Battery Firmware Hacking

Inside the innards of a Smart Battery

Charlie Miller
Accuvant Labs
charlie.miller.com
Twitter: 0xcharlie

July 12, 2011

Table of Contents

Introduction
Ever wonder how your laptop battery knows when to stop charging when it is plugged into the wall, but the computer is powered off? Modern computers are no longer just composed of a single processor. Computers possess many other embedded microprocessors. Researchers are only recently considering the security implications of multiple processors, multiple pieces of embedded memory, etc. This paper takes an in depth look at a common embedded controller used in Lithium Ion (Li-Ion) and Lithium Polymer batteries, in particular, this controller is used in a large number of MacBook, MacBook Pro, and MacBook Air laptop computers.

This is especially important because if an attacker can modify the operation of such an embedded controller, it may be possible to cause a safety hazard such as overheating the battery or even causing it to catch on fire. Additionally, being able to take control of such an embedded controller could provide a mechanism for attacker persistence even in the presence of a complete system reinstall. It might also provide a quick method for implanting a device by simply switching out the battery in it. It might also provide a method for an attacker to observe trusted operations like communications to and from the TPM chip.

In this paper, we demonstrate how the embedded controller works. We reverse engineered the firmware and the firmware flashing process for a particular smart battery controller. In particular, we show how to completely reprogram the smart battery by modifying the firmware on it. This is possible due in part to Apple's use of default passwords for both unsealing the battery and opening up full access mode to it. Also, we reverse engineer the checksum used by the firmware to ensure only legitimate firmwares are used, and illustrate how to disable this checksum so we can make changes. We present a simple API that can be used to read values from the smart battery as well as reprogram the firmware. Being able to control the working smart battery and smart battery host may be enough to cause safety issues, such as overcharging or fire, although we do not actually do this to the batteries in this work.

Background
Batteries today are not just chemical cells, but are part of a larger system which is used to enforce safety rules, ensure optimal charging/discharging conditions, and report accurate charge levels. This system is called the Smart Battery System (SBS), see Figure 1.

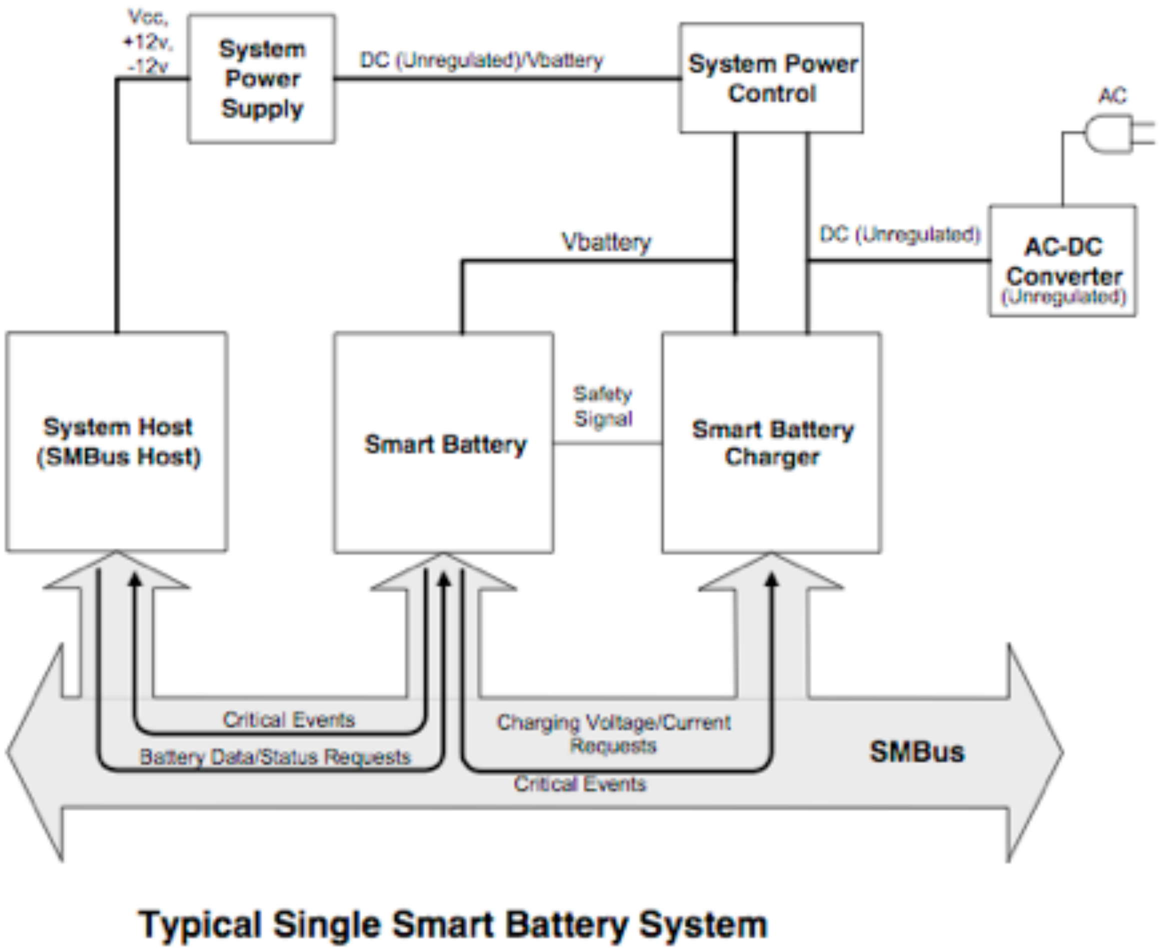


Figure 1: A Smart Battery System, from sbc100.pdf

The three components of a smart battery system can communicate via the System Management Bus (SMBus). The SMBus is a two-wire interface based on the operation of I2c. SMBus provides a control bus for system and power management related tasks. The format of the data sent over the SMBus is outlined in the Smart Battery Data Specification. These components work together to optimally charge the Smart Battery when necessary and protect the Smart Battery from receiving too much charge.

In this diagram, there are three components. The first is the system host. This is the piece of hardware being powered by the battery. In our case, it is an Apple laptop computer. It uses information obtained from the battery to determine its operation, such as when to go into sleep mode. The smart battery is a battery equipped with specialized hardware that provides present state, calculated and predicted information to its SMBus Host. The Smart Battery Charger is a a battery charger that communicates with a smart battery and alters its charging characteristics in response to the results of this communication.

Using SBS we can query these devices and set particular values. Initial experimentation shows that adjusting the safety features of the Smart Battery and even making it report some incorrect values does not affect the amount of current delivered to the Smart Battery. The amount of current delivered to the Smart Battery still slowly decreases over time until it stops. Therefore, it seems that multiple components must be manipulated in order to overcharge a battery. So, besides the Smart Battery, the Smart Battery Charger must also be manipulated, either directly over SBS or indirectly by making the Smart Battery communicate erroneous values to the Charger.

Sniffing the SMBus gives clues to what particular values are important, see the corresponding section below.

Safety is a primary design goal in the Smart Battery System specifications. In fact, the Battery Charger Specification goes as far as to say "*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. However, if an attacker can control this "primary intelligence", problems could arise.

Inside a macbook battery

The actual smart battery itself consists of multiple components as well. Disassembling the macbook battery, see Figures 2-8. Start by removing the Torx screws.

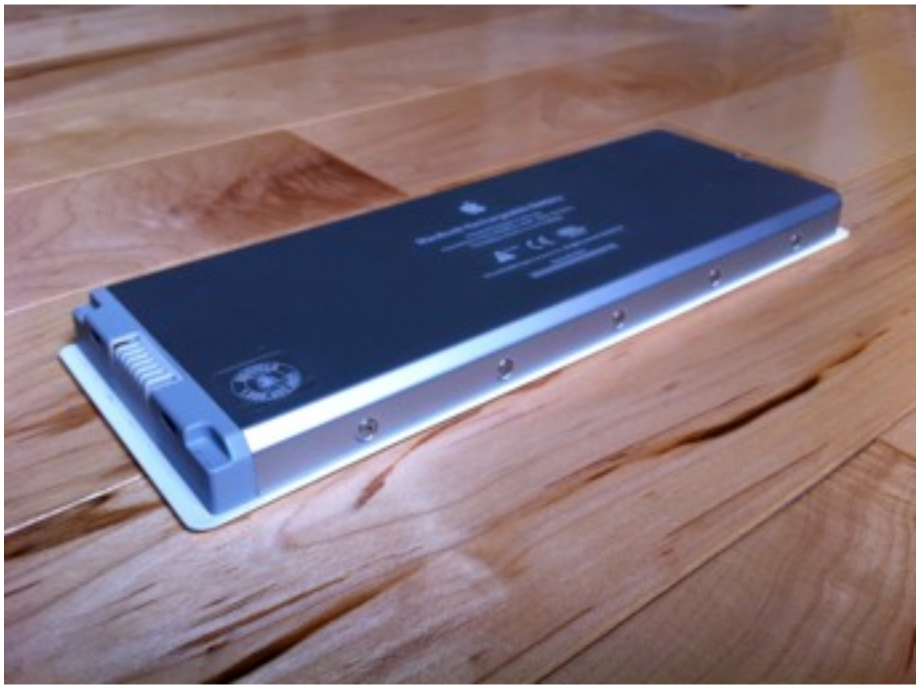


Figure 2: The Macbook battery

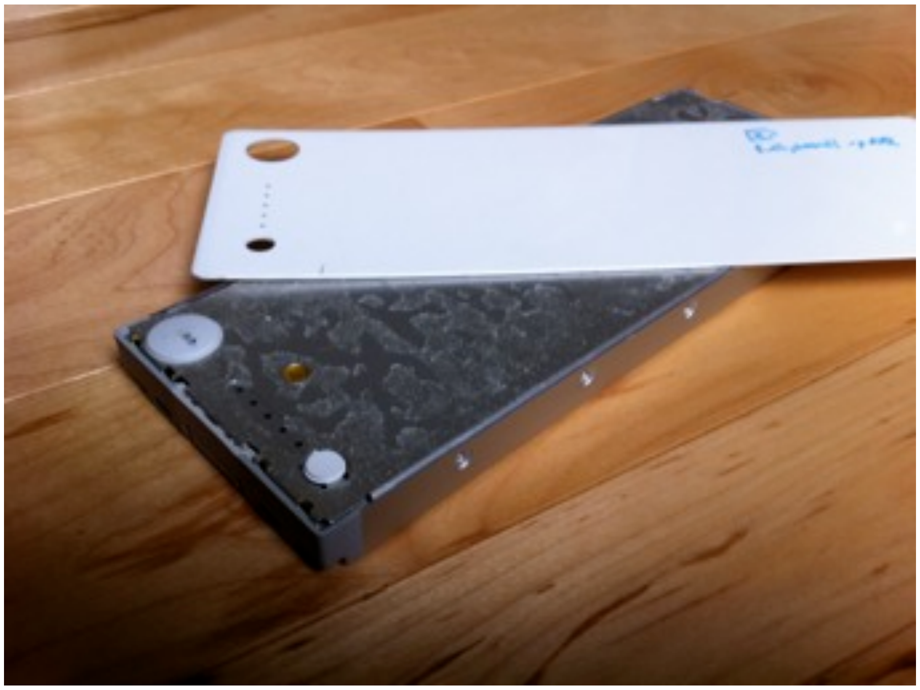


Figure 3: Rip off the cover

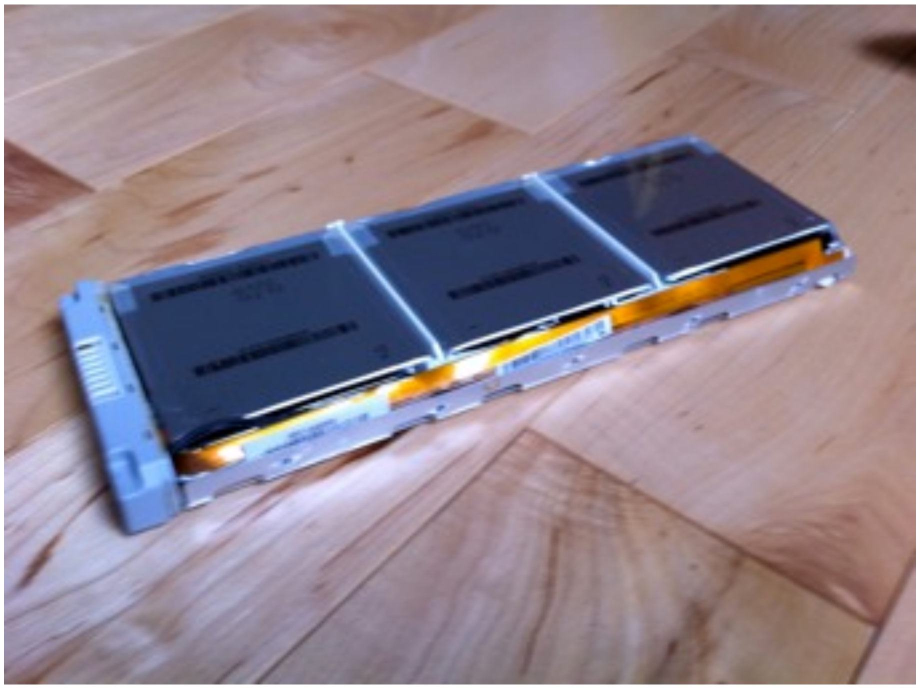


Figure 4: 6 Lithium Polymer cells inside. The electronics are on the end.

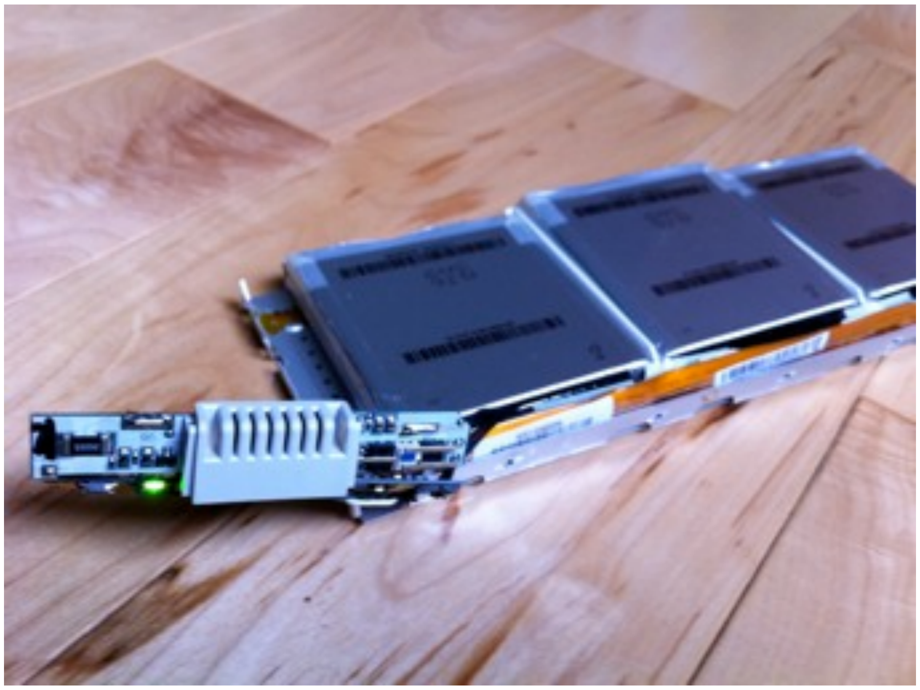


Figure 5: You can see some of the chips in this shot.

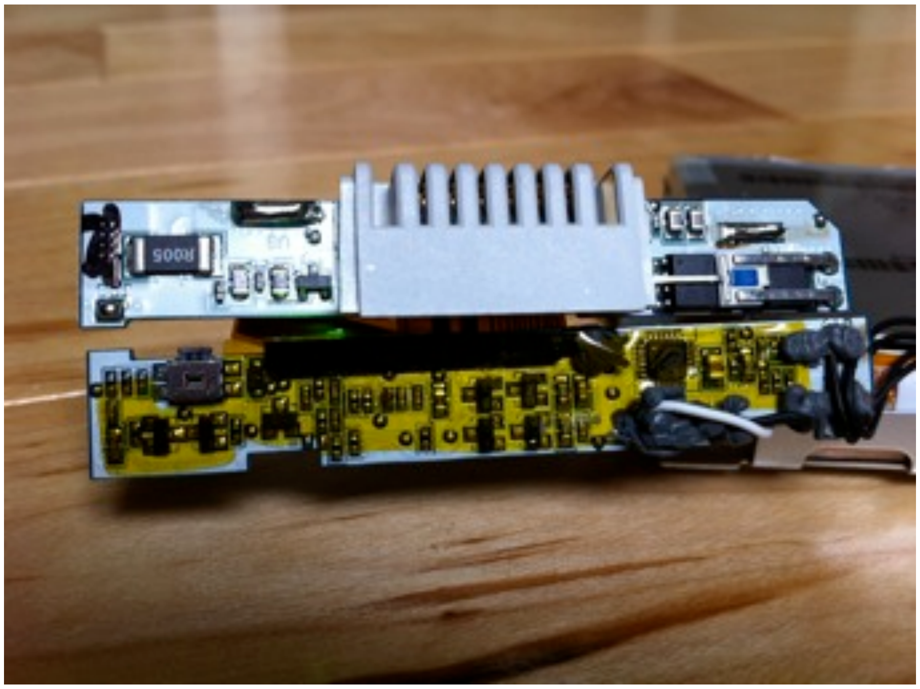


Figure 6: The bq29312 is the chip under yellow film on the right of the board

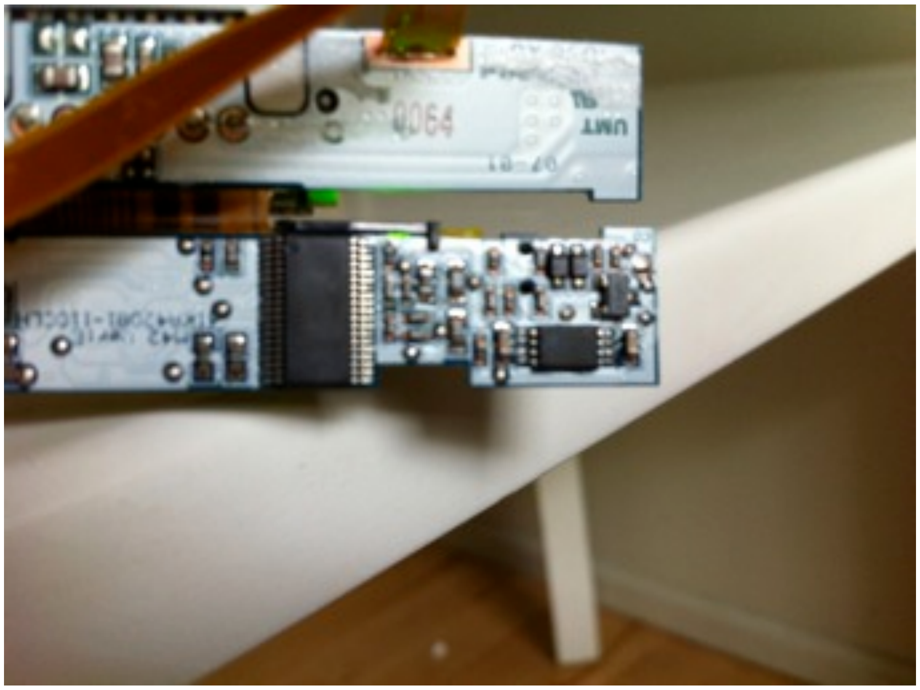


Figure 7: The larger chip is the bq20z80. The next biggest is the bq29412.

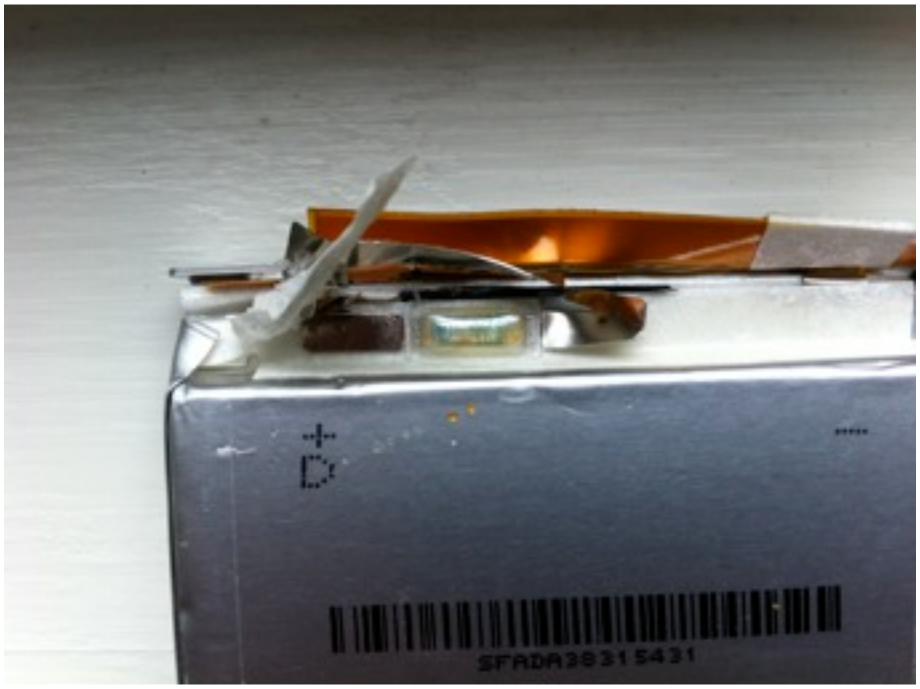
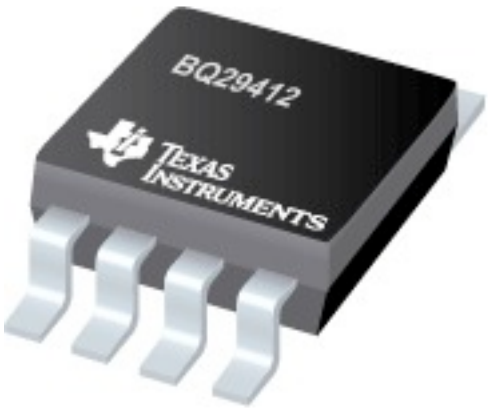


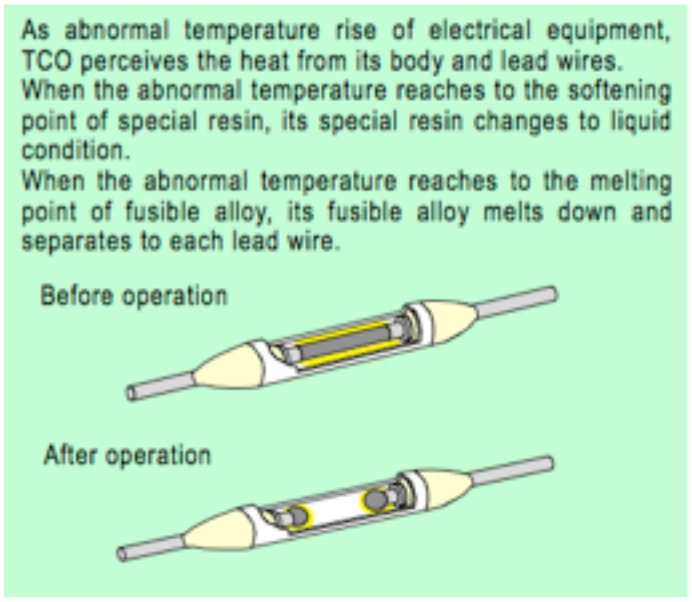
Figure 8: Here is a MU092X Thermal cutoff.

We’ll now briefly describe what these most important pieces of the smart battery do.



The chip in Figure 7 marked 29412 is a Texas Instruments bq29412 overvoltage protection IC for 2, 3, or 4-cell LiOn battery packs. The bq29412 has a fixed overvoltage protection threshold of 4.45V. If one of the cell’s voltages exceeds this threshold for more than a programmable amount of time, an external fuse is blown to prevent further charge.

The 29312 chip is a Texas Instruments bq29312 battery protection analog front end (AFE). The bq29312 provides safety protection for overcharge, overload, short-circuit, overvoltage, and undervoltage conditions in conjunction with the battery management host.



As abnormal temperature rise of electrical equipment, TCO perceives the heat from its body and lead wires.
When the abnormal temperature reaches to the softening point of special resin, its special resin changes to liquid condition.
When the abnormal temperature reaches to the melting point of fusible alloy, its fusible alloy melts down and separates to each lead wire.

Each cell has a Matsushita MU092X thermal cutoff. If the temperature rises too high, it will permanently cutoff the connection to the cell.

For Apple laptop batteries, the main component is the bq20z80 Gas Gauge (or similar model). This component interacts with the host via SBS data over SMBus. It has reprogrammable Flash Data as well as elements of the firmware itself can be changed.

The way the three chips described above interact is seen in Figure 9. Together they make up the smart battery system. Only the bq20z80 can be communicated with directly from the system host. The other chips cannot be directly accessed from the SMBus, but values set in the bq20z80 can impact their behavior.

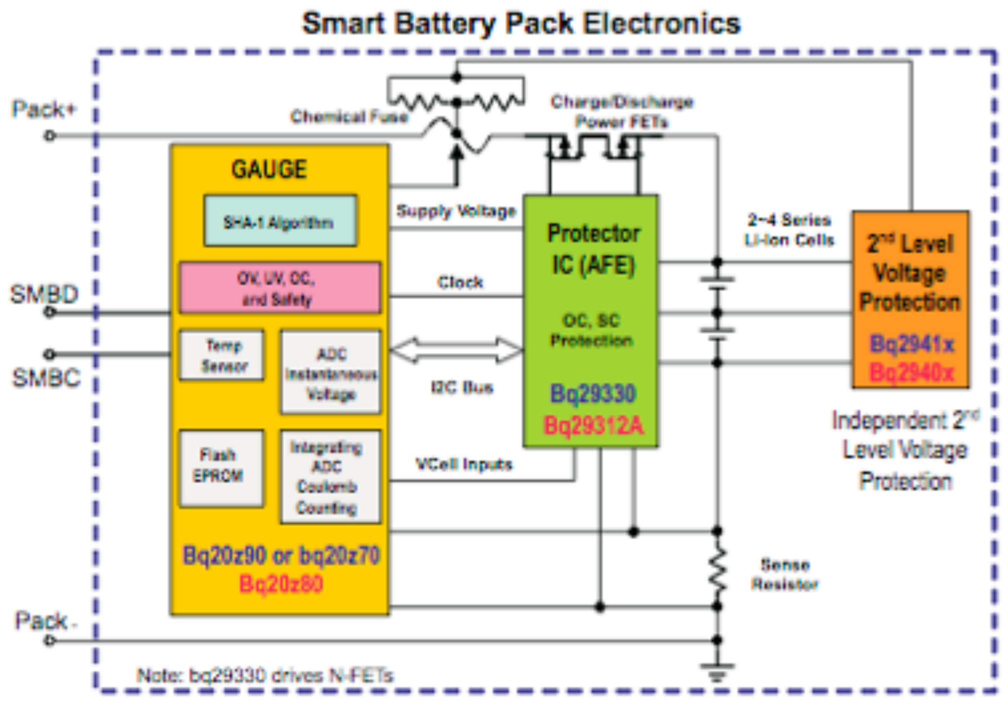


Figure 9: Smart battery electronics

For example, you can set some AFE registers via DataFlash 1st level safety, followed by a ManufacturerAccess reset(). More on this later.

SBS (Smart Battery Specification)

We mostly ignore the low level mechanisms of how the i2c/SMBus communication occurs as it is not necessary for what we are doing. On a higher level, SBS data is sent from one SMBus component to another either by broadcast or on request.

There are a variety of SBS commands which can be sent to either a Smart Battery, Smart Battery Manager, or Smart Battery Charger. Let's consider the bq20z80. There are a large variety of SBS commands which return (or set) values such as Voltage, Current, Battery Model, etc. See the specification for a complete list.

Smart Battery System Manager (SBSM)
Its not entirely clear what the Smart Battery Manager is on my computer/battery. But the generic spec indicates things it should support. It may be part of the gas gauge functionality. Regardless, the SBSM autonomously connects one or more batteries to power the system, controls the charging of multiple batteries, reports the characteristics of the battery(s) powering the system, whether AC power is present, etc. The SBSM must maintain an SMBus connection between the battery or batteries powering the system and the system host.

When the SBSM determines that the configuration has changed for any reason, the SMBus host must be notified that a change has occurred. There are two acceptable methods:

1. The SBSM masters the SMBus to the host and writes its BatterySystemState() register to the SMBus host (SMBus WriteWord protocol to the SMBus Host with the command code set to the selector's address 0x14).
2. The SBSM may notify the system independent of the SMBus by using a signal line that would cause the host to query the SBSM's BatterySystemState() and the BatterySystemStateCont

It is expected that the SBSM will operate in an entirely autonomous manner, independent of any high- level control such as that provided by an application or system BIOS. This autonomy allows the system to charge multiple batteries while the host intelligence is not operational (e.g., when the system is off or suspended). Since the SBSM operates autonomously, it is totally responsible for the battery system's safe operation and for maintaining the power integrity of the system.

In order to preserve bandwidth on the system host SMBus, the SBSM may intercept and respond directly to commands directed to either the Smart Battery or the Smart Battery Charger.

SBS commands include:

BatterySystemState 0x01	r/w
BatterySystemStateCont 0x02	r/w
BatterySystemInfo 0x04	r

Smart Battery Charger (SBC)
A SBC is a battery charger that periodically communicates with a Smart Battery and alters its charging characteristics in response to information provided by the Smart Battery. The electrical characteristics of the Smart Battery Charger feature charging characteristics that are controlled by the battery itself, in contrast to a charger with fixed charging characteristics that will work with only one cell type.

The Smart Battery Charger interprets the Smart Battery's critical warning messages, and operates as a SMBus slave device that responds to ChargingVoltage() and ChargingCurrent() messages sent to it by a Smart Battery. The Spec says that the charger is obliged to adjust its output characteristics in direct response to the ChargingVoltage() and ChargingCurrent() messages it receives from the battery, although in practice, this doesn't seem to be the case.

In Level 2 charging, the Smart Battery is completely responsible for initiating the communication and for providing the charging algorithm to the charger.

The Level 3 Smart Battery Charger may act as a slave or poll the Smart Battery to determine the charging voltage and current the battery desires, and then dynamically adjust its output to meet the battery's charging requirements.

None of the communications require the system's Host take any action to control charging.

Charging voltage and current is supplied by the battery to the charger. it will send the ChargingCurrent() and ChargingVoltage() values to the Smart Battery Charger. The Smart Battery will continue broadcasting BOTH of these values an interval of not less than 5 seconds nor greater than 1 minute in order to maintain correct charging.

The SBS commands contain things like:
• 0x11: ChargerSpecInfo
• 0x12: ChargerMode
• 0x13: ChargerStatus
• 0x14: ChargingCurrent
• 0x15: ChargingVoltage
• 0x16: AlarmWarning
• 0x21: SelectorState
• 0x22: SelectorPresets
• 0x24: SelectorInfo

The Smart Charger on my Macbook says it is a Level 2 Charger when queried and so should wait for the battery to talk to it. However, when sniffing I see something querying the battery. If it is not the charger, then what is it?

Smart Battery (SB)
A battery equipped with specialized hardware that provides present state, calculated and predicted information to its SMBus Host under software control. On my macbook, the actual embedded controller (gas gauge) is the Texas Instruments bq20z80. See Sluu276 for a full description of available SBS commands to the smart battery.

There are a large number of SBS commands that can be issued to a bq20z80. There are 35 standard SBS commands that will do things like return the voltage, current, cycle count, serial number, etc. The standard commands are ones that any device should be able to query. There are also 24 extended SBS commands. These require special privileges to interact with. In order to access these SBS commands, you must "unseal" the battery, that is, authenticate to it. There are two ways to authenticate and unseal the battery. The first is with a 4 byte unseal key that is passed using the ManufacturerAccess->Unseal Device SBS command. Another way is through the Authenticate SBS command which uses a SHA-1 based authentication transform. Extended SBS commands include things like getting safety alerts and controlling the hardware FETs. An even higher mode of privilege is possible called Full Access. A separate 4 byte password is needed to enter Full Access mode from Unsealed mode. Full access mode allows you to, for example, enter Boot Rom mode which allows low level access to the hardware.

Macbook batteries ship with a default unseal password (0x36720414). This was found by reverse engineering a Macbook battery update. On Macbook batteries, the full access mode password is also hardcoded and default (0xffffffff).

There are also a large variety of Flash Data values available. This contains things like configuration information. In order to access Flash Data, one writes the subclass ID to SBS command 0x77 (DataFlashClass) then reads or writes the value for that Flash Data using SBS command 0x78 (DataFlashClassSubClass).

In Mac OS X, there is a mechanism for (root privileged) processes to communicate with the battery using IOKit. You can open up the IOService AppleSmartBatteryManager. Then, you can read and write to this service using the IOConnectMethodStructureIsStructure0 API. I have code which can read and write arbitrary SBS commands to the battery.

Sniffing

At this point we know how to talk to the battery. However, we don't have any insight into how the system normally operates. Presumably, the smart charger communicates with the smart battery to determine how much charge to deliver to the battery. In this case, we'd like to understand this communication. In order to do this, we need to be able to sniff the SMBus.

Looking at the connection between the removable battery and the computer, there are (at least) 6 wires that could be the 2 SMBus/i2c wires we are looking for. It is a little difficult to attach probes to the battery while it is in the computer, but it can be done. If you remove the keyboard, the connection is exposed, see Figure 10.

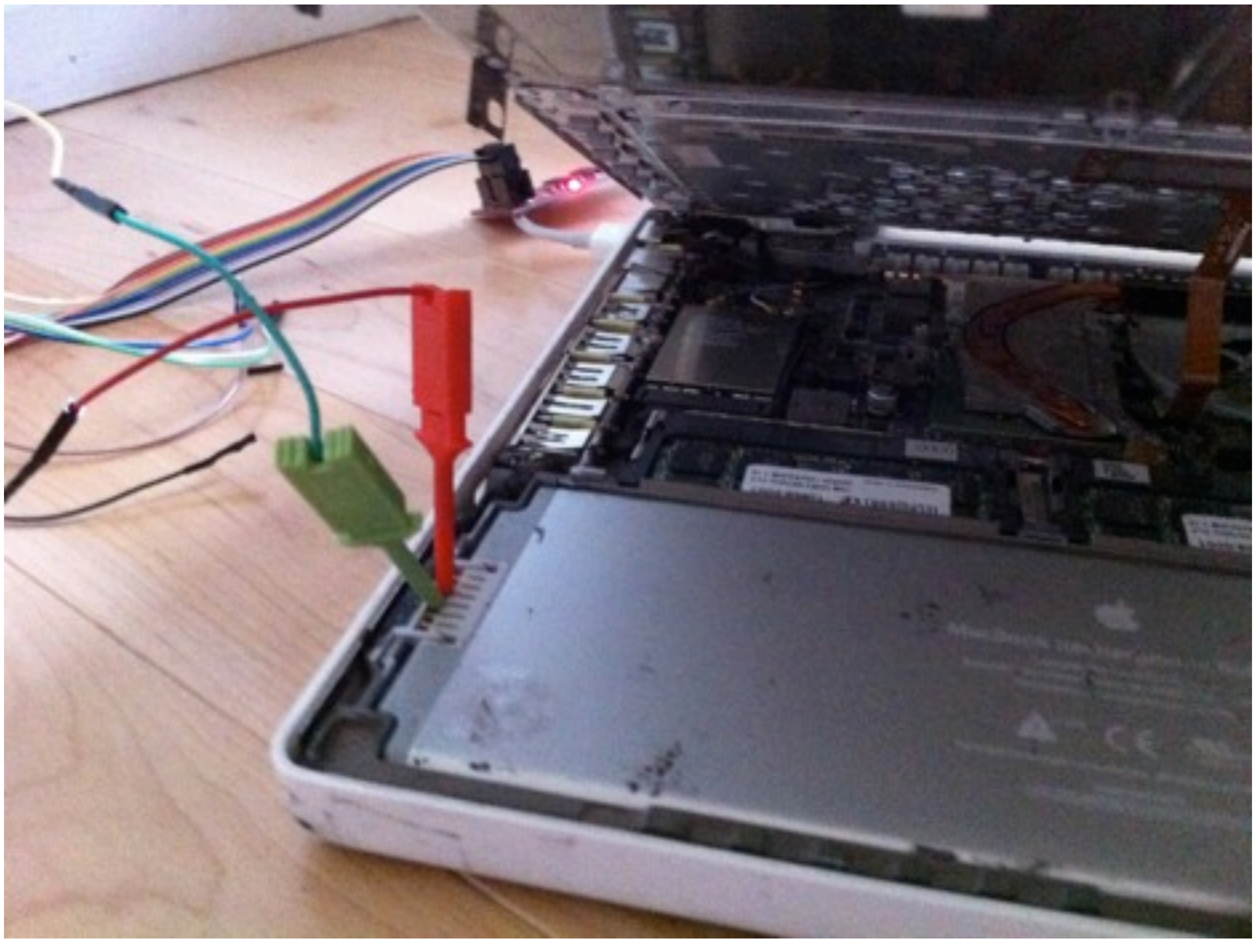


Figure 10: Sniffing the battery to host communications

I connected leads from all 6 wires to a Saleae logic analyzer. By looking at the output, it is clear which two are the SMBus wires. As the Saleae has an I2C data processor, you can actually see the data being transferred, see Figure 11.

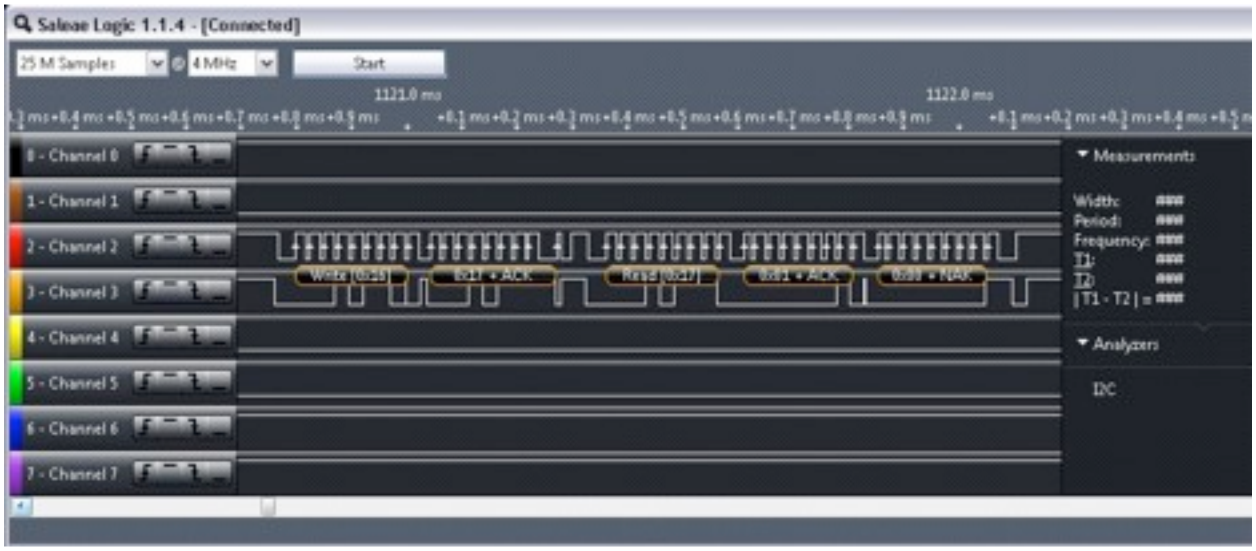


Figure 11: Raw i2c communication.

You can see the raw i2c communication. Figure 12 shows a closer look at the i2c.

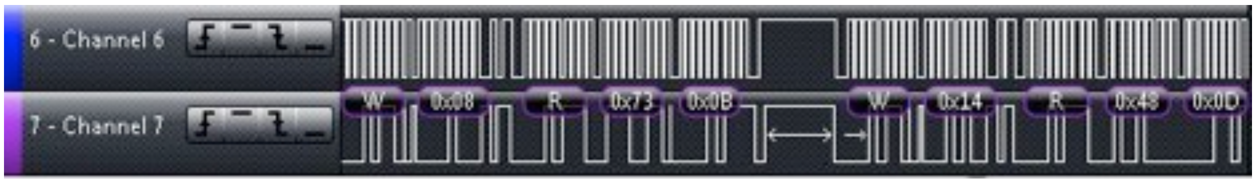


Figure 12: i2c decoded communication

Here we see a write to SBS command 0x8 (Temperature) with the response being 0xb73 which is 293.1K (or 67.9F) and a write to SBS command 0x14 (Charging Current) with the response 0xd48 which is 3400mA.

Below are some of the SBS commands that were sniffed in different situations

Charging, power on

W1616 R178000,	Battery Status: INIT
W1608 R17BF0B,	Temp: 0x0bbf
W1614 R17480D,	Charging Current: 0x0d48 = 3400mA
W160A R17FC02,	Current: 0x02fc = 764mA
W1609 R176130,	Voltage: 0x3061 = 12385mV
W1603 R170160,	Battery Mode: 0x6001: CHGM (disable broadcasts), AM (disable Alarm broadcasts)
W160D R175B00,	RSoC: 0x5b = 91% charged
W160F R17DD11,	Remaining Capacity: 0x11dd = 4573mAh
W1610 R17CC13,	Full Charge Capacity: 0x13cc = 5068mAh

Charging power off:

W1616 R178000
W1608 R17C10B
W1614 R17480D
W160A R17E902
W1609 R177730
W1603 R170160
W160D R175B00
W160F R17F611
W1610 R17CC13

Still charging power off (values that change are highlighted)

W1616 R178000
W1608 R17C30B
W1614 R17480D

W160A R17E802
W1609 R179830
W1603 R170160
W160D R175C00
W160F R172A12
W1610 R17CC13

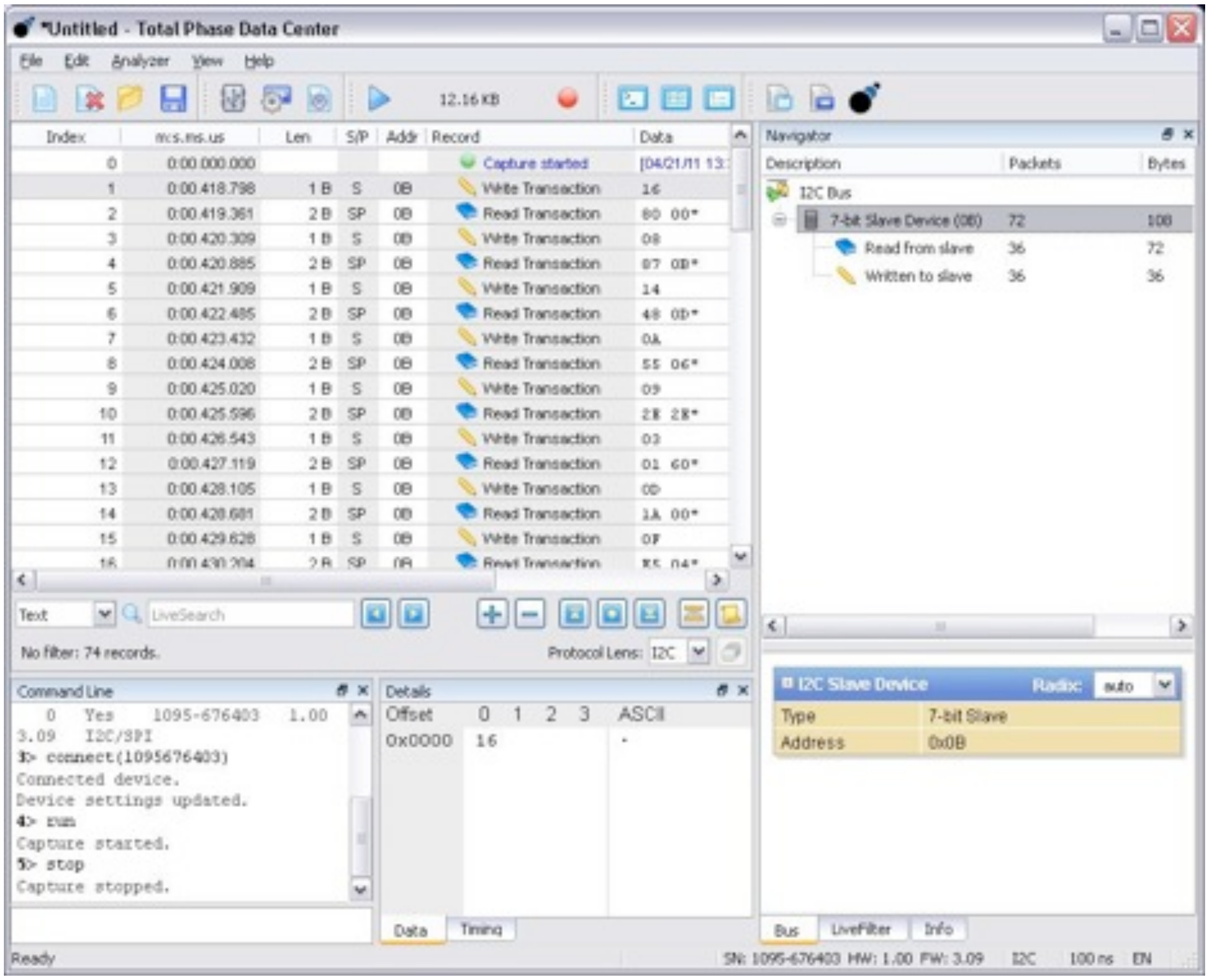
So in the case when the computer is powered off, and thus it is all smart battery hardware talking, you see reads from 03, 08, 09, 0a, 0d, 0f, 10, 14, 16. These correspond to reading the values of

- Battery Mode
- Temperature
- Voltage
- Current
- Relative State of Charge
- Remaining Capacity
- Full Charge Capacity
- Charging Current
- Battery Status

This traffic shows reads/writes to SMBus address 0x16/0x17 which are reads/writes to 0xb which is the smart battery. (0xb = 0x16>>1) When the OS power is off, there is something (presumably the charger) querying the smart battery. Again, presumably, these queries help determine how much current for it to send to the battery.

sluu276 states the following about the Current SBS command, which is one of the ones queried: “The Current function is the average of four internal current measurements over a 1-second period.”. This is what we use to monitor the amount of current being delivered to a battery.

Beagle sniff
You can only do so much with the Saleae logic analyzer. A better tool for the job is the Beagle I2C protocol analyzer. It needs to be hooked up exactly right. When looking down at the battery as it is plugged into the computer, the third wire is hooked to the Beagle SDA lead and the fourth to the SCL lead. It is also vital that both grounds are hooked to something grounded.


*Untitled - Total Phase Data Center

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	07 0D*
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	0F
16	0:00.430.204	2 B	SP	0B	Read Transaction	E5 04*

No filter: 74 records.

Protocol Lens: I2C

Description	Packets	Bytes
I2C Bus		
7-bit Slave Device (0B)	72	108
Read from slave	36	72
Written to slave	36	36

```
   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.
```

Offset	0	1	2	3	ASCII
0x0000	16				.

I2C Slave Device — Radix: auto

Type	7-bit Slave
Address	0x0B

Ready — SN: 1095-676403 HW: 1.00 FW: 3.09 I2C 100 ns EN

Figure 13: Beagle GUI recording traffic

Over a 58 minute period, I sniffed with the power off while charging and I saw only those same reads mentioned above with the Saleae. Notice that when power is on I see additional reads, namely from SBS commands:

00, 0c, 0b, 17, 12, 11, 13, 3f, 3e, 3d, 3c as well as writes to 00.

The write to 00 is a write of 0x53. Since you only see it when the OS is powered up, it is safe to say this is the operating system querying the battery. As for that write to 00, the value of 0x53 is not in the spec so I don't know what its doing there.

One interesting thing is that when I try to program the firmware of the battery, these reads and writes keep occurring, which could potentially mess up the firmware. But either by accident or design, in Boot Rom Mode this write is interpreted as a Smb_FlashWrAddr. But since Smb_FlashWrAddr is a block write and the normal write to 00 is a word write, it ignores this stuff. The write occurs every 30 seconds.

By looking at the values queried with OS power off, and assuming these are what affect the amount of current that gets sent to the battery, you can see what values the battery reports that are important. Again, these were the ones that are queried:

16: Battery Status
08: Temperature
14: Charging Current
0A: Current
09: Voltage
03: Battery Mode
0D: RSoC
0F: RC
10: FCC

It is hard to detect which of these values are really affecting how much current arrives at the device, but it probably is not the ones that describe state with bitfields or Charging Current since those don't change except when it stops charging. I don't think its the RSoC since this reports whole number percentages but you don't really see the changes of current at that rough granularity, but rather smoother (although it could be interpolating). It also doesn't seem to have much to do with Temperature, although that quantity does change a bit. That leaves **V, RC, FCC**, see Figure 14 for a graph of how these values changed over time where Series 1 is Voltage, Series 3 is RC, and Series 2 is current.

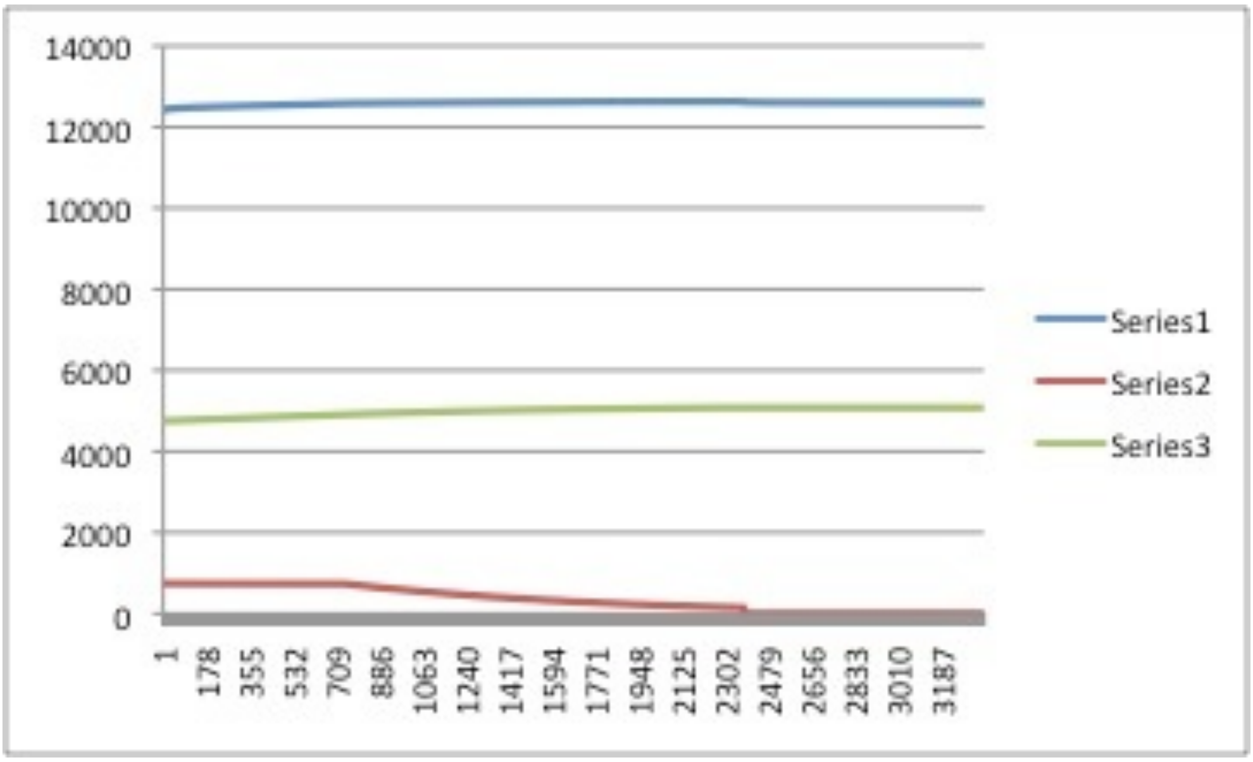


Figure 14: Sniffed V, RC, FCC as the battery recharges

Different battery modes

The gas gauge can be in different modes. Normally it runs in what is called sealed mode. Then it can go to unsealed mode, then to full access mode, then to boot rom or calibration mode. To go from sealed to unseal, write the password to SBS 0x0. To return to sealed, write 0x0020 to SBS 0x0.

To unseal
write_word(0, password_LSW)
write_word(0, password_MSW)

To seal
write_word(0, 0x0020);

Enter boot rom from full access mode
write_word(0, 0xf00);

Exit boot rom (execute flash)
send_byte(8);

Enter calibration mode from full access
write_word(0, 0x40);

Exit calibration mode
send_byte(0x73);

Calibration mode

See either slua355b.pdf or slua355a for more information on this.

I don't know much about calibration mode. Somehow, it is used to calibrate the device. Presumably, you can tell it things like the current you are delivering it and then it can compare to what it reads for calibration purposes. You might be able to use this to screw it up, i.e. convince it that it wasn't getting much current when really it was. The following commands were pulled out as examples of what you can do in calibration 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. That last thing says what to use as device, see below

//
Bit 0 Bit 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 Bit 7

Coulomb Counter Offset Board Offset ADC Offset Temperature, Internal Temperature, External 1 Temperature, External 2 Current
Voltage
Bit 8 Bit 9 Bit 10 Bit 11 Bit 12 Bit 13 Bit 14 Bit 15
Pack Gain Pack Voltage AFE Error Reserved Reserved Reserved Run ADC Task Continuously Run CC Task Continuously

read_word(0x52, y)	y = bit field about which things are calibrated. (poll with this)
send_byte(0x72)	transfer results to data flash
send_byte(0x73)	exit Calibration mode.

Boot ROM mode

While in Boot Rom mode, SBS commands do not behave according to the SBS spec. Instead they follow the ROM API, see sluu225.pdf. While in Boot ROM mode, it is possible to do low level reads and write to the data flash and the instruction (firmware) flash. While in Boot ROM mode, the battery cannot give or take charge. The battery was not designed to have Boot ROM mode entered except at the factory. It is very easy to brick the battery while in Boot ROM mode.

Texas Instruments Evaluation Module

It is possible to buy from Texas Instruments an evaluation board which contains the smart battery components as well as some software to aid in programming it. This kit is very useful for figuring out how things work without ruining your real batteries. The BQ20Z80EVM-001 is a complete evaluation system for the bq20z80/bq29312A/bq29400 Li-Ion smart battery chipset. The EVM includes one bq20z80/bq29312A and the independent 2nd level overvoltage protection bq29400, the circuit module with current sense resistor and thermistor, an EV2300 PC interface board for gas gauge interface, a PC serial cable (USB), and Windows-based PC software, see sluu205a, see Figure 15.

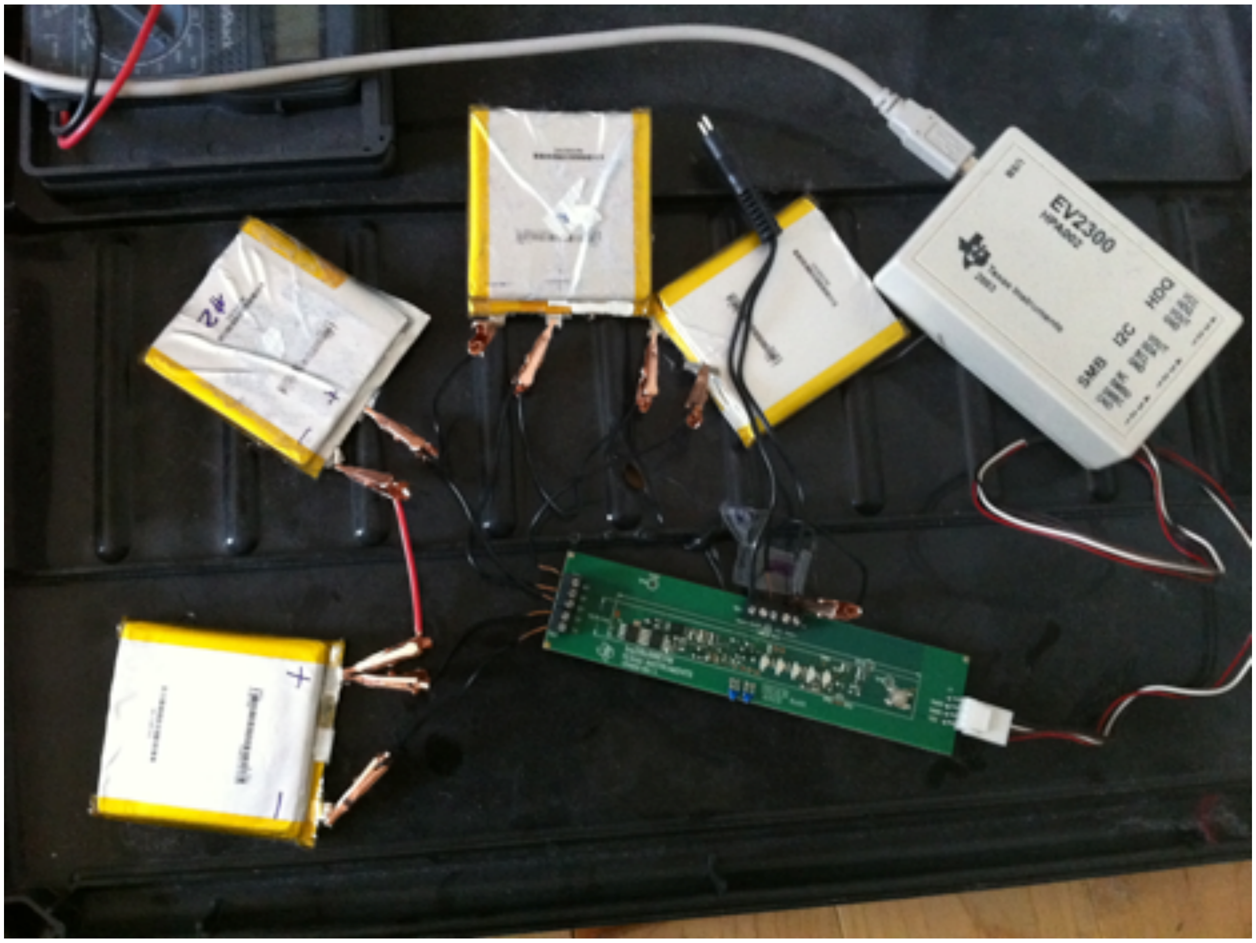


Figure 15: The bq20z80 test rig

Using the software provided, it is possible to communicate with the test gas gauge over SMBus. It is also possible to reprogram the flash with provided firmwares. If you sniff the communication during reprogramming, you can get a better idea of how to use the Boot ROM API as there is no other example of its use during normal operation, or with any patches provided by Apple.

bq Evaluation Software
Version 0.9.65
Battery Management Team,
Texas Instruments Inc.

All rights reserved. © 2006 Texas Instruments Inc.
Note: The use of this program is governed by the License Agreement.

Figure 16: The TI software to program the test gauge

Reprograming the test gauge

Using the Beagle while the TI program reprograms the battery, you can see what reprogramming does. It erases everything, reprograms the data flash and then the instruction flash. It does the instruction flash out of order, see below for details. It also reads checksums to verify success although this is presumably not strictly necessary, as it does not have anything to do with the checksum contained in the actual firmware.

```
// setup
<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>
```

It does the 00 and 01 rows last so if something goes wrong, it won't start up and crash. It is a safety mechanism, nothing else.

Device layout

By using the Boot ROM API, you can read or write various things (more later). In particular, you can dump the Data Flash and separately, the Instruction Flash. There is another part of the device where ROM code sits and cannot be accessed, even in Boot ROM mode.

According to sluu225, The bq802xx contains 6K of mask ROM code, consisting of boot-ROM code and library routines. The boot-ROM code executes at reset and detects whether the bq802xx is configured to boot into the application program in flash memory. If not, the boot ROM makes available a set of SMBus-accessible routines for flash programming and verification, and reading or writing the data memory space (including hardware registers). The ROM also contains library routines, which can be called from programs running in flash memory.

The Data Flash comes in 0x40 rows of 32 bytes each for a total of 0x800 (2048) bytes. This Data Flash resides at address 0x4000 - 0x4500 according to sluu225. We see it from 0x4000-0x4800.

The Instruction Flash comes in 0x300 rows of 32 triwords for a total of 0x12000 (73728) bytes. The instructions go from address 0 to 0x12000, although the chip considers each triword to be at a seperate address. That is, the second triword (bytes 3,4,5 from the instruction flash) are at address 0x1. Due to the harvard architecture, these addresses may not correspond directly to the data flash addresses.

sluu225 says SMB ROM functions can be found a little after 0x4000 and ROM at 0x8000.

My firmware image shows something at 0x6000 (flash data?) and functions at 0x8000.

You cannot read the actual ROM instruction code, or at least I don't know how. Doing reads with BootROM functions only seems to read the data flash. Reading with the instruction flash reading functions produces more code, but it is just repeats of the instruction flash.

The actual ROM functions found at 0x8000 are documented in the Boot ROM API document and can aid in reversing the firmware.

For example, dumping the data flash can be done quite easily in Boot ROM mode if you know the address:

```
unsigned char *read_row_data(unsigned char rownum){
    unsigned short addy = 0x4000 + (0x20 * rownum);
    write_word(kSetAddr, addy);
    int len = 0;
    return (unsigned char *) read_block(kReadRAMBlk, &len);
}

void read_flash_data(FILE *fd){
    for(int i=0; i<0x40; i++){
          unsigned char *row = read_row_data(i);
          fwrite(row, 1, 32, fd);
    }
}
```

Reprogramming the real battery gas gauge

Although it took many failed attempts, it seems that it is possible to reprogram the Instruction Flash and the Data Flash for the real battery. There are some very weird things that happen if you try to read/write a word/triword. It will sometimes work and other times read/write something random. The code that I have that finally works writes a row and then verifies the write was correct by reading that row. It loops rewriting until the write is verified. In practice this can take up to 3 rewrites.

Figure 17: Battery graveyard

One interesting thing is that erasing a row of Data Flash changes all the values to ff while erasing a row of Data Instructions changes it all to ff ff 3f, a NOP.

I have patch_* functions which do the patches without error.

The battery firmware itself

The actual firmware is machine code for a CoolRISC 816 8-bit Microprocessor. This was found by Dion Blazakis by googling some opcodes from the end of the firmware. Texas Instruments considers this information proprietary and would not reveal it.

Most registers are 8 bit (1 byte), although some are combined into larger 16-bit registers. The entire address space is 16 bits. IDA Pro cannot disassemble this machine code. I wrote a processor script for IDA Pro to disassemble this firmware called bq20z80.py.

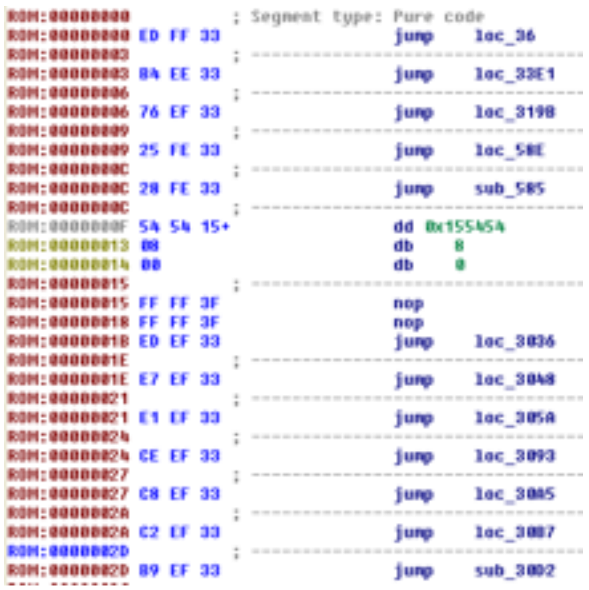
The CoolRISC is Harvard RISC-like architecture. Each instruction is 22-bits wide. When loading it into IDA Pro, there is a portion of ROM (which I don’t actually have) at 0x18000-0x18100 (actually 0x6000). Also there is some data flash accessed at

0x12000-0x12100. (actually 0x4000) This off by a factor of 3 stuff is because each instruction is 3 bytes but the processor uses addresses as if they were each one byte.

According to sluu225, the reset and interrupt vectors of the bq802xx are populated with JUMP instructions. They are defined in the assembly support file crt0.s and are arranged in flash program memory as follows:

0x0000: jump main
0x0001: jump xinHandler
0x0002: jump pinHandler
0x0003: jump cinHandler
0x0004: jump smbWaitIntr

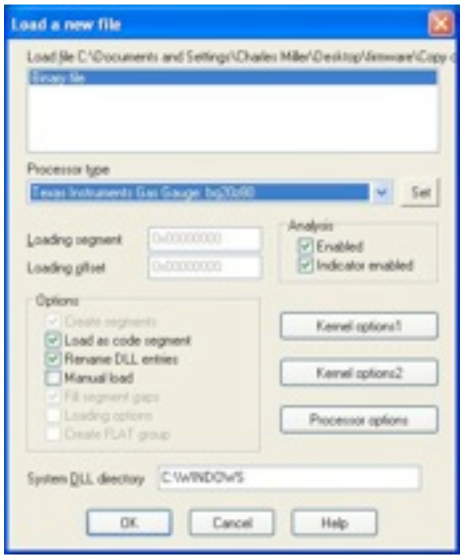
Control is transferred here by the boot ROM when it finds the flash integrity word defined as 0x155454 at address 0x0005. We see all of this in IDA Pro in Figure 18.



```
ROM:00000000           ; Segment type: Pure code
ROM:00000000 ED FF 33                      jump    loc_36
ROM:00000003           ; ---------------------------------
ROM:00000003 84 EE 33                      jump    loc_33E1
ROM:00000006           ; ---------------------------------
ROM:00000006 76 EF 33                      jump    loc_3198
ROM:00000009           ; ---------------------------------
ROM:00000009 25 FE 33                      jump    loc_58E
ROM:0000000C           ; ---------------------------------
ROM:0000000C 28 FE 33                      jump    sub_585
ROM:0000000C           ; ---------------------------------
ROM:0000000F 54 54 15+                     dd 0x155454
ROM:00000013 08                            db    8
ROM:00000014 00                            db    0
ROM:00000015           ; ---------------------------------
ROM:00000015 FF FF 3F                      nop
ROM:00000018 FF FF 3F                      nop
ROM:0000001B ED EF 33                      jump    loc_3036
ROM:0000001E           ; ---------------------------------
ROM:0000001E E7 EF 33                      jump    loc_3048
ROM:00000021           ; ---------------------------------
ROM:00000021 E1 EF 33                      jump    loc_305A
ROM:00000024           ; ---------------------------------
ROM:00000024 CE EF 33                      jump    loc_3093
ROM:00000027           ; ---------------------------------
ROM:00000027 C8 EF 33                      jump    loc_3085
ROM:0000002A           ; ---------------------------------
ROM:0000002A C2 EF 33                      jump    loc_3087
ROM:0000002D           ; ---------------------------------
ROM:0000002D B9 EF 33                      jump    sub_3002
```

Figure 18: reset and interrupt handlers for the firmware

See below on how to configure IDA to view the firmware using the processor script.


Load a new file

Load file C:\Documents and Settings\Charles Miller\Desktop\firmware\Copy

Binary file

Processor type: Texas Instruments Gas Gauge bq20z80

Loading segment: 0x00000000

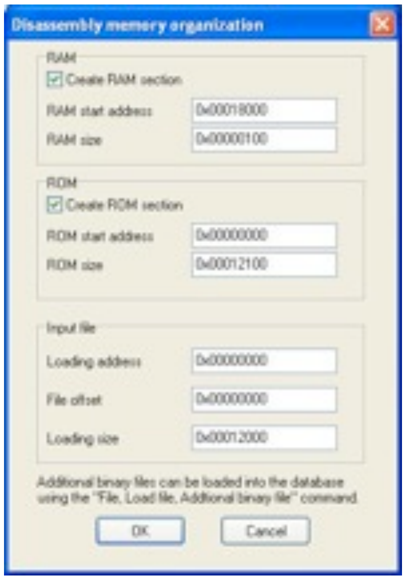
Loading offset: 0x00000000

Analysis
- [✓] Enabled
- [✓] Indicator enabled

Options
- [✓] Create segments
- [✓] Load as code segment
- [✓] Rename DLL entries
- [ ] Manual load
- [✓] Fill segment gaps
- [ ] Loading options
- [ ] Create FLAT group

System DLL directory: C:\WINDOWS

Figure 19: IDA Pro settings for newly defined processor


Disassembly memory organization

RAM
- [✓] Create RAM section

RAM start address: 0x00018000

RAM size: 0x00000100

ROM
- [✓] Create ROM section

ROM start address: 0x00000000

ROM size: 0x00012100

Input file

Loading address: 0x00000000

File offset: 0x00000000

Loading size: 0x00012000

Additional binary files can be loaded into the database using the "File, Load file, Additional binary file" command.

Figure 20: More IDA Pro settings

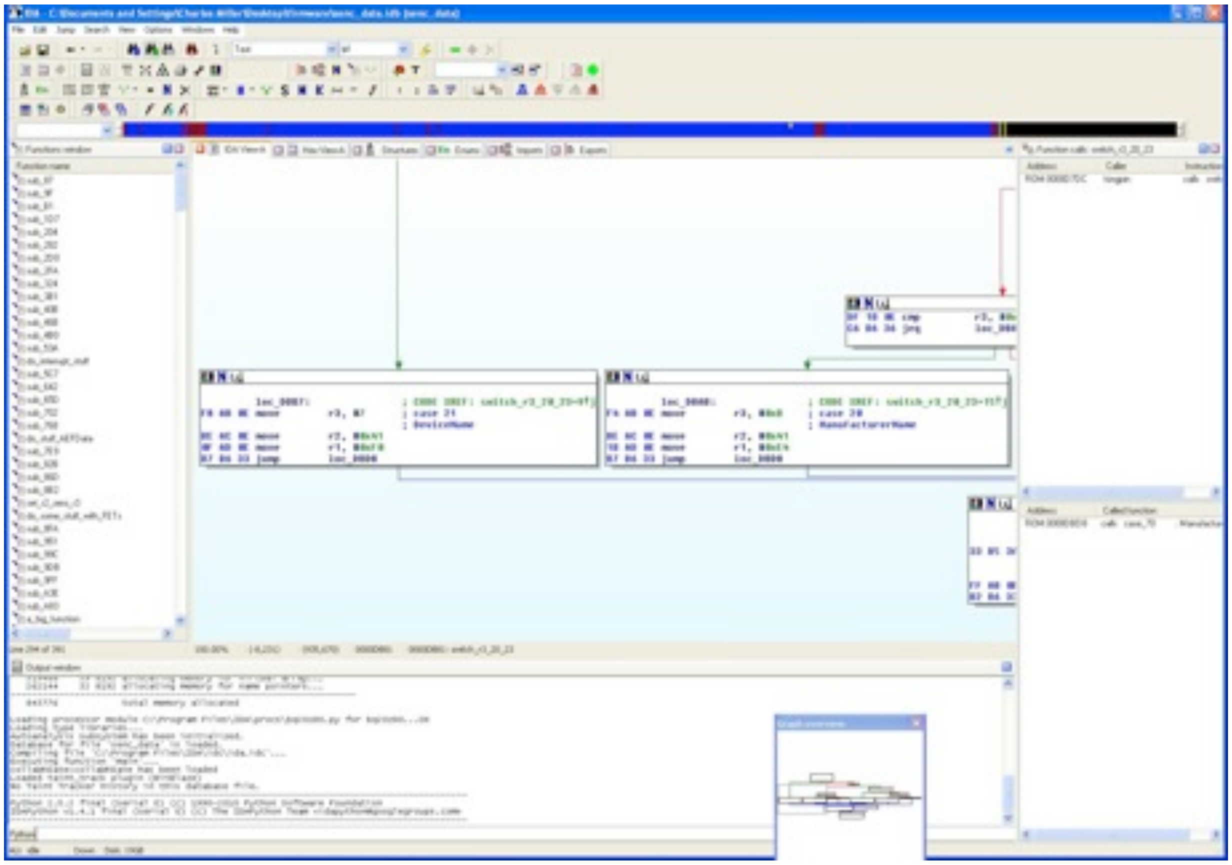


Figure 21: IDA Pro disassembly of firmware

How to switch device name for manufacturer name. Change the constants.

After you load the firmware, run the IDApython script disassemble_all.py. This tries to disassemble starting at every third byte.

Firmware checksum

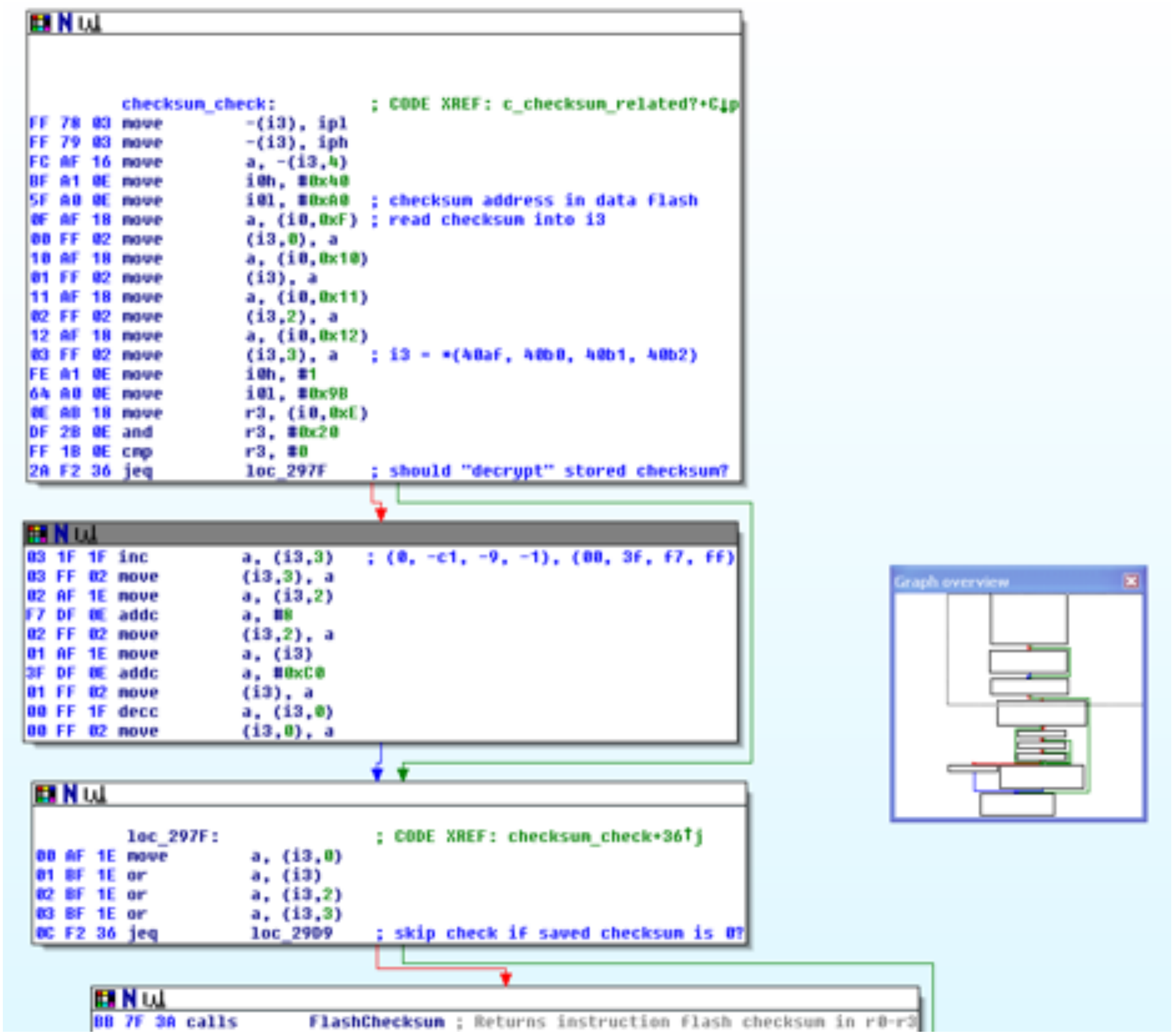
On the test gauge, doing individual firmware writes of either one triword or one row cause the DFF bit of the Permanent Failure Status register (PFStatus) to be set. The DFF bit is documented to mean:

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.

Therefore, there is some kind of checksum that is violated if you change the instruction flash. Since the instruction flash is all instructions and the only two things that change in a reprogram is the data flash and the instruction flash, it makes sense that the checksum is stored in the data flash somewhere.

If you look at the data flash, which is 0x800 bytes long, there is not an obvious checksum.

The location of the checksum can be found via static reverse engineering of the firmware image. If you look at the binaries, you can see them accessing flash as you'd expect at address 0x4xxx, for example in the case of SBS command 0x17. Well, the BootROM function FlashChecksum is at 0x8044 (0x180cc in IDA) according to the BootROM API. This function is only called twice. Once it is called in a response to 0x22 for manufacturer status, which some docs say should provide the instruction flash checksum (in practice this doesn't actually work) and the other is in some place where they compare the result to something stored from flash memory. That place in flash memory is the location of the checksum. Also, there appears to be a check that if the stored checksum is 0 it doesn't bother to check it (for some versions of firmware, the stored checksum is "decoded" before comparison and so it has to be an "encoded" 0 to not check it). So if we write a 0 to the location of the stored checksum, it has the effect of disabling the checksum and allowing arbitrary modifications. Since this is in the data flash, it can be changed by simply changing the data flash, i.e. you don't have to go into BootROM mode to do it. The location of the stored checksum will be firmware version dependent. In many versions of the firmware, the checksum shows up as the second dword for Flash data subclass 57, which happens to be one of the undocumented portions of data flash.



```
            checksum_check:             ; CODE XREF: c_checksum_related?+C↓p
FF 78 03 move           -(i3), ipl
FF 79 03 move           -(i3), iph
FC AF 16 move           a, -(i3,4)
BF A1 0E move           i0h, #0x40
5F A0 0E move           i0l, #0xA0  ; checksum address in data flash
0F AF 18 move           a, (i0,0xF) ; read checksum into i3
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 = *(40aF, 40b0, 40b1, 40b2)
FE A1 0E move           i0h, #1
64 A0 0E move           i0l, #0x9B
0E A8 18 move           r3, (i0,0xE)
DF 2B 0E and            r3, #0x20
FF 1B 0E cmp            r3, #0
2A F2 36 jeq            loc_297F    ; should "decrypt" stored checksum?
```

```
03 1F 1F inc            a, (i3,3)   ; (0, -c1, -9, -1), (00, 3F, f7, FF)
03 FF 02 move           (i3,3), a
02 AF 1E move           a, (i3,2)
F7 DF 0E addc           a, #0
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:                   ; CODE XREF: checksum_check+36↑j
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_2909    ; skip check if saved checksum is 0?
```

```
00 7F 3A calls          FlashChecksum ; Returns instruction flash checksum in r0-r3
```

Figure 22: Firmware checksum verification function for version 1.10.

The 1.12 version of the firmware lets changing it to zero disable checksums.

The older 1.10 version of the firmware has some “encoding” of the checksum, see Figure 22.

Modifying the firmware

At this point, you can change the smart battery’s firmware to do anything you want. If you want to affect the amount of current it receives, the Beagle sniffing indicates we probably want to change some combination of the Voltage (0x9), RC (0xf), and FCC (0x10). These are all set in the same function in the firmware. We’d like to change the firmware so that when you query for those values, it actually returns something else. So, In particular, we’d like to change it to something that meets the following criteria:

- Something returned from the same function

- Something that allows read-word functionality
- Something writable, so we can control it
- Something not used by anything seen in Beagle with power on or off

Example: “Hotel battery”

We start simple and demonstrate some of the API my code provides. I named my batteries after the military way of calling letters. Sadly, this particular battery was the eighth battery I played with, all previous versions having been bricked.

Dump instruction and data flash

Initial data dump

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

Then, to check that things are working, do it again.

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

This verifies that reads are consistent (which doesn’t naturally happen, your code has to check for erroneous reads, it only took me 6 batteries to figure that one out!).

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

The md5sum of the data changes because the Flash data contains, among other things, the number of times its been put into Boot ROM mode. A diff shows they only differ in a few spots.

```
$ diff golf*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
```

If you wanted to you could figure out what SBS data classes those bytes corresponded to, if they are documented.

Disable firmware checksum

Looking back at the disassembly in Figure 22, we can see that the checksum can be found at address 0x40af. By reading the data from the saved data flash file and comparing it to the return of all the subclass calls, we can determine that the address 0x40af corresponds to the second dword of subclass 57. (Alternatively, you can just directly modify the Data flash in Boot ROM mode). To see this the first 10 bytes read from subclass 57 are

01 13 0 4e **00 3f f7 ff** 1 f4

while locations 0xaf-0xb2 from data file (corresponding to address 0x40af-0x40b2) are also 00 3f f7 ff.

The following code will read subclass 57, modify the checksum, and then write the new values into subclass 57, all in unsealed mode.

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

    rb[4] = 0x00;  //00 3f f7 ff
    rb[5] = 0x3f;
    rb[6] = 0xf7;
    rb[7] = 0xff;

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

The checksum should now be disabled.

Make a simple firmware change

The final bytes of the instruction flash are simply unused NOPs. We can patch some of them without affecting the firmware to verify the ability to make changes and have the battery still function.

```
    int worked = patch_firmware(0x11f94, (unsigned char *)
"\x01\x02\x03", 3, 1);
```

Dump firmware again and diff.

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
```

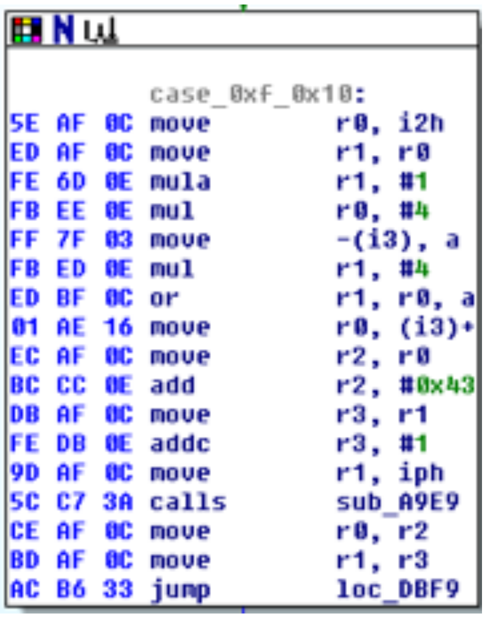
The battery still functions normally after this change since we disabled the checksum.

Change Remaining and Full Charge Capacity readings

These are SBS 0f, 10. We try to assign them to ManufacturerDate and SerialNumber (SBS commands 1b and 1c, respectively). First we change these values to ones we want:

```
    write_word(kManufactureDate, 0x122a);
    write_word(kSerialNumber, 0x13cc);
```

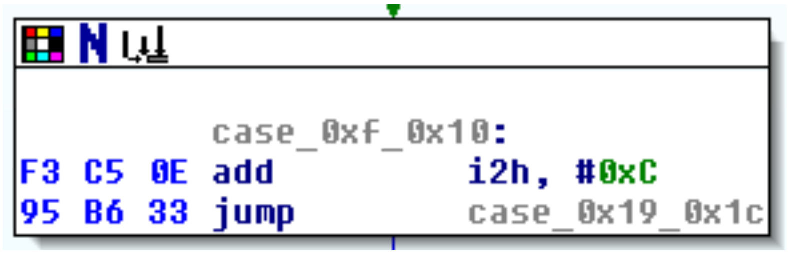
The function at 0xdaeb handles a subset of SBS commands from 0x1-0x1c. It uses many conditionals to figure out what to do based on the SBS command which is contained in the i2h register. SBS commands 0xf and 0x10 are handled in the same case. Commands 0x1b and 0x1c are handled in the same case along with 0x19 and 0x1a. In order to make SBS command 0xf and 0x10 behave like 0x1b and 0x1c, we just have to add 0xc to i2h and jump to the other case (at 0xdc3e).



```
                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
```

Figure 23: pre-patch.

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



```
         case_0xf_0x10:
F3 C5 0E add         i2h, #0xC
95 B6 33 jump        case_0x19_0x1c
```

Figure 24: Post patch

Now, no matter how much charge the battery has, it will always report that it has 4650mAh out of a possible 5068mAh.

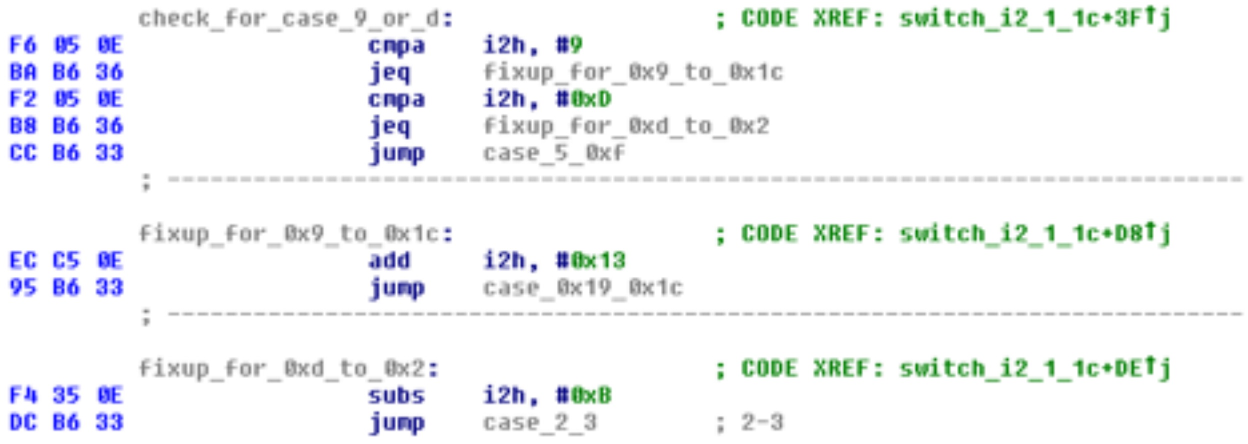
Change Voltage and RSoC

You can make similar changes to how the battery responds to Voltage and RSoc, which are controlled by SBS commands 0x9 and 0xd, respectively. These are a little harder to change since they're grouped in with all the SBS commands from 0x5 to 0xf. We utilize some of the space in the firmware from the previous case that we made obsolete after the jump we added. For now we send command 0x9 to what is returned by Full Charge Capacity and RSoC, which reflects a percentage of charge remaining, to Remaining Time Alarm, which by default is always 10.

First we change the bytes that aren't being used anymore.

```
    int worked = 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);
```

These bytes correspond to:



```
          check_for_case_9_or_d:                  ; CODE XREF: switch_i2_1_1c+3F↑j
F6 05 0E                      cmpa    i2h, #9
BA B6 36                      jeq     fixup_for_0x9_to_0x1c
F2 05 0E                      cmpa    i2h, #0xD
B8 B6 36                      jeq     fixup_for_0xd_to_0x2
CC B6 33                      jump    case_5_0xF
          ; ---------------------------------------------------------------

          fixup_for_0x9_to_0x1c:                  ; CODE XREF: switch_i2_1_1c+D8↑j
EC C5 0E                      add     i2h, #0x13
95 B6 33                      jump    case_0x19_0x1c
          ; ---------------------------------------------------------------

          fixup_for_0xd_to_0x2:                   ; CODE XREF: switch_i2_1_1c+DE↑j
F4 35 0E                      subs    i2h, #0xB
DC B6 33                      jump    case_2_3        ; 2-3
```

Figure 25: Patched code.

Then we just need to redirect flow of the case statement for 0x5 to 0xf to this new code.

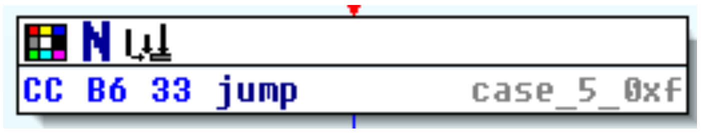

```
CC B6 33 jump        case_5_0xf
```

Figure 26: Jump before

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

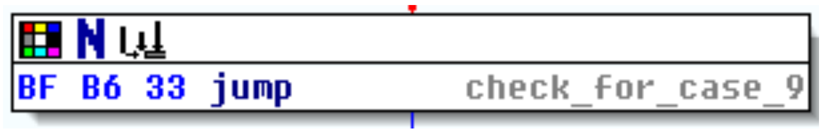

```
BF B6 33 jump        check_for_case_9
```

Figure 27: Jump after

We can now see that it is giving incorrect values for Voltage, Remaining Capacity, and Full Charge Capacity.

```
printf("Voltage: 0x%02x\n", read_word(kVoltage));
printf("Remaining Capacity: 0x%02x\n", read_word(kRemainingCapacity));
printf("Full Charge Capacity: 0x%02x\n", read_word(kFullChargeCapacity));
printf("Got manufacture date 0x%x\n", read_word(kManufactureDate));
printf("Got serial number 0x%x\n", read_word(kSerialNumber));
```

Prints out:

```
Voltage:                0x13cc
Remaining Capacity:     0x122a
Full Charge Capacity:   0x13cc
Got manufacture date 0x122a
Got serial number 0x13cc
```

At this point we can spoof any of the values that are queried from the Smart Battery. This does affect the amount of current delivered to the battery.

Conclusions

Smart batteries are responsible for reporting on battery status and charging conditions. They also enforce many of the safety features which prevent overcharging and over heating of the battery. The Smart Battery Charging Specification calls them the primary intelligence of the Smart Battery System.

When batteries are shipped from the factory, they are in the sealed state. This prevents modification of values used by the battery. In order to unseal the battery, the user must authenticate to it, either using a password or some other cryptographic primitive, depending on how it is configured. In order to enter full access mode, another password is required. Full access mode allows entering Boot ROM mode, wherein the firmware may be read or written to, subject to a firmware checksum which makes sure accidental modifications are not made.

For the batteries that ship with all the Apple laptops I tested, the password to unseal the battery and the password to enter full access mode are the hard coded values provided in Texas Instruments documentation. I this work, I provide API functions which can be used to communicate with the battery. This allows the ability to make arbitrary configuration changes as well as dumping of the data flash and instruction flash. I provide IDA Pro scripts to disassemble the machine code from the firmware. We provide a way to disable the firmware checksum as well as to make arbitrary changes to the smart battery firmware. Due to the nature of the Smart Battery System, changes made to the smart battery firmware may cause safety hazards such as overcharging, overheating, or even fire.

Special Thanks

Thanks to the many people who helped with this project, including KF, S7ephen, Barnaby Jack, Dion Blazakis, and Ralf-Phillipp Weinmann, and Joe Grand.

Bibliography

SLUU276 - bq20z80-V110 + bq29312A Chipset Technical Reference Manual: great description of system http://focus.tij.co.jp/jp/lit/er/sluu276/sluu276.pdf

SLUA364B - Theory and Implementation of Impedance TrackTM Battery Fuel-Gauging Algorithm in bq20zxx Product Family

SLUA375 -Impedance Track Gas Gauge for Novices

SMBus Specifications - http://smbus.org/specs/smbus20.pdf

MU092x http://www.mana.com.hk/pdf/Panasonic/Thermal%20Fuse/EYP4MU092X_spec.pdf

Smart Battery Charger Specification http://sbs-forum.org/specs/sbc110.pdf

bq803xx ROM API v 3.0 http://focus.ti.com/lit/ug/sluu225/sluu225.pdf. This isn't exactly the right API for our chipset, but it is close enough.

bq20z80EVM-001 Battery Management Solution Evaluation Module http://focus.ti.com/lit/ug/sluu205a/sluu205a.pdf

CoolRISC 816 8-bit Microprocessor Core Hardware and Software Reference Manual http://xemics.com/docs/xe8000/coolrisc816_databook.pdf

Charging Lithium-Ion Batteries: http://batteryuniversity.com/learn/article/charging_lithium_ion_batteries