HackFest 2013

Introduction, take two.

After returning home from Kansas Fest 2013, I thought I would quickly finish up this project and send it off for inclusion in the KFEST page. I became the victim of the "best laid plans" plague. Work got into a major crunch time, and just in time for Thanksgiving I got laid off as part of the 34,000 person work force reduction. The day before Thanksgiving, the last actual work day, on my way home I was rear ended and the car I was driving was destroyed, sending me to the hospital with a concussion. In the course of all this, I lost the document below, and the disk with my code on it.

I'm still unemployed, but have now, in April, found the document and disk, so have started on it again. I have been teaching myself python, and in the process have relearned hacking.

Introduction, take one.

My original thought was to determine why an Apple //c could boot a Prodos disk from the external drive, but not a DOS 3.3 disk. This was inspired by my //c , from eBay, could not write to the internal drive, and as I worked on some other programs, I would have to move my disk from the internal drive to the external one. This led me to start boot tracing the //c ROM, to refresh my memory about the process, and Apple II I/O. This is where I discovered problem one: my memory is not as good as it was 30 years ago.

So I started to build this “tool”, an interactive Applesoft program, to drive the tracing, and to document my discoveries and thoughts. This is where I hit problem two: I have been a professional programmer for 41 years. I didn’t discover that this was a problem until two months after KFest was over. I had forgotten how to hack! Hacking does not involve paying attention to minute details as one goes along. It involves charging forward, and when something doesn’t work, attend to it until it does, and then back into the fray. It also involves doing things that are convenient, and later designing improvements.

Boot tracing: the tool

The high level goal here is to trace a disk boot, and document the steps involved, and create a file with the process and results documented. The resulting program can then be pared down to just the critical steps that are the setup for the next phase.

The bulk of the tool can be divided into a few functions that need to be provided:

- Relocate Basic’s load point
- Install “&” hook and routines
- Move blocks of memory

- Disassemble and comment code
- Modify code
- Execute code, and return control to the calling program
- Save output in a file.

It all looked so simple. As often happens, real life is messy.

Step 1: Relocate Applesoft's load point

My first thought here was to change the Applesoft load point as the first step of the program. This is easier to say than to do. The versions I found in several resources involve the program copying a relatively small assembler program into memory and running it. This program relocates the basic program to the new location in memory, first by moving it past where it will go, initializing the target locations, and then moving it there. The versions I found also trimmed off the re-locator. This works well with a load and go program, but an interactive program with a run-review-modify-run cycle is not such a good match.

A more useful approach was the method of poking the location into the pointer bytes, poking $00 into the first byte of the program space, and then loading the program. This can be done from an exec-file, so that the user (me) just has to do this once.

Since I am working on DOS 3.3 based disks, I need to move the program out of the boot processes' way. Also, boot0 on ROM is located at $Cs00, where s is the slot number. On my //c (version code byte FF), that is C600-C73F. The boot process uses memory pages $03, $08, $1B thru $3F, and relocates DOS to $9D thru $BF.

The logical hole for my tool to go in starts in page $40, so the ROM would be located at $46 and $47, Applesoft load at $4800, and the "&" routines between $40 and $45. This turns into:

```
POKE 104,72
POKE 72*256,0
LOAD BOOT-TRACE or RUN BOOT-TRACE
```

Not too bad. To be a little more obvious, the locations could be expressed as digit (0-F as 0-15) * 16 + digit, $48 becoming 4*16+8 rather than 72.

For use later is the knowledge that the contents of $68 – 2 gives the boot0 page location, and ($68)-3 the page to load the "&" routine to, all when ($68) > 8. (In April, it occurred to me to try hacking a program to do this, and I created BT0, to do this all. If the pokes and PRINT D$;"LOAD or RUN xxx" is all in one basic statement, it all works.)

Step 2: Start the Program

The memory page the program starts at is our reference point. Our program needs to start by loading its tools, the code for the "&" utilities etc. If the program is to be restarted in the boot trace process, we need a flag to show whether initialization has occurred. I have chosen to use the first byte of the ampersand routine location as that flag: 0 = load, <>1 = already there. In this case I would add to the exec file, before the LOAD line:

POKE 69*256,0

And in the program code to initialize variables, and load and hook the "&" routine:

```
1 P=PEEK (104):A=P-3:L=P-2:IF P=8 THEN A=3: L=46 :REM A=AMPERSAND ADDR,L=BOOTCODE ADDR
2 D$=CHR$(4):D=4
3 IF PEEK (A*256) = 0 THEN PRINT D$;"BLOAD A1,A";A*256:POKE1013,76:POKE 1014,0:POKE 1015,A
4 CO=A*256+1:LI=A*256+10
5 A1 = 3*16+12:A2=A1+2:A4=A1+6
```

If the routine is already loaded, we skip loading it, and setting the hook. The first part of the Ampersand routine is code cribbed from the monitor ROM. The list routine has been in the same location in all versions of 8 bit Apple II, so I copied the driver from the ROM so I could alter the number of lines of disassembly listed, $14 (20) is hardcoded into the ROM. Working with this, I discovered that if X is loaded with '0', the disassembly continues from after the last address disassembled. Any other value and it copies two byte zero page location A1 to location PC and starts disassembling from there. This led to realization that I could use the X register value to select which routine to run. Zero and one drive the disassemble, two and above go to later routines. CO is the address of the X register value, and LI is the address of the lines to disassemble value.

As I hacked away, I started by loading the ampersand routine with peeks and pokes. As I added more code, this was getting out of hand, so I switched to an assembler routine. I will provide a commented listing at the end. (And in April, I moved the load of the &-routine into BT0.)

Step 3: Moving blocks of code

The monitor move command takes the form:

destination < begin . end M

In the ROM one can find that the zero page labels reformat that to: A4 < A1.A2 M. I used 3 as the command value to select the move routine. The corresponding Ampersand routine is a call to MOVE in ROM, and a return. To move the boot0 ROM to our target location, we just poke the destination address into A4L,A4H, and the boot ROM start locations into A1L/A1H and A2L/A2H.

25 POKE A4,0:POKE A4+1,L:POKE A1,0:POKE A1+1,12*16+6:POKEA2,3*16+15:POKE A2+1,12*16+7
26 POKE CO,3:&

Boot code moved. (Again, in April I moved this to BT0 also. Please see the commented listing below.)

Step 4: Disassemble and comment code

This is the core of the program. To start, set the command byte to '01', set A1L,A1H to the code start address, and set LI to the number of lines to list, and then issue the & call.

51 POKE A1,0:POKE A1+1,L:POKE CO,1:POKELI,20:&

This lists the first 20 assembler instructions, leaving the print cursor at the end of the last line. To add a comment to the last line, add a :print' your comment '; after the semicolon. To add a comment between lines, print an end-of-line followed by a print of your comment, again ending with a semicolon.

To continue disassembly from where you left off, change the command byte to "00" and your lines if different from 20, and &.

50 PRINT '* START OF BOOT0';
51 POKE A1,0:POKE A1+1,L:POKE CO,1:POKELI,20:&:PRINT' END PART 1';
52 POKE CO,0:PRINT:PRINT'* PART 2';:&:PRINT

Run this and you discover the need to pause between screens. Add a line like GET A$ at the end of each screen full.

Occasionally one or more data bytes are in-line with code, and they can confuse part of the listing. To solve this, I have added another "&" routine, command = "2", to print one byte and adjust the listing counter. I have seen boot routines that insert bytes to do just this, to try to stop boot tracing.

Those are the basic tools of exploration. To be useful, we also need the tools to control the booting from the disk.

Step 5: Modify Code

This is the fun part, where we play the hidden game. There are four important instructions that are generally useful:

```
4C B3 FE   JMP $FEB3  Reconnect basic
4C EA 03   JMP $03EA  Reconnect DOS
AD E8 C0   LDA $C0E8  Turn off disk motor
EA         NOP
```

The first returns control to interactive basic. The second reconnects DOS's hooks, though if it is needed, you may need to reboot and reload your program. The third protects your disk drive, and potentially your sanity. I have added the turn off disk and reconnect basic to the end of the "&" routine at A*256+xx. In our copied BOOT0, at the end of its execution is a JMP $0801. I replace that on the first pass with a jump to code that turns off the motor and returns to basic, line 42 in BT0 below. Once at the basic prompt you have the choice of entering RUN to restart your program from the start, or GO TO line number. The first resets all your variables, the second retains them. This can be helpful in automating the early part of your boot tracing later on in your first efforts, and in boot tracing other disks.

Generally, you modify code to do what you want, and to retain control of the process. You can do this by poking code into RAM, or by using the monitor (CALL -151) to enter the bytes in hex. See the examples below. One of the things one has to do is modify branch conditions.

Step 6: Save output in a file

This is where my first efforts at hacking this failed. I spent a great deal of time crafting code to allow the user to select the slot and drive, catalog the disk, etc. This worked fine, but only correctly saved the disassemble output when a sector break occurred when a comment was being printed by basic. The sector read/write routine in DOS stepped on zero page or stack data used by the monitor.

I have tentatively shelved this step to some later time.

Step 7: Execute code, and return control to the calling program

Now, with the code corrected for its new location, and a trap to catch it in place, the next step is to execute the code. This can be done from the monitor with, in my case 4701G, or from basic. I have added command 4 to do this.

Step 8: interactive use

I used this basic set of tools to determine the patches needed for my Apple //c ROM 255. It would have gone faster with a machine that used the standard disk 2 card, as its ROM is relocatable, so no address fix-ups would be required.

The procedure is to load the &-routine, copy and relocate Boot-0, modify it to return to basic, relocate Basic's load point, and load the boot trace program. BT0 does this. Then run the modified boot-0.

Once back in basic, start adding disassembly instructions, and figuring out what boot-1 does. Do note that on 16-sector disks, byte 0 at 800 indicates the number of sectors that are read by boot-0. Your first disassembly should be 801 thru whatever. Some copy protected disks do their protect code starting in this sector.

Enjoy.

Code: BT0 - Boot Trace Start

This is the prep program that loads the &-routine, copies the boot-0 code, loads BT1 then runs the modified boot-0.

```
 1 REM ********************
 2 REM * BOOT TRACE 0
 3 REM ********************
 4 REM * BY FORREST LOWE
 5 REM ********************
 6 REM * 1. SET UP & ROUTINE
 7 REM * 2. LOAD & ROUTINE
 8 REM * 3. USE SAME TO MOVE ROM CODE
 9 REM * 4. ADJUST ROM CODE FOR NEW LOCATION
10 REM * 5. MODIFY BASIC START
11 REM * 6. LOAD BT1
12 REM ********************
15 REM * LOAD &ROUTINE AT $4500
20 D$=CHR$(4):A=4*16+5:PRINT D$;"BLOAD A1,A"A*256
21 POKE 1013,76:POKE 1014,0:POKE 1015,A
22 A1 = 3*16+12: A2 = A1 + 2: A4 = A1 + 6:REM MON ROM LOCS
23 CO = A*256+1:REM COMMAND BYTE ADDRESS
25 REM MOVE BOOT0 TO RAM
30 POKE A4,0: POKE A4+1,4*16+6
31 POKE A1,0: POKE A1+1,12*16+6
32 POKE A2,3*16+15: POKE A2+1,12*16+7
33 POKE CO,3:& :REM USE & ROUTINE TO DO MOVE
40 REM NOW ADJUST ABSOLUTE ADDRESSES
41 POKE 17980,71: POKE 17997,70: POKE 18173,70
43 REM LAST JUMP TO STOP MOTOR AND START BASIC
44 POKE 18169,104: POKE 18170,69
50 REM ADJUST BASIC PGM AND LOAD BOOT TRACE PROGRAM
51 POKE 104,4*16+8: POKE 256*(4*16+8),0:PRINT D$;"LOAD BT1"
52 POKE CO,5:&:REM RUN BOOT0 AND START BT1
```

Code: BT1 - Interactive Boot Trace

```
 1 REM ********************
 2 REM * BOOT TRACE 1
 3 REM ********************
 4 REM * BY FORREST LOWE
 5 REM ********************
 6 REM * INTERACTIVE BOOT TRACE
```

```
 7 REM ********************
10 P=PEEK(104):A=P-3:L=P-2:IFP=8THENPRINT"PLEASE RUN BT0 FIRST":STOP
11 POKE 1013,76:POKE 1014,0:POKE1015,A:REM REHOOK &
12 CO=A*256+1:LI=CO+9
20 REM YOUR CODE HERE
50 STOP
```

Code: A1 The ampersand routine

```
Disassembly Driver from ROM, with mods.
4500-  A2 00     LDX  #$00
4502-  E0 02     CPX  #$02
4504-  10 17     BPL  $451D
4506-  20 75 FE  JSR  $FE75 CO 0 or 1
4509-  A9 16     LDA  #$16
450B-  48        PHA
450C-  20 D0 F8  JSR  $F8D0
450F-  20 53 F9  JSR  $F953
4512-  85 3A     STA  $3A
4514-  84 3B     STY  $3B
4516-  68        PLA
4517-  38        SEC
4518-  E9 01     SBC  #$01
451A-  D0 EF     BNE  $450B
451C-  60        RTS
Hex print 1 byte as assembly
451D-  E0 03     CPX  #$03
451F-  10 2D     BPL  $454E
4521-  A5 3A     LDA  $3A   CO 2
4523-  85 3C     STA  $3C
4525-  A5 3B     LDA  $3B
4527-  85 3D     STA  $3D
4529-  20 92 FD  JSR  $FD92
452C-  A9 A0     LDA  #$A0
452E-  20 ED FD  JSR  $FDED
4531-  A9 A0     LDA  #$A0
4533-  20 ED FD  JSR  $FDED
4536-  A9 A0     LDA  #$A0
4538-  20 ED FD  JSR  $FDED
453B-  A0 00     LDY  #$00
453D-  B1 3C     LDA  ($3C),Y
453F-  20 DA FD  JSR  $FDDA
4542-  20 C2 FC  JSR  $FCC2
4545-  A5 3C     LDA  $3C
```

```
4547-  85 3A     STA  $3A
4549-  A5 3D     LDA  $3D
454B-  85 3B     STA  $3B
454D-  60        RTS
Move block of code
454E-  E0 04     CPX  #$04
4550-  10 06     BPL  $4558
4552-  A0 00     LDY  #$00   CO 3
4554-  20 2C FE  JSR  $FE2C
4557-  60        RTS
Jump to boot 0
4558-  E0 05     CPX  #$05
455A-  10 03     BPL  $455F
455C-  4C 01 47  JMP  $4701
Jump to boot 1
455F-  4C 01 08  JMP  $0801
4562-4567 EA EA EA EA EA EA   NOP NOP NOP etc
4568-  AD E8 C0  LDA  $C0E8 Turn off drive motor
456B-  4C B3 FE  JMP  $FEB3 Reconnect basic
```