14

; exit via RTS (SEC is irrelevant here;
; it's used by the following routine
; (not shown) to indicate an error
; finding the data prologue)
3D6B-   38           SEC
3D6C-   60           RTS
```

The combination of

- code on consecutive tracks starting on track $01 (minimizes drive head movement)
- scatter reads (minimizes disk movement per track)
- in-place denibblizing and no copy protection (minimizes memory copies and checksum loops)

means the entire boot process takes about three seconds.

Quod erat liberandum.

## Changelog

2015-09-06

- typos [thanks qkumba]

2015-05-24

- initial release

EOF