Read 'N Roll

2015-05-04

Contents

---------------Read 'N Roll--------------
A 4am crack   2015-05-04

Name: Read 'N Roll
Version: 1.0
Genre: educational
Year: 1988
Authors:
  - Jan Davidson
  - Julie Baumgartner
  - J. M. Albanese
  - L. X. Savain
  - T. S. DeBry
Publisher: Davidson & Associates, Inc.
Media: two double-sided 5.25-inch disks
OS: ProDOS 1.4
Other versions: none (preserved here
  for the first time)

Only disk 1, side A is bootable, so
I'll start there.

Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

COPYA
  disk read error on last pass

Locksmith Fast Disk Backup
  read error on T22,S00; copy boots to
  ProDOS then quits to program selector

EDD 4 bit copy (no sync, no count)
  works

Copy][+ nibble editor
  can't find any evidence that T22,S00
  even exists

Disk Fixer
  can't find any combination of
  parameters that can read T22,S00

Why didn't COPYA work?
  intentionally bad sector on T22

Why didn't Locksmith FDB work?
  Probably a nibble check in the first
  SYSTEM file that reads the unreadable
  sector on T22

Next steps:

  1. Trace the first .SYSTEM file
  2. Find nibble check and disable it
  3. There is no step 3 (I hope)

Chapter 1
In Which We Learn Much About Things That Ultimately Prove Irrelevant, And Very Little Else

```
[S6,D1=original disk 1A]

]PR#7
]CAT,S6,D1

/RNR

 NAME            TYPE  BLOCKS  MODIFIED

 PRODOS           SYS      32  <NO DATE>
 RNR.SYSTEM       SYS       3  <NO DATE>
 TK.ABS           BIN      28  <NO DATE>
 TEST.FONT        BIN       7  <NO DATE>
 RNR              BIN      38  <NO DATE>
 ENTRY.PIC        TXT       7  <NO DATE>
 IMGS             TXT      14  <NO DATE>
 DP0              TXT       9  <NO DATE>
 MO0              TXT      48  17-FEB-88
 PRINTER.DRIVERS  TXT      12  <NO DATE>
 INTER.DRIVERS    TXT       5  <NO DATE>
 DUMP.HIRES.R     BIN       5  <NO DATE>
 CERTPIC          TXT       8  <NO DATE>
 PRINTER.DATA     TXT       1   1-JAN-88
 CTEXT            TXT       4  <NO DATE>
 CE0              TXT      17  16-FEB-88
 QUIT             BIN       1  <NO DATE>
 PR0              TXT       3  16-FEB-88

BLOCKS FREE:    23     BLOCKS USED:  257

]PREFIX /RNR
]-RNR.SYSTEM
...works...
```

OK, I can boot from my hard drive, then run the program successfully from the original disk. Whatever copy protection there is, it's not dependent on the PRODOS file.

```
[S6,D1=non-working Locksmith copy]

]PR#7
]PREFIX /RNR
]-RNR.SYSTEM
...quits via ProDOS quit handler...

Time to start tracing RNR.SYSTEM.

]PR#7
]PREFIX /RNR
]BLOAD RNR.SYSTEM,A$2000,TSYS
]CALL -151
*2000L

2000-   A2 00       LDX   #$00
2002-   BD 1A 20    LDA   $201A,X
2005-   9D 01 08    STA   $0801,X
2008-   BD 1A 21    LDA   $211A,X
200B-   9D 01 09    STA   $0901,X
200E-   BD 1A 22    LDA   $221A,X
2011-   9D 01 0A    STA   $0A01,X
2014-   E8          INX
2015-   D0 EB       BNE   $2002
2017-   4C 01 08    JMP   $0801

*2017:60
*2000G
*801L

0801-   4C 52 08    JMP   $0852
```

```
$852L
.
. ProDOS-y initialization
.
; subroutine gets file info via ProDOS
; MLI (command $C4) -- filename is at
; the address pointed to by (X/A)
088C-   A9 6A       LDA   #$6A
088E-   A2 0A       LDX   #$0A
0890-   20 D3 09    JSR   $09D3

The string at $0A6A is "RNR", so this
is operating on the RNR.BIN file (not
to be confused with RNR.SYSTEM, which
is the program we're running right now
that was auto-executed by ProDOS, nor
the disk itself, which is also named
RNR.)

; open the "RNR" file (MLI $C8), but at
; $4060 instead of its default starting
; address ($0860, according to an
; extended catalog listing)
0893-   A9 60       LDA   #$60
0895-   8D 95 0A    STA   $0A95
0898-   A9 40       LDA   #$40
089A-   8D 96 0A    STA   $0A96
089D-   20 E8 09    JSR   $09E8

; get file EOF (MLI $D1)
08A0-   20 10 0A    JSR   $0A10
```

```
; read part of the file (MLI $CA)
08A3-   A9 A0        LDA   #$A0
08A5-   85 64        STA   $64
08A7-   A9 17        LDA   #$17
08A9-   85 65        STA   $65
08AB-   AD 92 0A     LDA   $0A92
08AE-   8D 38 09     STA   $0938
08B1-   AD 93 0A     LDA   $0A93
08B4-   8D 39 09     STA   $0939
08B7-   20 2E 0A     JSR   $0A2E

; move to auxiliary memory
08BA-   20 5D 09     JSR   $095D

*95DL

; copy to aux memory
095D-   A9 60        LDA   #$60
095F-   85 3C        STA   $3C
0961-   85 42        STA   $42
0963-   A9 40        LDA   #$40
0965-   85 3D        STA   $3D
0967-   A9 08        LDA   #$08
0969-   85 43        STA   $43
096B-   A9 00        LDA   #$00
096D-   85 3E        STA   $3E
096F-   A9 58        LDA   #$58
0971-   85 3F        STA   $3F
0973-   38           SEC
0974-   4C 11 C3     JMP   $C311
```

According to "Inside the Apple //e" (pp. 296-8), $C311 copies data from main memory to aux memory and back. (Aux memory is what you get by having an 80-column card, 128K instead of 64.)

```
The routine itself takes 4 parameters:

  ($3C/$3D) starting address
  ($3E/$3F) ending address
  ($42/$43) destination address in the
            other memory bank
  carry bit set for main->aux copy, or
            clear for aux->main copy

So this is copying $4060..$5800 in
main memory (from the "RNR" file we
just read) to $0860 in aux memory.

*88DL

; set the mark within the open RNR file
; (MLI $CE)
08BD-   20 3A 09    JSR   $093A

; read more of the file (MLI $CA)
08C0-   A9 00       LDA   #$00
08C2-   85 60       STA   $60
08C4-   85 64       STA   $64
08C6-   A9 40       LDA   #$40
08C8-   85 61       STA   $61
08CA-   A9 50       LDA   #$50
08CC-   85 65       STA   $65
08CE-   20 2E 0A    JSR   $0A2E
```

```
; and copy that part to aux memory
08D1-   A9 00       LDA   #$00
08D3-   85 3C       STA   $3C
08D5-   85 42       STA   $42
08D7-   85 3E       STA   $3E
08D9-   A9 40       LDA   #$40
08DB-   85 3D       STA   $3D
08DD-   85 43       STA   $43
08DF-   A9 90       LDA   #$90
08E1-   85 3F       STA   $3F
08E3-   38          SEC
08E4-   20 11 C3    JSR   $C311

; check if we're at the end of the file
; yet (I assume because this is a
; generic routine) -- on this disk it
; always returns #$FF
08E7-   20 22 09    JSR   $0922

; will not branch
08EA-   F0 2D       BEQ   $0919

; read more from the file
08EC-   A9 20       LDA   #$20
08EE-   85 65       STA   $65
08F0-   A9 00       LDA   #$00
08F2-   85 60       STA   $60
08F4-   A9 FF       LDA   #$FF
08F6-   85 64       STA   $64
08F8-   A9 40       LDA   #$40
08FA-   85 61       STA   $61
08FC-   20 2E 0A    JSR   $0A2E
```

```
; and copy it to aux memory
08FF-   A9 00        LDA   #$00
0901-   85 3C        STA   $3C
0903-   85 42        STA   $42
0905-   A9 FF        LDA   #$FF
0907-   85 3E        STA   $3E
0909-   A9 40        LDA   #$40
090B-   85 3D        STA   $3D
090D-   A9 90        LDA   #$90
090F-   85 43        STA   $43
0911-   A9 60        LDA   #$60
0913-   85 3F        STA   $3F
0915-   38           SEC
0916-   20 11 C3     JSR   $C311

; close the file (MLI $CC)
0919-   20 56 0A     JSR   $0A56

; copy more bits to aux memory
091C-   20 77 09     JSR   $0977
091F-   4C 8F 09     JMP   $098F

*98FL

; copy even more bits to aux memory
098F-   A9 00        LDA   #$00
0991-   85 3C        STA   $3C
0993-   85 42        STA   $42
0995-   A9 08        LDA   #$08
0997-   85 3D        STA   $3D
0999-   85 43        STA   $43
099B-   A9 5F        LDA   #$5F
099D-   85 3E        STA   $3E
099F-   A9 08        LDA   #$08
09A1-   85 3F        STA   $3F
09A3-   38           SEC
09A4-   20 11 C3     JSR   $C311
```

```
; subroutine to read an entire file --
; filename is at the address pointed to
; by (X/A), $0A6E, which is "TK.ABS"
09A7-   A9 6E       LDA   #$6E
09A9-   A2 0A       LDX   #$0A
09AB-   20 C4 09    JSR   $09C4

; read file "TEST.FONT"
09AE-   A9 75       LDA   #$75
09B0-   A2 0A       LDX   #$0A
09B2-   20 C4 09    JSR   $09C4

; set up a jump to AUX memory (see
; "Inside the Apple //e", p. 300)
; jump address ($0860) is stored in
; $03ED/$03EE
09B5-   A9 60       LDA   #$60
09B7-   8D ED 03    STA   $03ED
09BA-   A9 08       LDA   #$08
09BC-   8D EE 03    STA   $03EE

; use main memory's stack and zero page
09BF-   B8          CLV

; transfer control from main to aux mem
09C0-   38          SEC

; call XFER routine
09C1-   4C 14 C3    JMP   $C314

*9C1:4C 59 FF
*801G
<beep>
```

Execution reaches here, even on my non-working copy. I haven't found the copy protection routine yet.

Chapter 2
In Which We Make A Soul-Crushing Discovery And Almost Give Up

Execution continues in auxiliary memory at $0860 (set up at $09B5). This code was originally loaded from the "RNR" file at $4060 then moved to aux memory. Patching it may be tricky (if it comes to that), but I can load it up in main memory and take a look.

```
*BLOAD RNR,A$860
*860L

0860-   4C BE 18    JMP   $18BE

*18BEL

; zero page initialization
18BE-   A9 19       LDA   #$19
18C0-   85 19       STA   $19
18C2-   A9 68       LDA   #$68
18C4-   85 18       STA   $18

; check for 80-column card using bit 1
; of the machine ID (set by ProDOS)
18C6-   A9 02       LDA   #$02
18C8-   2C 98 BF    BIT   $BF98
18CB-   F0 05       BEQ   $18D2

; switch to 80-column mode
18CD-   A9 00       LDA   #$00
18CF-   20 00 C3    JSR   $C300

; is DELETE key held down on boot?
18D2-   AD 00 C0    LDA   $C000
18D5-   AC 10 C0    LDY   $C010
18D8-   C9 FF       CMP   #$FF
```

```
; if yes, branch over a bunch of
; stuff (I tried this and the program
; crashed -- maybe it hooks into some
; external routines that the developers
; had installed?)
18DA-   F0 6E       BEQ   $194A

; other initialization
18DC-   A0 FF       LDY   #$FF
18DE-   84 32       STY   $32
18E0-   AD DF 0E    LDA   $0EDF
18E3-   8D EA 0E    STA   $0EEA
18E6-   AD E0 0E    LDA   $0EE0
18E9-   8D EB 0E    STA   $0EEB
18EC-   A9 00       LDA   #$00
18EE-   A2 11       LDX   #$11
18F0-   9D A3 08    STA   $08A3,X
18F3-   CA          DEX
18F4-   10 FA       BPL   $18F0

; $0885 is a wrapper around the ProDOS
; MLI, and this command ($CC) closes
; all open files and flushes everything
18F6-   A9 CC       LDA   #$CC
18F8-   8D 8A 08    STA   $088A
18FB-   E8          INX
18FC-   8E 92 08    STX   $0892
18FF-   E8          INX
1900-   8E 91 08    STX   $0891
1903-   20 85 08    JSR   $0885

; set the Ctrl-Y vector
1906-   A9 4C       LDA   #$4C
1908-   8D F8 03    STA   $03F8
190B-   A9 4A       LDA   #$4A
190D-   8D F9 03    STA   $03F9
1910-   A9 19       LDA   #$19
1912-   8D FA 03    STA   $03FA
```

```
; set reset vector
1915-   A9 BE       LDA   #$BE
1917-   8D F2 03    STA   $03F2
191A-   A9 18       LDA   #$18
191C-   8D F3 03    STA   $03F3
191F-   20 6F FB    JSR   $FB6F

; clear system bitmap (tracks which
; memory pages are in use)
1922-   A9 00       LDA   #$00
1924-   A2 17       LDX   #$17
1926-   9D 58 BF    STA   $BF58,X
1929-   CA          DEX
192A-   10 FA       BPL   $1926
192C-   AD 84 08    LDA   $0884
192F-   8D FD BF    STA   $BFFD
1932-   AD 6C 08    LDA   $086C
1935-   8D 38 18    STA   $1838
1938-   AD 6D 08    LDA   $086D
193B-   8D 39 18    STA   $1839
193E-   A0 17       LDY   #$17
1940-   D0 0A       BNE   $194C

;[always skipped]
;1942-   A9 18       LDA   #$18
;1944-   85 19       STA   $19
;1946-   A9 9B       LDA   #$9B
;1948-   85 18       STA   $18
;194A-   A0 0F       LDY   #$0F
```

```
; memory move
194C-   AD 70 08     LDA   $0870
194F-   85 86        STA   $86
1951-   AD 71 08     LDA   $0871
1954-   85 87        STA   $87
1956-   B9 6C 08     LDA   $086C,Y
1959-   91 86        STA   ($86),Y
195B-   88           DEY
195C-   10 F8        BPL   $1956
195E-   D8           CLD
195F-   A9 6C        LDA   #$6C
1961-   85 1A        STA   $1A
1963-   4C 0D 0C     JMP   $0C0D

*C0DL

; stack fiddling
0C0D-   86 88        STX   $88
0C0F-   A0 08        LDY   #$08
0C11-   B1 86        LDA   ($86),Y
0C13-   AA           TAX
0C14-   9A           TXS
0C15-   A6 88        LDX   $88
0C17-   4C D5 08     JMP   $08D5

*8D5L

; ah, ($18) is an address, initialized
; as $1968 (at $18BE)
08D5-   A0 01        LDY   #$01

; get the word at ($18) and put it in
; zero page $1B and $1C
08D7-   B1 18        LDA   ($18),Y
08D9-   85 1C        STA   $1C
08DB-   88           DEY
08DC-   B1 18        LDA   ($18),Y
08DE-   85 1B        STA   $1B
```

```
; increment ($18) by 2
08E0-   18          CLC
08E1-   A5 18       LDA   $18
08E3-   69 02       ADC   #$02
08E5-   85 18       STA   $18
08E7-   90 02       BCC   $08EB
08E9-   E6 19       INC   $19

; zero page $1A is $6C (set at $195F),
; so this is all an indirect jump to
; the address listed at $1968
08EB-   4C 1A 00    JMP   $001A

*1968.1969
1968- 1D 1B

*1B1D.1B1E
1B1D- F2 0D

Execution continues at $0DF2.

*DF2L

; fiddling with the addresses again
; ($1B) and ($18)
0DF2-   A5 19       LDA   $19
0DF4-   48          PHA
0DF5-   A5 18       LDA   $18
0DF7-   48          PHA
0DF8-   18          CLC
0DF9-   A5 1B       LDA   $1B
0DFB-   69 02       ADC   #$02
0DFD-   85 18       STA   $18
0DFF-   98          TYA
0E00-   65 1C       ADC   $1C
0E02-   85 19       STA   $19
```

```
; and jump back to...$08D5
0E04-   4C D5 08    JMP   $08D5
```

Oh no. I just figured this out. This is not the code I'm looking for. This is the code that interprets the code I'm looking for. This is an interpreter.

Now what?

I've had some limited success with reverse engineering Pascal p-code, but that was

1. painful,
2. only possible because p-code is well known and meticulously documented, and
3. still painful

I don't even know what language this "code" is written in.

Let's regroup.

Chapter 3
In Which We Regroup, Take Stock, And Continue From First Principles

Here's what I know:

* This disk is copyable with EDD bit copy, so there is probably no fancy nibble check that relies on timing bits or desynchronized nibbles.

* The program runs (from the original disk) even if I first boot ProDOS from my hard drive. That's not even the same version of ProDOS as the original disk.

* Other disks by this publisher (e.g. Spell It Plus!, crack no. 201), have similar disk characteristics (bad sector on track $22) and behavior (copies quit via ProDOS program selector).

* Under ProDOS, "quitting via the program selector" and "reading a raw sector" are accomplished by sending commands to the ProDOS MLI.

* All calls to the ProDOS MLI route through the same entry point ($BF00).

This gives me an idea. Since I can run this program even after booting ProDOS from my hard drive, maybe I can also install a little logging program that lets me see ProDOS MLI calls as they whiz by.

```
]PR#7
]CALL -151
```

```
*BF00L

BF00-   4C 4B BF    JMP   $BF4B
```

OK, at the end of my routine, I'll need to jump to $BF4B to get to the real MLI entry point.

```
; get, print, and store the top address
; on the stack
0100-   BA          TSX

; high byte
0101-   BD 02 01    LDA   $0102,X
0104-   8D 22 01    STA   $0122
0107-   20 DA FD    JSR   $FDDA

; low byte
010A-   BD 01 01    LDA   $0101,X
010D-   8D 21 01    STA   $0121
0110-   20 DA FD    JSR   $FDDA

; print a space
0113-   A9 A0       LDA   #$A0
0115-   20 F0 FD    JSR   $FDF0

; increment the address we got earlier
; (and put in $0121/$0122)
0118-   EE 21 01    INC   $0121
011B-   D0 03       BNE   $0120
011D-   EE 22 01    INC   $0122
```

```
; this line of code will now load the
; MLI command that was passed to the
; ProDOS MLI (it's the byte immediately
; after the JSR to the MLI entry point)
0120-   AD FF FF    LDA   $FFFF

; print that too
0123-   20 DA FD    JSR   $FDDA

; print a carriage return
0126-   A9 8D       LDA   #$8D
0128-   20 F0 FD    JSR   $FDF0

; wait for a key
012B-   2C 10 C0    BIT   $C010
012E-   AD 00 C0    LDA   $C000
0131-   10 FB       BPL   $012E

; jump to real MLI
0133-   4C 4B BF    JMP   $BF4B

*PREFIX /RNR
*BLOAD RNR.SYSTEM,A$2000,TSYS

; route all MLI calls through my logger
*BF00:4C 00 01

; run the program
*2000G
```

I've annotated the following output with the name of each MLI command. The logger only outputs the two columns of hex (stack address + command ID).

```
09E0 C4           ; GET_FILE_INFO
0A03 C8           ; OPEN
0A1C D1           ; GET_EOF
0A4E CA           ; READ
0955 CE           ; SET_MARK
0A4E CA           ; READ
0A62 CC           ; CLOSE
09E0 C4           ; GET_FILE_INFO
0A03 C8           ; OPEN
0A1C D1           ; GET_EOF
0A4E CA           ; READ
0A62 CC           ; CLOSE
09E0 C4           ; GET_FILE_INFO
0A03 C8           ; OPEN
0A1C D1           ; GET_EOF
0A4E CA           ; READ
0A62 CC           ; CLOSE
```

[...clears screen in 80-column mode and continues...]

```
0889 CC           ; CLOSE
0889 C8           ; OPEN
0889 D1           ; GET_EOF
0889 CA           ; READ
0889 CD           ; FLUSH
0889 CC           ; CLOSE
0889 80           ; READ_BLOCK     <-- !
0889 C8           ; OPEN
0889 D1           ; GET_EOF
0889 CA           ; READ
0889 CD           ; FLUSH
0889 CC           ; CLOSE
4059 65           ; QUIT
```

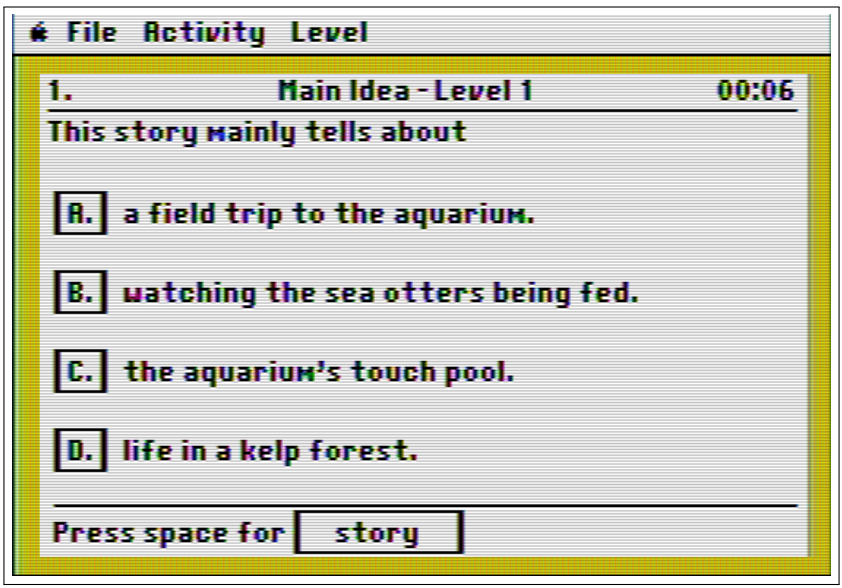
Several interesting things here. I've stepped through some of the initial commands already, so they come as no surprise. I know that it loads three files, "RNR", "TK.ABS", and "TEST.FONT". I even know where it switches to 80-column mode ($18CF).

But that's where it gets interesting. After it switches to 80-column mode, every stack address is the same ($0889) except the final QUIT command ($4059). So all of those MLI calls are coming from inside the interpreter, presumably some MLI wrapper function.

Also, that READ_BLOCK (command $80) is very suspicious. That's a raw sector read (really two sectors, since ProDOS does everything by blocks). So, it

1. issues a READ_BLOCK MLI call, then
2. opens/reads/closes a file, then
3. quits at $4059

Also, there is a file named "QUIT" on this disk which loads at $4000. If that isn't all connected, it would be a coincidence of magnificent proportions.



Chapter 4
In Which Up Is Down, Black Is White, And Failure Is Success

I have another crazy idea: upgrade my logger to modify the MLI command on the fly. That is, if the command on entry is $80 (READ_BLOCK), change it to something else, like $CC (CLOSE). Why CLOSE? Because it will always fail. No, really. READ_BLOCK takes 3 parameters; CLOSE takes 1. The first thing the MLI does is check whether the parameter count is correct for the given command; if not, it exits immediately with return code $02.

(And remember, I want the READ_BLOCK command to fail. On the original disk, it fails because of the intentionally bad sector on track $22. Failure is success.)

```
]PR#7
]CALL -151

; get the top stack address and put it
; in $0116/$0117
0100-   BA          TSX
0101-   BD 02 01    LDA   $0102,X
0104-   8D 17 01    STA   $0117
0107-   BD 01 01    LDA   $0101,X
010A-   8D 16 01    STA   $0116

; increment the address we got from
; the stack (does not touch the stack
; itself, just our local copy)
010D-   EE 16 01    INC   $0116
0110-   D0 03       BNE   $0115
0112-   EE 17 01    INC   $0117

; now this instruction will load the
; MLI command
0115-   AD FF FF    LDA   $FFFF
```

```
; is it READ_BLOCK?
0118-   C9 80        CMP   #$80

; no, branch without changing anything
011A-   D0 11        BNE   $012D

; yes, change the MLI command to CLOSE
; by modifying the STA instruction at
; $012A
011C-   AD 16 01     LDA   $0116
011F-   8D 2B 01     STA   $012B
0122-   AD 17 01     LDA   $0117
0125-   8D 2C 01     STA   $012C
0128-   A9 CC        LDA   #$CC
012A-   8D FF FF     STA   $FFFF

; call the real MLI
012D-   4C 4B BF     JMP   $BF4B

*PREFIX /RNR
*BLOAD RNR.SYSTEM,A$2000,TSYS

; set the trap
*BF00:4C 00 01

; execute the program
*2000G
...works, and it is glorious...
```

Now I have a way to get a copy to load. But how should I make this change permanent? I could install this MLI trap on boot and literally route every MLI command through it, changing any READ_BLOCK commands on the fly. But I'm not 100% sure that there aren't other legitimate uses of READ_BLOCK buried deep within the program. I'm also not 100% sure that the trap won't get overwritten if the stack gets too big, and I'm not sure where else I could put it in memory that wouldn't ever be used during the entire lifetime of the program. It all just feels inelegant. It's not a solution; it's a prototype.

A real solution would be to find the interpreted code that calls that MLI function wrapper (near $889) with the READ_BLOCK command and checks for its expected failure. Then I can either change the if-then logic, or NOP it out somehow, or possibly just change that one READ_BLOCK command to CLOSE. Like my prototype, but without affecting every single MLI call and without taking up any extra memory.

OK, that's what I want to do. Find the interpreted code that ends up issuing the READ_BLOCK MLI call, and change the MLI command from $80 to $CC. A one byte patch. Let's go.

Chapter 5
One Byte To Rule Them All, And In The Darkness Patch Them

```
]PR#7
]CALL -151

; get the top stack address and put it
; in $0122/$0123
0100-   BA          TSX
0101-   BD 02 01    LDA   $0102,X
0104-   8D 23 01    STA   $0123
0107-   BD 01 01    LDA   $0101,X
010A-   8D 22 01    STA   $0122

; increment the address I just copied
; from the stack
010D-   EE 22 01    INC   $0122
0110-   D0 03       BNE   $0115
0112-   EE 23 01    INC   $0123

; print the address of the current
; opcode, which is stored in ($18)
0115-   A5 19       LDA   $19
0117-   20 DA FD    JSR   $FDDA
011A-   A5 18       LDA   $18
011C-   20 DA FD    JSR   $FDDA
011F-   EA          NOP
0120-   EA          NOP

; check the MLI command
0121-   AD FF FF    LDA   $FFFF

; is it READ_BLOCK?
0124-   C9 80       CMP   #$80

; no, branch
0126-   D0 03       BNE   $012B

; yes, break
0128-   4C 59 FF    JMP   $FF59
```

```
; continue to real MLI (everything
; other than READ_BLOCK command)
012B-   4C 4B BF    JMP   $BF4B

*PREFIX /RNR
*BLOAD RNR.SYSTEM,A$2000,TSYS

; set the trap
*BF00:4C 00 01

; and go
*2000G

[output before 80-column mode omitted]
1968
411F
411F
411F
411F
411F
5191
<beep>
```

The interpreter is at $5191 when it issues the raw block read command. That may not be the actual MLI command ID. In fact, it's almost certainly not, since you need to pass a bunch of other parameters with a READ_COMMAND MLI call (unit number, data address, and block number). But it's probably nearby.

```
*5180.519F

5180- 0D 97 0E 09 1D D1 0D BB
5188- 08 80 00 FD 1C D1 0D EC
5190- 1C 1F 0C 88 47 45 54 5F
5198- D5 63 51 F2 0D BB 08 30
```

Bingo. I see an $80 byte at $5189. I bet that's the MLI command in the interpreted code.

Turning to my trusty Disk Fixer sector editor, I look for the surrounding opcodes.

```
[Disk Fixer]
  --> "F"ind
    --> "H"ex
      --> "08 80 00 FD 1C D1 0D EC"
```

It finds one match on T1E,S05.

T1E,S05,$28 change "80" to "CC"

```
]PR#6
```
...works, and it is glorious...

The other three sides are COPYA-able, which suggests that there is no further copy protection. Just to be safe, I searched each disk for that sequence of interpreted opcodes. There were no matches anywhere else.

Quod erat liberandum.

A 4am crack No. 300

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