The missing

Samsung EVO 840 - 250 GB SSD

Repair Manual

First version: July 2016
Current version: 27.04.2018

Introduction

Unfortunately my Novena laptop got physically damaged, so I took the SSD (Solid-State-Disk) from the Novena to a data-recovery company, who analyzed the SSD and discovered that the SSD has a problem in its startup phase that is most likely due to a firmware corruption, but the physical damage is actually not a problem for the SSD.
According to the company it is very likely that all other Novena SSDs (in Laptops, Desktops and Heirlooms) with the same firmware have the same problem and might also loose data.
Therefore I started to work on analyzing the firmware, and the firmware update process.

I have created the following guideline how you can extract and verify the firmware from your SSD, so that we can find out together, which models are actually affected.
I will try to explain what you can do to protect your own SSD, or recover if the same or a similar problem happens to your SSD.

What you should do now:
* **Backup your data!** But not to a SSD. SSDs seem not to be very well suited for backups from my point of view, I would suggest to use HDDs (Hard-Disk-Drive) instead.

What you should do when such a problem happens to your SSD:
Check back here, perhaps we have a solution for it already.
If you need urgent access to your data, search for a data recovery company that has a PC-3000 SSD from AceLabs available, at the moment of this writing, I think it is likely that they could recover your data with it.

This is how the problem of my SSD looks like from the kernel point of view (the following is the output from the “dmesg” command under Linux):

```
[    1.203395] ahci-imx 2200000.sata: fsl,transmit-level-mV value 1025, using
00000024
[    1.203432] ahci-imx 2200000.sata: fsl,transmit-boost-mdB value 0, using
00000000
[    1.203464] ahci-imx 2200000.sata: fsl,transmit-atten-16ths value 8, using
00002800
```

```
[    1.203494] ahci-imx 2200000.sata: fsl,receive-eq-mdB not specified, using
05000000
[    1.203543] ahci-imx 2200000.sata: Looking up target-supply from device tree
[    1.206436] ahci-imx 2200000.sata: SSS flag set, parallel bus scan disabled
[    1.206494] ahci-imx 2200000.sata: AHCI 0001.0300 32 slots 1 ports 3 Gbps 0x1
impl platform mode
[    1.206531] ahci-imx 2200000.sata: flags: ncq sntf stag pm led clo only pmp
pio slum part ccc apst
[    1.208050] scsi host0: ahci_platform
[    1.208473] ata1: SATA max UDMA/133 mmio [mem 0x02200000-0x02203fff] port
0x100 irq 71
[...]
[    6.592593] ata1: link is slow to respond, please be patient (ready=0)
[   11.212587] ata1: COMRESET failed (errno=-16)
[   16.602585] ata1: link is slow to respond, please be patient (ready=0)
[   21.222578] ata1: COMRESET failed (errno=-16)
[   26.612582] ata1: link is slow to respond, please be patient (ready=0)
[   56.252588] ata1: COMRESET failed (errno=-16)
[   56.261143] ata1: limiting SATA link speed to 1.5 Gbps
[   61.292587] ata1: COMRESET failed (errno=-16)
[   61.301345] ata1: reset failed, giving up
[   61.310158] ahci-imx 2200000.sata: no device found, disabling link.
[   61.319008] ahci-imx 2200000.sata: pass ahci_imx..hotplug=1 to enable hotplug
```

So it seems the SATA (**S**erial **AT** **A**ttachment) host tries to initiate the communication with COMRESET (somewhere between second 1.206531 and second 1.208473), but it never receives the COMINIT answer the SSD should give. 5 seconds later at second 6.592593 is complains that there is still no answer and a minute later at second 61.301345 it gives up. Theoretically the COMINIT should come within a second after the COMRESET.
If you are interested in the details of the SATA protocol, I can recommend this presentation:
http://de.slideshare.net/niravdesai7121/sata-protocol
On page 30 you can see the timing of COMRESET and COMINIT. The SATA host (computer) should start with COMRESET, and the SSD should reply with COMINIT.
Afterwards they can calibrate, negotiate the speed, … and then they have a “link”.

From OCZ SSDs I have heard that they have some kind of panic lock, where the SSD stops booting to prevent further damage when it discovered a potential problem. This behaviour on OCZ SSDs is often triggered when waking up a Laptop from hibernation. It could be that this is an instance of a similar panic lock for Samsung SSDs.

Layer 0 - Physical layer

Ok, let's start with the physics.
The SSD is contained in a metal case, to open it you have to open 3 screws, one of them is visible, the other 2 are behind the plastic sticker.
The screw has a Pentalobe format, so you need a Pentalobe screwdriver.
I found a working one in the https://www.ifixit.com/Store/Tools/64-Bit-Driver-Kit/IF145-299-1
(It's a bit funny that you need a 64 Bit Driver kit to access a 32 Bit CPU ;-)
When you have opened it, the SSD PCB (Printed-Circuit-Board) should look like the one on the upper right hand side of this image:

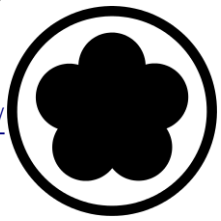
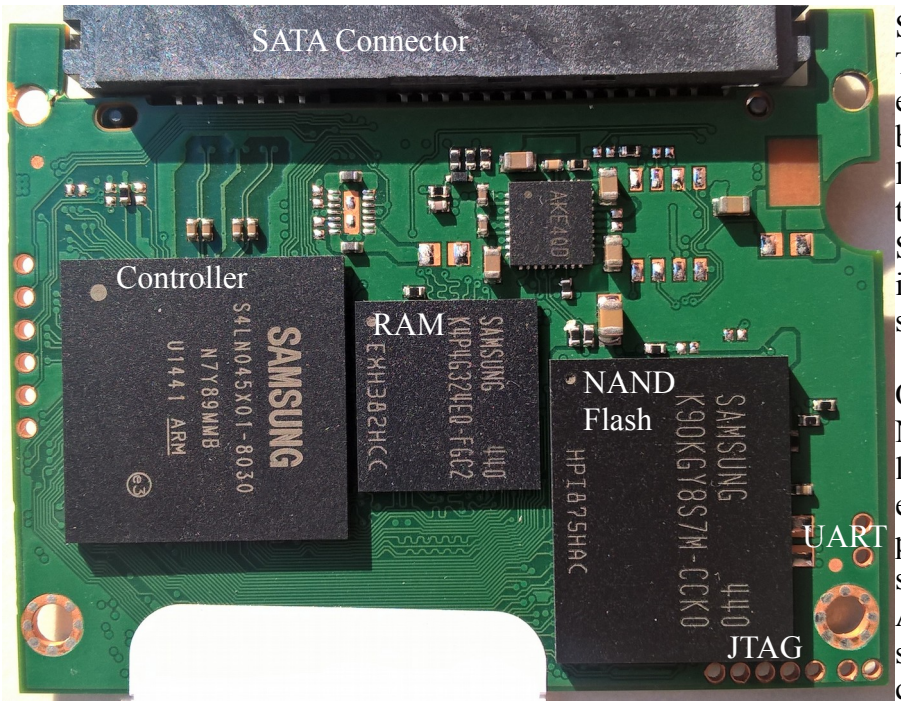


Image 1: Pentalobe



So what do we have here?
The long thing on the upper end is the SATA connector, the big square chip on the bottom left is the controller CPU, in the middle we have some SDRAM and on the right side is the actual NAND Flash storage chip.

On the back side is the second NAND Flash chip in the same location (I guess that makes it easy to route the traces, perhaps the address lanes are shared)
Another thing that strongly suggests to put 2 NAND Flash chips on directly opposite sides of a PCB is the temperature sensing. You want a connection that is as short as possible from the temperature sensor to the NAND flash chips where the temperature has to be measured. The easiest way to achieve that is by placing the NAND flash chips at the same location, and the temperature sensor directly next to both of them. (The temperature connection is done by the ground plane in the PCB or a very wide trace)

The controller CPU (the one one the left side of the image) is called Samsung MEX (the product number is S4LN045X01-8030), it has 3 ARM Cortex R4 cores, which are based on the ARMv7-R architecture, and likely have about 400 MHz. MEX is the fifth generation of the chip, MAX was the first one, MBX the second one, ...
The following things are written on the controller CPU:
SAMSUNG S4LN045X01-8030
N7Y89MMB
U1441 ARM → 1441 means it was produced in the year 2014/Week 41
It is said to have an 8 channel controller, if you want to learn more about what that means, I suggest the following page:
http://www.cactus-tech.com/en/resources/blog/details/solid-state-drive-primer-8-controller-architecture-channels-and-banks

The SDRAM chip:
Samsung 512 MB Low Power DDR2 SDRAM
Samsung, 4Gb, LPDDR2 SDRAM, 1CH x 32, 8 banks, 134-FBGA, MONO, 1066Mbps, 1.8V/1.2V/1.2V:
512MB LPDDR2 DRAM:
The following things are written on it:
SAMSUNG 440
K**4**P**4G**32**4**E**Q**-F**G**C2
K:=Memory
4:=DRAM
P:=LPDDR2 (guess)
4G:=4G, 8K/64ms Density
32:=x32 Bit Organisation

```
        4:=8 internal Banks
         E:=Interface ?
          Q:=SSTL-2 1.8V VDD, 1.8V VDDQ
           -F:=7th Generation
             G:=FBGA Package
              C:=Commercial, Normal Temperature&Power range (0-
95°C)
```
EXH382HCC

And now the NAND chips that actually hold your data:
NAND TLC 128 GB: (19nm Toggle Mode 2.0 TLC (3-bit per cell) NAND (Model# K90KGY8S7M-CCK0))
SAMSUNG 440
```
K90KGY8S7M-CCK0
K:=Memory
 9:=NAND Flash
  0:=3-Bit MLC (TLC)
   KG=128G
     Y8=Organisation x8?
       S=Voltage ?
        7=Mode ?
         M=1st Generation
           C=CHIP BIZ D : 63-TBGA
            C=Commercial, Normal(0°C-95°C) & Normal Power
             K=Customer Bad Block ?
              0=Pre-Program Version:None
```

I measured the voltages and connectivity of a very similar PCB which uses the same Controller and slightly different Flash chips to create a pinout:
http://www2.futureware.at/~philipp/ssd/K9CHGY8S5M-CCK0-Pinout.pdf
The chip has 0.8mm pitch and BGA-316 form factor.

K9CHGY8S5M-CCK0 Pinout

SAMSUNG K9CHGY8S5M-CCK0 Pinout

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
A																	A
B				1,84	GND	1,83	GND	0,91	2,95	GND	2,95	GND	2,95				B
C			GND	2,96	GND		0,9			1,84	GND			12,3			C
D		2,96	GND				0,9			GND	GND	GND	GND	GND	2,96		D
E		1,83	GND			0,59	0,9			GND	1,83		GND	GND			E
F		GND	GND	1,83	GND	0,58	0,89				1,83		GND	GND	GND		F
G		1,83	GND	2,96	GND					1,84	1,83	1,83		GND	2,96		G
H		2,96	GND	1,83	GND	1,83	1,82					1,83		GND			H
J		1,83	1,84	GND	0,58	0,57	1,83					1,84	0,2	GND	GND		J
K		1,83	GND	GND	0,57	0,59	1,83			1,83		1,84		GND	2,95		K
L			GND		0,6	0,59	1,83			1,83	0,89	0,97		GND	1,83		L
M		1,84	GND		0,59	0,59	1,83			1,83	0,9	0,97		0,91			M
N		2,96	GND		1,83					1,83	0,91	0,95	GND	GND	1,84		N
P		GND	GND		1,83						0,91	0,94	GND	1,82	1,84		P
R		2,95	GND		1,83					1,82	1,82	GND	1,84	GND	2,96		R
T		2,96	GND		1,83	1,83	1,84					GND	2,96	GND	1,84		T
U		GND	GND	GND		1,84	GND			0,87	0,87	GND	1,84	GND	GND		U
V			GND	GND		1,83	GND			0,87	0,9	0,2		GND	1,83		V
W		2,95	GND	GND	GND	GND	GND			0,87	0,89	0,1	0,19	GND	2,96		W
Y			12,3			GND	1,83			0,88	0,87	GND	2,96	GND			Y
AA				2,95	GND	2,96	GND	2,96	0,91	GND	1,83	GND	1,84				AA
AB																	AB
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	

Yellow: this pin is connected to the same pin of another flash chip
e.g. Power-Lines are usually connected the same way (yellow)
e.g. Chip-Select lines are connected differently (gray)
1,83-1,84 Volt power rail, all connected together on the PCB
2,96 Volt Power-Rail, all connected together on the PCB
12 Volt Power-Rail, all connected together on the PCB

Form-Factor: BGA-316, 16 CE_n assignments for quad 8-bit data access
Pin Reduction does not seem to be used

The only product selection guide that contained relevant information that I could find was from 2010, so it did not contained all variations, but it is still a helpful guide to understand Samsung´s naming conventions:
http://www.samsung.com/global/business/semiconductor/file/media/SamsungPSG_july2010_final-2.pdf - Page 16
If anyone can provide any further information to explain the variations we have here, I would be interested to hear.

Unfortunately at first I couldn't find datasheets for any of the other chips used on the SSD.

There are a few smaller chips on the PCB: JS4TAA and AKE4QD, "ABS 431 .WD"
There is also a Chip named "GUILL TI 48" on it, which is from Texas Instruments
https://chipworks1.force.com/DefaultStore/ccrz__ProductDetails?viewState=DetailView&cartID=&sku=TEX-GUILL_PKG_2&store=DefaultStore

fzabkar from forum.hddguru.com has identified the chips and provided the following information:

> “The mystery components appear to be marked especially for Samsung.
> JS4TAA appears to be a 5V STEF4S electronic fuse manufactured by STMicroelectronics. "JS4" appear to be the important characters in the part number.
> GUILL is a TPS62130D2 synchronous step-down DC-DC converter manufactured by Texas Instruments.
> AKE40D appears to be a multiple-output switchmode DC-DC converter, probably with integrated power sequencing. "AKE" appear to be the important characters in the part number.
> I'm guessing that the ABS part is an SPI flash memory.”

The ABS part is actually not a SPI flash, but a I2C device, and my current guess is that it is the temperature sensor, similar to those devices:
http://www.ti.com/lit/ds/symlink/tmp275.pdf
http://www.nxp.com/documents/data_sheet/LM75A.pdf
http://ww1.microchip.com/downloads/en/DeviceDoc/25095A.pdf
I logged the following I2C communication with an Oscilloscope:
I2C
Time,Dir,ID,Data,ACKed,
1.0870288E-01,Write,18,08 02,Y,
1.0890320E-01,Write,18,00,Y,
1.0903376E-01,Read,18,00 77,N,
1.0923408E-01,Write,18,01 02 29,Y,
1.0949680E-01,Write,18,04 05 50,Y,
1.0975952E-01,Write,18,04,Y,
1.0989008E-01,Read,18,05 50,N,
1.1008976E-01,Write,18,02 05 00,Y,
1.1035280E-01,Write,18,02,Y,
1.1048336E-01,Read,18,05 00,N,
1.1068304E-01,Write,18,03 04 B0,Y,
1.1094608E-01,Write,18,03,Y,
1.1107664E-01,Read,18,04 B0,N,

On (another?) SSD I got the following I2C traffic for every Smart command:
Write 0x30 Data 05h
Read 0x31 Data 22h 2Ch
So 0x05 is obviously the command to read out a single temperature value, and for the temperature values we read, we got the following table:
22 A8 = 42°C
22 88 = 40°C
22 48 = 36°C
22 3C = 35°C
22 38 = 35°C
22 34 = 35°C
22 2C = 34°C
22 28 = 34°C
22 20 = 34°C
22 20 = 34°C
22 20 = 34°C
22 24 = 34°C
22 1C = 33°C
22 1C = 33°C
22 18 = 33°C

With the 2 constants 0x30 followed by 0x31 I was able to identify the I2C code in the firmware.

On the various SSDs from Samsung the ABS part, or a similar chip is always located directly besides the Flash chips on the PCB, which is necessary to measure the temperature. (You could put a SPI flash anywhere on the board)

I found the following document which explains the thermal management strategy of Samsung with their newer 950 series starting at page 13:
http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_950_PRO_White_paper.pdf
So the thermal management that is used in the EVO840 is likely similar but not that advanced.

Later on the actual “ABS” chip was found by fzabkar:
MCP9844T-BE/MNY, Microchip, +/-1 degC, 1.8V Digital Temperature Sensor, marking ABS, TDFN-8:
http://ww1.microchip.com/downloads/en/DeviceDoc/20005192B.pdf

Layer 1 - Link layer

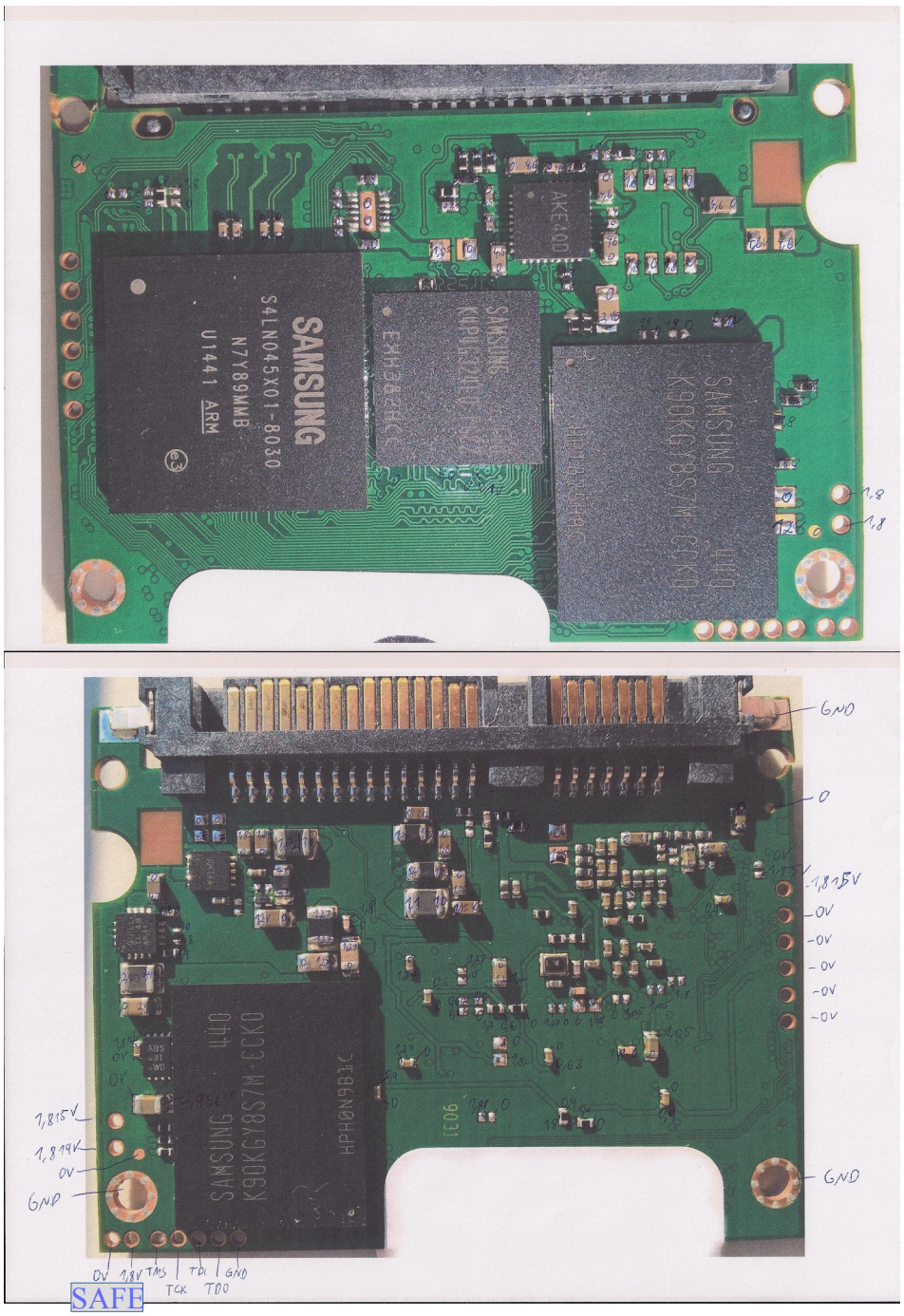
SSD's need to be electrically powered, this one needs to be powered through the SATA port.
At first I used a USB-SATA adapter for supplying power.

One relevant topic regarding the power-supply is the grounding for the JTAG port:
I tried to attach the USB-SATA adapter to the USB power of a secondary laptop, and there I measured a nearly constant 2 Volt difference between the GND (ground) of the SSD and the GND of the Novena.
The signal level of the GPIOs of Novena is 3.3 Volts, so I thought that +- 2 Volts could be too much for proper signaling reception.
When I plugged the USB adapter into a USB port of the Novena, both Novena and SSD had the same GND level (0 Volt difference).
The downside of the USB-SATA adapters is that you do not get access to the initialization communication so PCI SATA adapter or SoC is preferred if you want to analyze the SATA behavior.

I powered the SSD and measured all the voltages on all the Pins I could find, first on a good SSD, documented on printed up-scaled photos and then verified them on my broken SSD, and the voltages were all OK.

The voltages on the SATA Data pins were identical too, so I have some more confidence that the problem is not a physical problem of the SATA interface, and that all the power-management is good.

You can measure the power with a multi-meter on various places. Make sure that you configure your multi-meter correctly to read voltages, and that you always touch only a single pin, that you do not short out 2 pins next to each other. When you have sharp probe heads, make sure you do not push too hard on the pins, to not squish the pins!



I have created a separate PDF with both front-side and backside images where you should be able to better zoom in so that you can look at the details:
http://www2.futureware.at/~philipp/ssd/SamsungEVO840Voltages.pdf

Later on I took an Arduino with a 4-Relais shield and connected 2 power cables to it, so that I can switch on/off the SATA power supply (one relais for the 5V rail and one relais for the 12V rail, and finally one relais for the Safe-Mode) programmatically by sending switch on/off commands to the Arduino. Nowadays I would recommend a dedicated USB relais board with at least 4 ports, I don't like the connection between the Arduino and the relais board, it isn't very stable.
Controlling the SATA data port was done later on by using a USB-SATA converter connected to the USB port on the Novena, which can be controlled with the novena-usb-hub tool. (It enables/disables the power supply to the USB port)

JTAG

Now regarding the JTAG interface:
I found the JTAG pinout of the Samsung SSD on Twitter:
https://twitter.com/bsmtiam/status/623241828033114112
(Thanks a lot to bsmtiam!)
I found some more pins on the board:
There are 2 pins right next to the JTAG interface (which I marked 0V, 1.8V TRST on the second photo). The one next to the JTAG pins is a RESET line (TRST/RESET).

Later on I found that ACE Laboratory (makers of PC-3000) figured out the 4 pins next to JTAG:
The 2 pins right next to the JTAG (0V, 1.8V) pins are activating the “SAFE Mode” when shorted.
The other 2 pins (marked 1,815 V and 1,819V) are a UART port, which provides debugging functionality in SAFE Mode.

The big round connectors are GND.
On the other side, next to the CPU (big quadratic chip), there are 6 Test_Pins which I number the following way:
Pin#1 would be the one nearest to the SATA connector, PIN#6 is the one farthest away from the SATA connector.
Pin#1 is connected to a pin of an unpopulated footprint nearby. Pin#1 has 1.816V voltage (and 7MegaOhm to GND), the same is on the connected pad of the unpopulated footprint nearby. The other pad of the unpopulated footprint has 0V voltage and is connected with GND. (So the unpopulated footprint is likely for a Pull-Down Resistor or a Capacitor)
Pin#5 is connected to the "Optional" pin on the SATA power connector. This is likely a GPIO from the CPU. See http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_845DC_01_Device_Activity_Signal_DAS.pdf

Now where to get a JTAG adapter?
Well, the Novena has a few GPIOs that are pinmuxed (multiplexed) with the serial ports and are easily accessible:
https://www.kosagi.com/w/index.php?title=Novena_headers_to_linux_GPIO_mappings

I wanted to use the Novena as a JTAG master without a SATA disk, so that I could plug the disk-under-test into the SATA port if I wanted too, so I took a spare 4GB MicroSD card and installed the standard Novena image on it: http://repo.novena.io/novena/images/novena-mmc-disk-r1.img
I learnt about the pinmuxing in Linux called PINCTRL (https://www.kernel.org/doc/Documentation/pinctrl.txt), and learnt that it is defined in the device-tree.
I tried out the pins, and found that only 2 of the documented pins were actually usable as GPIOs, the other pins were not configured in the device-tree.
So I tried to update the kernel from the repository (https://www.kosagi.com/w/index.php?title=Updating_novena_repo_key) to the current 4.4 kernel, which also came with a new device-tree configuration, and suddenly I could drive 4 of the pins as GPIOs, which is sufficient for basic JTAG access.

If you have a Samsung 840 EVO but you do not have a Novena board, you can also use/buy a cheap Altera USB Blaster which is about 3 times slower than the Novena with sysfsgpio.
I also bought various other JTAG interfaces which did not work: In-Circuit Open-OCD, BAITE JTAG ICE, Raspberry Pi 3, OLIMEX ARM-USB-OCD-H all did not work. The Raspberry Pi3 showed signal integrity issues that might possibly be overcome with an improved driver.

So for the Novena, you need the following OpenOCD configuration:
interface sysfsgpio
transport select jtag
Each of the JTAG lines need a gpio number set: tck tms tdi tdo
sysfsgpio_jtag_nums 163 16 162 20

Then I wanted to try the RESET line, so I needed 1 or 2 more GPIOs. I changed the Device-Tree to pinmux the lines from UART to GPIO,
and documented everything on the Wiki:
https://www.kosagi.com/w/index.php?title=Novena_as_JTAG_adapter

One problem I found out later on is that Novena and Raspberry Pi GPIOs are somewhat slow, and that you might want to try faster JTAG adapters. With my Novena and OpenOCD I was able to read memory with 6 Kbyte/second. With the newer imx_gpio driver, it is raised to 8 Kbyte/second. I added a toggle command into OpenOCD and improved it to 13KB/s. The bottleneck with sysfsgpio is the Linux Kernel API for GPIOs, where you need 2 syscalls per toggle. Creating a faster Kernel interface and perhaps DMA accelerating the GPIOs should speed it up a lot. If anyone could implement that in the Linux kernel and in OpenOCD, that would be great.
Later on I bought and tried several other JTAG interfaces, but I had a lot of troubles with them, the Samsung SSDs seem to be very picky about signal integrity and voltages (1.8V!!!). So far, Novena and Altera USB Blaster (the normal ones, a Revision C marked one did not work for me) are still the only reliable JTAG adapters for Samsung SSDs.

Safe Mode

I found a product called PC-3000 on the market for data recovery that is able to recover data from Samsung SSD's: http://www.acelaboratory.com/pc3000-SSD.php through the SATA interface.

When you click on "Video Tutorials" and watch the Samsung video, it shows that you have to shorten 2 pins on the PCB and then power on the SSD to get the SSD into "Safe Mode". Several weeks later I found a better video:

http://blog.acelaboratory.com/pc-3000-ssd-samsung-family.html

The RESET line has to be shorted to the pin next to it during powerup, and afterwards there is a UART available on the other 2 pins.

From what I have read, back in the HDD world, Safe-Mode meant that the harddisk would not power up the motors to actually read/write on the harddisk, so you could safely talk to the HDD controller chip about it's firmware, without risking any power problems.
It was not quite clear for me, what Safe-Mode could actually mean for a SSD.
After some playing around with SAFE-Mode, I found out it means the following:
* Only the first Core mex1 is activated, mex2 and mex3 are unpowered in SAFE-Mode, which hints that in normal (non-SAFE) mode, mex1 is responsible for waking up mex2 and mex3 likely somewhere at the end of the initialisation.
* The SSD claims to have about 500 MB size, not 250 GB anymore, which resembles the RAM size in the SSD, so you most likely get direct access to the RAM this way (which is where the is read from/to on the way to the Flash). Reading anything didn't gave me any of the original contents, so it also seems as if the Flash isn't mounted, the Flash encryption is not active, …
* It always has the serial number SN000000000000, so it obviously tries not to read anything from the configuration that is stored in the Flash, so that it cannot crash during the initialisation due to garbage in the configuration memory.

On the ARM side of things, I found that there is a Processor PIN called VINITHI that is read(sampled) once during boot, which decides, whether the CPU should boot from the address 0x00000000 or from the address 0xFFFF0000. So theoretically this ARM Pin could be the Safe Mode pin, so that it boots from a different code there. But I did not see any memory available at 0xFFFF0000 during any of my traces, so this option seems to be unlikely. It could be that there is some memory mapped at 0xFFFF0000 during boot and unmapped again by the boot code, but I did not saw any hints about that yet, so I think that is not the method used for booting.
So if the Safe-Mode exists (yes, it does) on EVO 840 – 250 GB, and it is not the Processor Pin to run at 0xFFFF0000, then I think it would have to be a GPIO and the firmware would have to branch depending on the GPIO afterwards, which would be read by the Firmware during booting.
It finally turned out that the Safe-Mode pin is actually just one of many GPIO pins (UART, I2C), which is sampled by the first stage bootloader.

From the physical point of view: Since there are no >3 Volt Test-Pins available anywhere on the board, the connection between the 2 pins would likely have to pull the GPIO down to GND. To be able to measure a grounded Pin and to differentiate it from a free floating pin, a voltage has to be applied, something like 1.8 V would be what I would expect there.
If this voltage is only applied during booting, then an oscilloscope with triggering is needed to analyze it.

Layer 2 - Firmware

At first I searched on the Internet for Firmware updates.
On the Samsung website there is only the current firmware update available. But searching on the internet some more, I was also able to find previous firmware updates.

This is the current Firmware history I compiled:
1: EXT0AB0Q (~2013-07) (Original firmware, I could not find it on the internet)
2: EXT0BB0Q (2013-10)
3: EXT0BB6Q (2013-12-18 19:43) (currently available on the Samsung website)
4: EXT0CB6Q (2014-10-10 19:36) This is included in the Samsung_SSD840EVO_Performance_Restoration.zip
5: EXT0DB6Q (2015-03-27 18:35)

When you download the ISO image to update the firmware you get a file like this:
Samsung_SSD_840_EVO_EXT0BB6Q.iso
Inside the file there is the isolinux/btdsk.img , which can be unpacked with 7-Zip.
Inside the btdsk.img there are 3 interesting files:
samsung/DSRD/DSRDGUI0.EXE (firmware updater)
This EXE file is packed with WDOSX and can be unpacked with WDOSXUnpacker (which requires Python 2.7, it does not run unter Python3!):
https://raw.githubusercontent.com/0xDB/WDOSXUnpacker/master/WDOSXUnpacker.py
Then you have 2 files for the firmware:
samsung/DSRD/DSRD.enc (firmware update configuration file)
samsung/DSRD/FW/ext0bb6q/EXT0BB6Q.enc (firmware itself)

I decrypted half of it, which allowed me then to find a tool that is able to fully decrypt them:
https://github.com/ddcc/drive_firmware/blob/master/samsung/samsung.c
The obfuscation is a 4-Bit encryption of the upper nibbles in every byte.

I hope that this ishei not the only protection mechanism against malicious firmware-updates.

The firmware update configuration file contains a list of old firmware versions it can be applied to, and which new firmware version they should get, and I think whether a restart is needed.

At first I tried to decompile the firmware, but I saw that there are several problems with this approach: ARM has a Thumb mode, which is a different ISA (Instruction Set Architecture) that is only 16 bits per command, and it can switch between ARM (32 Bit) and THUMB (16 Bit) mode back and forth with every jump/call. So it depends on the compiler what code it wants to generate, and it is hard to guess whether a particular DWord is actually 1 ARM instruction or 2 THUMB instructions just from the raw binary. The usual method is to take one known entry point (assuming that the CPU normally starts in ARM mode, not in Thumb mode), and then follow to any jump/calls and learn from them whether the targets points are actually ARM or Thumb. But since I did not knew any entry points in the beginning, that approach did not looked good to me. The approach I decided to do is to single-step debug through the firmware through the JTAG interface, and to record for every single-stepped instruction, whether it is ARM or Thumb and then to feed those information from the traces back into the disassemblers and decompilers, which memory areas contain ARM code or Thumb code. (Look for .r2 files that are generated).
Another problem is the memory mapping. The firmware file consists of 10 parts that are loaded to different memory regions, and later on I found out that some of them are additionally mapped to other ranges, and that partly depends on the actual CPU core they are running on. This actual mapping is also interesting and helpful for disassembling/decompiling it.

Then I analyzed the format of the firmware file, and found that it contains a header, then 10 partitions (nicely aligned and not overlapping), and at the end some data that is outside the partitions. The previous firmware versions had a very similar structure, only a few partitions had slightly different sizes.

http://www2.futureware.at/~philipp/ssd/analyse/EXT0CB6Q.dec.html

Layer 3 - Communication

After succeeding to setup the JTAG link, I started OpenOCD and configured it. I created the following configuration files:

novena-jtag.tcl Novena JTAG configuration
novena-swd.tcl Novena SWD configuration (this is not applicable to this SSD, but if you want to analyze some other board which has a SWD interface with your Novena, this file is what you need.
samsung-mex.tcl Samsung MEX configuration

You then run OpenOCD with both configuration files:
openocd -f novena-jtag.tcl -f samsung-mex.tcl
I have copied both files into a openocd.cfg so that I can start openocd with "openocd -f openocd.cfg"
(If you are using a raspberry instead of the Novena, you would use some configuration for raspberry together with the samsung-mex.tcl instead of the novena-jtag)

JTAG SCAN:

Instructions 0-7 all seem to be only one Bit long, which is always zero. But it could also be 0 bits long.

```
> irscan auto0.tap 0 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 0 ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 1 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 1 ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 2 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 2 ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 3 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 3 ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 4 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 4 ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 5 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 5 ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 6 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 6 ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 7 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 7 ; drscan auto0.tap 64 0x0
0000000000000000
```

Now it gets interesting starting from instruction 8 (0100): Here we have a 32/37 Bit register. This is called the JTAG_DP_ABORT according to the specs.

```
> irscan auto0.tap 8 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF800000000
> irscan auto0.tap 8 ; drscan auto0.tap 64 0x0
0000000000000000
```

Instruction 9 is 1 bit long again:

```
> irscan auto0.tap 9 ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE
> irscan auto0.tap 9 ; drscan auto0.tap 64 0x0
0000000000000000
```

In Instruction 10 we have actual data, this instruction is possibly used for accessing the ARM cores, this is the JTAG_DP_DPACC register:

```
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF800000002
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0x0
0000000000000002
```

In instruction 11 we have even more interesting data, which is changing, this is the JTAG_DP_APACC register:

```
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0
0000000000000002
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF81843081A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF80000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF80000009A
```

Instructions 10 and 11 have some strange set/reset logic in them:

```
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF80000009A
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF800000002
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFF800000002
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0x0
0000000000000002
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0x0
0000000000000002
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0x0
0000000000000002
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
0000000000000002
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0xffffffffffff
FFFFFFF80000009A
```

> irscan auto0.tap 0xb ; drscan auto0.tap 64 0xffffffffffff
FFFFFFF80000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0xffffffffffff
FFFFFFF80000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xb ; drscan auto0.tap 64 0x0000000000000
000000000000009A
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0xffffffffffff
FFFFFFF80000009A
> irscan auto0.tap 0xa ; drscan auto0.tap 64 0xffffffffffff
FFFFFFF800000002

Instruction 12 and 13 seems to be one zero bit again:
> irscan auto0.tap 0xc ; drscan auto0.tap 64 0x0
0000000000000000
> irscan auto0.tap 0xc ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFFFFFFFFFE

Instruction 14 is the IDCODE:
irscan auto0.tap 0xe ; drscan auto0.tap 64 0xffffffffffffffff
FFFFFFFF4BA00477

And Instruction 15 (which should be BYPASS) seems to be 0:

If I didn't overlooked something, it seems that there is no Boundary-Test support available on the Samsung MEX :-(this would have been helpful to test the various traces on the PCB, whether they might be broken/shorted/…
I was hoping for a EXTEST/SAMPLE support to be able to boundary test the BGA balls, to make sure there is no physical problem there, like cold solder joints, unconnected BGA balls, ...
If anyone from Samsung or other vendors reads this: Please make sure in the future that your chips have Boundary-Scan available and provide at least BSDL files in your repair manual for professional failure analysis, even if there is only one Boundary scan-able chip on it, and Boundary Scan does not seem to make much sense on first sight.
Later on with the SAFE-Mode, I discovered that the SATA Port physically works properly, so the boundary scan is not necessary for me anymore, but I still believe that it would be helpful for other customers to be able to boundary test the devices in the field, to better diagnose problems.

Layer 4 - Memory

I bought a similar SSD, which luckily had the same PCB design, same chipsets and even the same Firmware version (EXT0CB6Q).
So I dumped the whole memory (tried through the full 4 GB address space of the 32 Bit processor).
I decided to separate the whole memory into 64KByte blocks. (Divide & Conquer)

I tried to read every 64KByte block. If the block was readable, it would take approximately 10 seconds (6 KB/s through JTAG. I would be interested if anyone got better performance with different JTAG adapters).
(Note for future projects: 64 KByte Blocks is perhaps too large, there might be smaller memory blocks somewhere.) In normal mode, I did not discover any smaller blocks, only in SAFE mode, I found a 32 KByte Block, or perhaps it was 96 KByte.
If the block was not readable, it would take approximately 1 second.

In parallel, I developed a single-stepper that traces through the firmware, and logs the Program-Counters and ARM<->Thumb instruction set changes, to get a feeling for where the CPU usually hangs out, and where code-regions are, and which instruction set they are using. The information about the instruction set is important for disassemblers like radare2, which have a hard time to guess the instruction set from raw firmware files otherwise.

The broken SSD continuously cycles through the following locations:
0x0081b814, 0x0081b816, 0x0081b818, 0x0081b81a, 0x0081b824, 0x0081b826, 0x0081b82a

The following is a dump of the register values:
(0) r0 (/32): 0x203B0000 (dirty)
(1) r1 (/32): 0x00000000
(2) r2 (/32): 0x00008000

```
> mex arm disassemble 0x0081b810 20 thumb
0x0081b810  0x408a          LSLS     r2, r1
0x0081b812  0xe007          B     0x0081b824
0x0081b814  0x68c1          LDR      r1, [r0, #0xc]    (Read word from
0x203B0000+0xc=0x203B000C)
0x0081b816  0xb289          UXTH     r1, r1            ; Clears the upper 16 Bits
0x0081b818  0x4211          TST r1, r2                 ; Checks for 0x8000
0x0081b81a  0xd003          BEQ      0x0081b824        ; It is usually equal
0x0081b81c  0x68c1          LDR      r1, [r0, #0xc]
0x0081b81e  0xf362010f      BFI r1, r2, #0, #16
0x0081b822  0x60c1          STR      r1, [r0, #0xc]
0x0081b824  0x68c1          LDR      r1, [r0, #0xc]    ; so we continue here, Loading the value again
0x0081b826  0xea124f11      TST.W    r2, r1, LSR #16   ; Test r2 with r1 shifted left (?)
0x0081b82a  0xd0f3    BEQ      0x0081b814              ; Branch if Equal, this is usually equal
0x0081b82c  0x4770          BX  r14
0x0081b82e  0x0000          LSLS     r0, r0, #00
0x0081b830  0x0000          LSLS     r0, r0, #00
0x0081b832  0x2038          MOVS     r0, #0x38
0x0081b834  0xb570    PUSH     {r4, r5, r6, r14}
0x0081b836  0x4604          MOV      r4, r0
0x0081b838  0x4615          MOV      r5, r2
0x0081b83a  0x0780          LSLS     r0, r0, #0x1e
```

Sometime later, I discovered that the different CPU cores have an at least slightly different view on the memory:

```
> mex1 mdw 0x40825088
0x40825088 44465331                        1SFD
> mex2 mdw 0x40825088
data abort at 0x40825088, dfsr = 0x0000000d
```

embedded:startup.tcl:21: Error: error reading target @ 0x017aa
> mex3 mdw 0x40825088
data abort at 0x40825088, dfsr = 0x0000000d
embedded:startup.tcl:21: Error: error reading target @ 0x017a4

These differences need further investigation.
The current results intermediate results are the following:
ATCM (A-Tightly-Coupled-Memory), starts at 0x00000000 and is 96-128KB large, this is individual memory per CPU-core. Core-1 gets the ROM with the bootloader instead of the ATCM at 0x00000000 when loading in SAFE-Mode. I haven't figured out how this is implemented, and haven't found a mechanism how this memory mapping could be changed from the CPU.
BTCM is loaded at 0x800000 and is 8MB shared between all the CPUs.

Later on, I discovered that the internal SPI flash is most likely 128KB in size, and it contains only the firmware that is used in SAFE mode, it does not contain any keys or the normally used firmware. The normally used firmware is stored in the so called SA area on the big NAND flash chips, most likely directly at the beginning of the NAND flash.

Layer 6 - Multi-Tasking

To get more information about the SSD, I started to do some behavioral analysis:
I applied different workloads:
* Reading sectors from the SSD
* Writing sectors to the SSD
* Reading geometry info from the SSD
* TRIMing sectors on the SSD
(* Firmware update - I have not tried that yet)
Those workloads are implemented in the workload_*.sh files.

Then I tried to selectively turn off and on the 3 CPU cores of the Samsung SSD through JTAG.
For reading and writing, I discovered that all 3 cores are needed.
If I halt any of the cores, the data transfer is stalled and it can only complete (or would timeout eventually) if I resume all 3 cores again.
Then I wanted to find out in which order the cores work together.
So I developed a tool that can try all possible variants of turning on and off the various cores(cores1.pl), and find out which is the shortest path to successfully complete the workload (corestrategy.pl).
When I tried this on a hdparm "geometry info" function (which queries from the SSD information about the size and geometry of the disk), it gave the result that it first needs a timeslice on core mex1 and then a timeslice on mex2, mex3 is not needed.
(And mex1 and mex2 do not need to be on at the same time)
Which brought me to the following theory:
mex1 is responsible for receiving data through SATA
mex3 is responsible for communicating with the NAND Flash, where the actual data is stored
mex2 is responsible for replying back through SATA
For reading or writing a sector from the SSD, the following timeslices are needed: 1->2->3 or 1→3→2.
So mex1 must come first (to parse the SATA request), then 2 and 3 must come afterwards, interestingly in any order?!? (This fact leads to the hint that there must be a SATA core, which is independent of the 3 ARM cores, which sends the response as soon as both cores have committed their parts to it.

(You can try those with the corestrategy.pl, which calls cores1.pl)

So all those 3 tasks can be handled (mostly) independently in parallel.
Later on it turned out that the cores have different roles than I thought: mex1 is responsible for taking commands from SATA (and likely also PCIe), it does initial parsing and then delegates to mex2 and mex3 for load balancing. mex2 and mex3 are nearly identical, and each of them controls 4 flash channels, which make 8 flash channels in total. mex2 and mex3 both can finalize a SATA response. Even later I found out that mex1 is likely called “HCORE” which stands for HOST-CORE, the core that interfaces to the host, the computer. And mex2 and mex3 are called FCORE, FLASH-CORE, they are responsible for interfacing with the FLASH. But since that information came too late, I will continue calling them mex1, mex2 and mex3 in this paper.

Layer 7 - Application

What I am primarily interested is resurrecting the SATA PHY (the physical interface for the SATA interface). But where is it?
So I searched for the SATA PHY.

One problem I learned with the workload testing above is that there are various timeouts in the SATA protocol, so that any request for reading/writing a block should finish within seconds rather than minutes.
Tracing the whole handling of the SATA controller during such a request is too slow.
And when the timeouts are reached, in some situations the computer and the SSD are getting so much out-of-sync that they are not able to recover the communication anymore,
and I have to manually disrupt the SATA communication by unpowering the USB-SATA bridge.
(This could be perhaps automated by Novenas USB-Hub powering through GPIOs, I haven't tried that yet)
So I transported the idea of DNA-sequencing into Debugging:
I designed a workflow to continuously read a single sector from the SSD with a distinctive address:

```
while true
do
dd if=/dev/sda of=/dev/null count=1 skip=1251255 #0x1317b7
#dd if=/dev/sda of=/dev/null count=1 skip=4235125 #0x409F75
done
```

While running this workflow, I usually let all the cores run. I randomly interrupt one core, trace and log 30 instructions, then I resume the core, so that it can fulfill the request and still be hard-realtime compliant. Similarly to Gene sequencing, I get short snippets of 30 instructions each,
which are randomly overlapping, which can be puzzled together again. (See puzzle.pl)
Some benchmarking: 1 Instruction: 3 seconds (mostly overhead), 10 Instructions: 5 seconds, 30 Instructions: 12 seconds, 100 Instructions: 31-44 seconds

One thing that helps a lot there is that I initially decided to log with each instruction what the next instruction after it actually is. So it would even work with single instruction snippets.
So I ran the workload and collected snippets, and then I searched through the snippets I got for the distinctive address I used for reading the block, and I found parts of it.

```
0x00000af8->0x00000afa Thumb Supervisor 0x00000af8  0x78c9      LDRB    r1, [r1, #0x3]
r1:0x00800D80=>0x00000025 r1:0x00800D80=>0x00000025 [0x00800D83]=1317b025
```

Then I changed the address to a different and distinctive address, and then the new address showed up in the same places:

0x00000af8->0x00000afa Thumb Supervisor 0x00000af8 0x78c9 LDRB r1, [r1, #0x3]
r1:0x00800DA0=>0x00000025 r1:0x00800DA0=>0x00000025 [0x00800DA3]=**409f**7025

The LDRB instruction only loads a single byte, but my tracer always reads 32 Bits (a DWord) from the memory location that is accessed. So we are accidentally finding the sector address here, and the CPU is only interested in a single byte next to it, which turned out to be the SATA request command byte.

So with trace sequencing, I found where the SATA PHY hands over the requested block address to the CPU:
0x00800DA0 seems to be one of the base addresses of an incoming SATA Request.
0x00800DA3 (base_addr+3) contains a byte with the SATA request command.
Some important SATA commands:
 0x25 read DMA extended (LBA48) (LBA=Logical Block Address)
 0x35 write DMA extended (LBA48)
 0x92 Download microcode (Firmware Update)
 0xb0 SMART

If you are more interested in the SATA commands:
http://www.t13.org/
http://www.t13.org/documents/uploadeddocuments/docs2006/d1699r3f-ata8-acs.pdf

Some of the base addresses I observed in the various sequences were
 (strings debugmex* |grep 0x00000af8 |grep -v LDRB |sort |uniq)
0x00800C00
0x00800C10
0x00800C40
0x00800C50
0x00800C90
0x00800CB0
0x00800DE0
0x00800D60
0x00800DF0
0x00800E00

So everything from 0x00800C00-0x00800E0F is definitely SATA Requests (perhaps the range is even bigger) and the whole 0x00800XXX is potentially SATA PHY related, I would say.
Another thing this tells us is that there are likely only 16 bytes for every SATA request available here.

Later on I discovered that there are actually 33 NCQ (Native-Command-Queuing) Buffers for the requests:
The first buffer starts at 0x00800C00, the second one at 00800C10, … so 0x00800DA0 is actually the 26th request buffer, and the final buffer starts at 0x00800E00 and ends at 00800E1F. And every request buffer is 16 bytes long, and contains the SATA command, the requested address, ...

And even another thing is that the sector address does not actually seem to be read by mex1, so the whole memory management, wear-leveling, block relocation and tabling seems to be done by different cores.

All those addresses are definitely in the BTCM range (TCM=Tightly Coupled Memory, which is a fast SRAM that is tightly coupled to the CPU, most ARM chips have 2 TCM interfaces, named ATCM and BTCM) (ATCM starts at 0x0 and is 0x20000 large, BTCM starts at 0x80000 and is 0x28000 large)
So either the SATA PHY can write into the BTCM itself (or with some DMA help), or the SATA PHY is memory mapped over the BTCM.

Then I looked at what happens after the SATA command byte is read by mex1.
The interesting thing is that the command byte (256 different possibilities) is hashed (multiplied, cut-off, shifted around, multiplied, shifted again),
and then it is looked up in a 16 entry hash table. Unfortunately hash-tables have the unfortunate property of possibly having collisions, and shoveling 256 possibilities into a 16 entry hash table makes that a practical problem,
so they created linked lists (or perhaps even trees) for every hash-table entry where the right entry for a specific command byte is searched.
When the right entry is found, and the function address is not null, the command-handler function is called. If it is null, an error handler is called.
I would suggest to replace this complex construct with a simple 256-entry function pointer table, where every command byte that is not defined is redirecting directly to the error handler.
That way you would get constant O(1) performance, and likely about 1 microsecond less latency for every SSD request, which the current SSD firmware wastes in the hash-table lookup.

Some more thoughts about the trace sequencing:
Even after some days of sampling, I still had sequence ends where I never had a sample that would match to it.
In one case I had 3 different sequences that all ended in the same instruction, but none of the sequences had the following instruction in it.
So it seemed to me that there are barriers that I do not fully understand yet.
The first and easy reaction was to just do more sampling of sequences.
But after thinking about it for a few days, I thought that perhaps the code runs at different speeds (e.g. due to interacting with fast TCM RAM or with slow IO devices elsewhere. Or Cache Hits/Misses)
If I always randomly interrupt the code, it is more likely that I hit slow running code than hitting fast running code, which should result in more sequences covering slow running code and less sequences covering fast running code.
Since I am always capturing 90 instructions, when the first instruction is a slow instruction, I might usually only get the first 90 instructions after a slow instruction and only very seldom instructions beyond the 90 after a slow instruction will be reached.
The way to verify this assumption is to calculate the distribution of all all the first instructions of a sequence among all instructions traced.
If there are significant differences, then the speed differences are likely a problem. (To be done ...)
But how can we mitigate this problem?
My current idea is identify dead-ends, and to specifically put breakpoints at those dead-ends, and to start tracing from where the break-point hits, instead of randomly halting the CPU and tracing from there.
When you succeeded to continue tracing at a dead end, then you will likely want to do the same again with the most likely newly found dead-end again. (To be done...)

So that's effectively a feedback-loop between sampling and sequencing.

The quest for the longest sequence.
So I developed my puzzle tool, which puzzles the sequences together.
I first started with the simple idea of taking the first sequence that was sampled, and at the end of the sequence, it looks through all occurrences of the needed next address.
It makes sure that the overlapping of a potential sequence with the sequence we need it for is good.
If the sequences fit, it attaches them together, and continues again at the end.
When we can find no more sequences, we are done.
One problem I saw quite early was that the algorithm often fell into short loops. What I did then was to randomize the list of potential sequences to be attached to the current one, which solved this short loop problem.
Another change I want to do is to remove the possibility to reuse sequences that were used already, which should remove long loops.

SHA256:

In the bad SSD, I found a SHA256 implementation in memory, in the blocks 0x80000000 and 0x80010000.
At 0x20064 in P23 which gets mapped to 0x80020264 I found the SHA1 hash initialization values ddcc7 on Reddit found the same on a the firmware for the Samsung 840 (not EVO).
https://www.reddit.com/r/ReverseEngineering/comments/2uwhls/samsung_ssd_firmware_deobfuscation_utility/

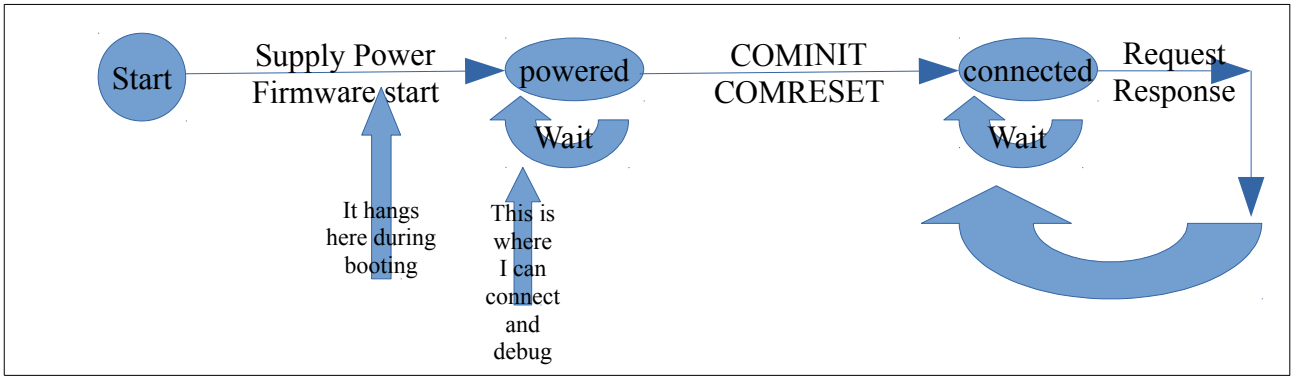
On the HDDGURU forum, I got the hint that SHA256 is used together with an elliptic curves digital signature to protect the firmware updates. Later on, I found something that looked like HMAC-SHA256, but it does not seem to be used for the firmware-update, it seems to be used elsewhere (TCG-OPAL perhaps?).

COMINIT/COMRESET

One question I researched is whether SATA PHYs usually automatically initiate/respond the COMINIT/COMRESET/COMWAKE signals, or whether the CPU has to signal to the SATA PHY that it should do that.
The SATA_PHY project on OpenCores seems to require this signaling from the CPU:
http://opencores.org/websvn,filedetails?repname=sata_phy&path=%2Fsata_phy%2Ftrunk%2Fdoc%2FSATA+PHY+Design+Manual+v1_0.pdf

2 other cores I found
http://www.synopsys.com/IP/InterfaceIP/SATA/Pages/default.aspx
http://ip.cadence.com/ipportfolio/ip-portfolio-overview/interface-ip/serdes-ip/sata-phy
do not seem to require signaling and do the initial communication themselves.
They only need a signal, whether they should behave as host or device, but this can be baked and optimized away, I guess.

In the end it turned out that COMINIT/COMRESET has to be signaled with the CPU, and mex1 cares about it: When I made the change from inserting the SSD into a USB drive bay (for easier handling) to connecting the SSD with separate power and data cables, I tried to see what happens when the SSD is only connected through power but does not get talked to through SATA data, and found that the SSD was able to cope with that:
So we have a state-machine that looks approximately like this:



As soon as the SSD is connected with power, it starts the firmware, when the initialization is done, it starts waiting for a connection with the HOST computer (a small loop that checks the status register 0x200000AC bits for the COMINIT signal), when that signal is given, it continues by configuring the SATA PHY for the connection, and then it starts waiting for Requests from the computer.

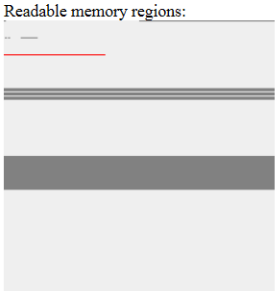
The problem with my SSD is that the Firmware start seems to initialize the SATA PHY properly in the beginning, but it then somehow deviates from the correct program flow and does not get to the "powered wait" loop. If the firmware would get to the "powered wait" loop, it could get the COMINIT signal from the SATA PHY and could continue properly.
So where and why does it deviate? Where are possible branches in the program flow that could lead to where the bad SSD is at the moment?
Later on, I found out that the reason is a problem in the flash channels that is detected by the firmware, and therefore it stops starting and does not enable the SATA interface.

Reverse Engineering Platform

To speed up reverse engineering, I started to develop a Crowd-Sourcing Reverse Engineering Platform:
http://www2.futureware.at/~philipp/ssd/analyse/EXT0CB6Q.dec.html

Memory:

Readable memory regions:



Continuous Memory Dump Sections: detailed Memory Map

ID	Start	Size	Short	Bytes	Writeable	Readonly	Comments
R1	00000000	00020000	2	128 KB	2	0	ATCM, GOOD=BAD
R2	00800000	00800000	128	8 MB	128	0	BTCM, Contains Instructions (Bad-Cycle IP:=0x0081b810) [P11->0x816000]
R3	10010000	00020000	2	128 KB	2	0	DMA, IPC
R4	10040000	00020000	2	128 KB	0	2	DEBUG ROM (CoreSight)
R5	10100000	00100000	16	1 MB	0	16	GOOD=BAD
R6	20000000	00600000	96	6 MB	9	87	PHYs (2028-202F, 2051-205F GOOD=BAD) Data: Read from 0x203B000C
R7	40000000	03000000	768	48 MB	768	0	BTCM of MEX1 (starts with GOOD=BAD) [P21+P22+P31+P32+P41+P42]
R8	44000000	03000000	768	48 MB	768	0	BTCM of MEX2 (starts with GOOD=BAD)
R9	48000000	03000000	768	48 MB	768	0	BTCM of MEX2(starts with GOOD=BAD)
R10	80000000	20000000	8192	512 MB	2	8191	RAM - [P23+P33+P43] - Samsung 512 MB LP-DDR2 SDRAM, but mostly readonly

Firmware Sections:

Partition	Start	Map	Size	Size	good	good2	bad
Phead	0	N/A	512	0,5KB			
P11	512	0x816000	81408	79 KB	47694	47696	41602
P21	81920	0x40000000	131072	128 KB	4	4	4
P22	212992	0x40808000	32768	32 KB	224	224	9
P23	245760	0x80000200	196608	192 KB	133055	133055	121
P31	442368	0x41000000	131072	128 KB	8423	8429	8060
P32	573440	0x41801000	16384	16 KB	4866	4866	4508

CPU Cores:

mex1	R	W	X
mex2	R	W	X
mex3	R	W	X

Trace Logs
SATA Command Table
Memory Map
Disassembling
Repair Manual

The first page is the memory view. In the upper left area you see an image of the 4GB memory space of the ARM cores. Every pixel represents a 64KB block of the memory. The dark-gray areas

are accessible memory, the light-gray areas are unallocated address-space. The red area is the one you have currently selected.
On the right side of it you see a table with all the contiguous memory regions that I have identified so far. When you move your mouse over the table, you will see the specific region highlighted in red on the left image. When you click on the start address a new window/tab will open with a memory browser. If you can identify anything interesting, please leave a comment by clicking on "Comment" and fill the form there.

In the lower left area you can see the firmware that was provided by Samsung on their website, and the sections, and where they are mapped to. When you click on the links in the partition number field, you get to a memory comparison between the bad SSD, and 2 different readouts of the good reference SSD. (If the 2 different readouts of the good SSD are different, that means that it is volatile and differences between the good and the bad SSD are likely not a problem).

On the lower right side you have a table of the 3 ARM cores and their read- write- and execute behavior. When you move your mouse over the cells, you can see the traces of the behavior on the image in the upper left area.
Below that, you have a link to the Debug Traces that I collected: http://www2.futureware.at/cgi-bin/ssd/logs

Memory view

		CPUs			
		mex1	mex2	mex3	mex?
	952085	mex1952085			
	?				good 2 MB good2 good3 30 MB
	analyse	mex1analyse 1 MB			
	bad	mex1-bad-idle			
	boottry	mex1-boottry			
	cutout1192835		mex2cutout1192835		
	devslp242845	mex1devslp242845			
	devslp693816		mex2devslp693816		
	encryption187565	mex1encryption187565 8 MB			
	identifydevice	mex1-identifydevice-good 2 MB			
	idle	mex1-idle 4 MB	mex2-idle 3 MB		
	init	mex1-init-bad 1 MB mex1-init-good-3 mex1-init-good-4 mex1-init-good-5 mex1-init-good-loop mex1_init mex1init-2 mex1init-goodcontinue 1 MB mex1init			
	initnodata	mex1-initnodata			
	loggg	mex1loggg			
	op1	mex1op1			
	queryseq	mex1queryseq 144 MB	mex2queryseq 140 MB		
	ramwait		mex2-ramwait-a	mex3ramwait-3 3 MB mex3ramwait-d mex3ramwait	
	ramwaitcontinue			mex3ramwaitcontinue-2 mex3ramwaitcontinue	
	read	mex1read-2 mex1read-3 mex1read 4 MB	mex2read 30 MB	mex3read 4 MB	
	read1317b7	mex1read1317b7 1 MB			
	read409F75	mex1read409F75			
	readComplete	mex1_readComplete 1020 KB			
	readcomplete	mex1readcomplete 13 MB	mex2readcomplete 1 MB	mex3readcomplete 1 MB	
	readincomplete		mex2_readincomplete 7 MB		
	safe	mex1-safe-good-read mex1-safe-press-r-UART_REGISTER mex1-safe-pressunknown 3 MB mex1-safe-pressunknown2 mex1safe-press-enter			
	safereadblock	mex1safereadblock-12345 mex1safereadblock-2 mex1safereadblock			

I sorted them by CPU core and by the workload that was executed during the trace.
Then I added search functions: When you click on the link on "x" on the addresses, it searches through all traces for all executions of this instruction, so that you can compare the different data that flew through this instruction, and see under which workloads the code is needed.

I have published the software behind this platform on GitHub:
https://github.com/thesourcerer8/CrowdRE
If you have any improvement ideas for the platform, please post issues on GitHub.

Regarding discussing and organisation, started a discussion thread on the Kosagi Forum for Novena Firmware: https://www.kosagi.com/forums/viewtopic.php?id=421

To discuss the hardware and electrical side of the SSD, I started a thread on the HDDGURU forum:
http://forum.hddguru.com/viewtopic.php?f=13&t=34306

I still have a lot of things that I started on that I haven't fully finished visualising yet.

For example, I measured the entropy of the blocks (bad,good,good2) and whether they are identical or different:
http://www2.futureware.at/~philipp/ssd/todo/changes.txt

I collected all read, write and execute accesses of the traces and compiled them in a list:
http://www2.futureware.at/~philipp/ssd/loganalyse.txt
One analysis I want to make is for read/write access to the same address by different cores, which could identify Inter-Core Communication.

Disassembling & Decompiling

After a lot of tracing, I reached some limitations of tracing:
Some code areas are hard to reach, similar to reaching 100% test coverage in testing. Especially if you do not know the sourcecode and it's intentions.
So I started to investigate the disassembling and decompiling again.
As targets I chose Radare2 and IDA Pro.
In my first experiments, I wanted to mount all the sections from the complete 4GB memory space into radare, so that radare can see all data and interpret it correctly. But then I decided to start with the firmware code that is used for initialisation of mex1, which located in the 128KB right at the beginning.
First I took all the comments, and filtered the Safe-mode comments away.
Radare gets `"CC comment" @address` and IDA Pro gets `MakeComm()`
I took all the trace logs, filtered them for the mex1 instructions in normal mode (not Safe mode), and generated `ahb` commands for radare2 and `SetRegEx()` for IDA Pro for every instruction to signal ARM/THUMB mode.
Then I searched for BL/BLX instructions in the trace logs, which are function calls. Even call addresses jump to ARM code, uneven addresses are signaling THUMB code, and have to be decremented for the real target address. For every function call, I generate `SetRegEx()`, `MakeFunction()` and `MakeName()` for IDA Pro and `ahb`, `f` and `af` for Radare.

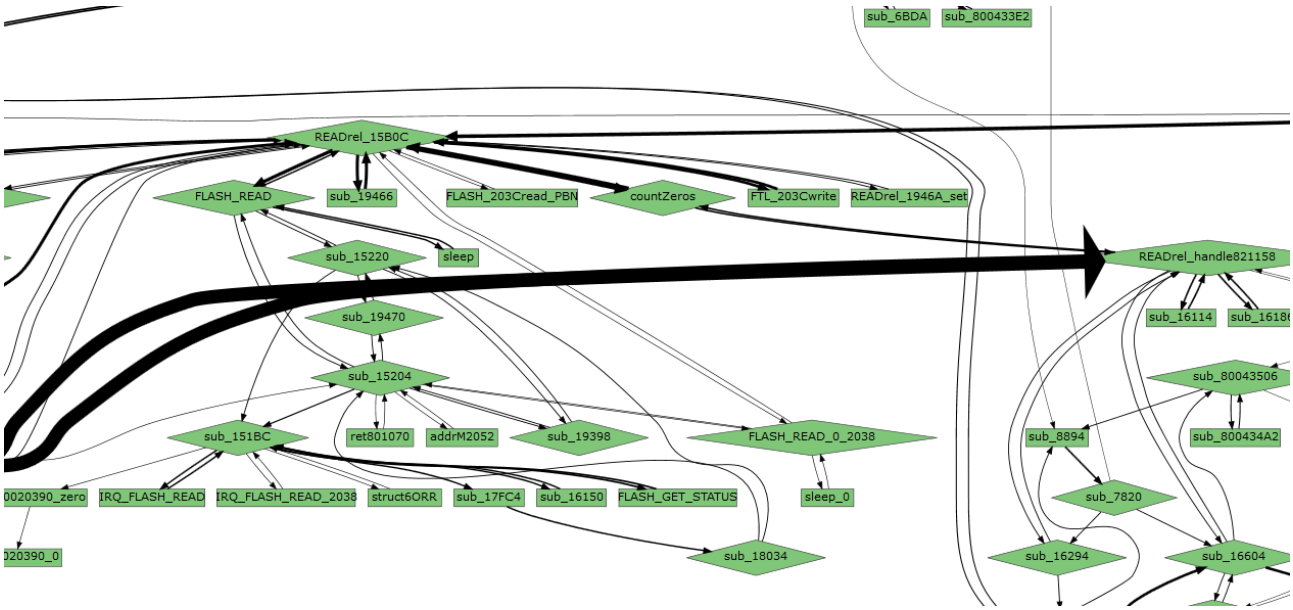
If you want to use radare2 or IDA Pro yourself, take a look here:
http://www2.futureware.at/~philipp/ssd/disasm.html

Later on, I gave tracing another try, and figured out how to configure Linux not to timeout on SATA requests. The following command raises the timeout for 1 million seconds, which was enough to trace the full behavior for the disk "sda" (you have to replace sda with the devicename of the SSD on your system)

```
echo 1000000 >/sys/block/sda/device/timeout
```

Firmware Flow visualsation

I used a commercial tool PQM for business process analysis to visualize the flows that I traced between the various functions, this helped a lot to understand what is actually going on in the firmware. The following screenshot shows read requests that are handled on mex2:



Flash Translation Layer (FTL)

The Flash Translation layer is spread across the 3 cores in the following way:

At first, MEX1 (likely named HCORE or Host-Core by Samsung) receives the SATA requests, as LBA512 addresses (LBA addresses for 512 Byte-Blocks. LBA stands for “Logical-Block-Addressing” which). It divides the LBA512 down to LBA4K and LBA8K addresses. 8K (and larger) requests are split up into 4K requests.

The LBA4K addresses modulo 511 are looked up in HashMap with a Linked list that is empty when starting and is getting filled primarily with Write requests, but sometimes also Read requests end up there.

The LBA4K address modulo 2 (also known as the LSB) is used to decide, whether the request is delegated to MEX2 or MEX3 (they are likely named F-Cores, or Flash-Cores by Samsung).

Then the request is delegated to MEX2 or MEX3, and only the LBA8K address is handed over with the delegation, together with a bitmask which signals which 512-byte-parts of the 8K block are actually to be read.

Now from the MEX2/MEX3 perspective:

MEX2/MEX3 receive the delegated command with the LBA8K address, and divides it by 4 to calculate the LBA32K address. (MEX2 intrinsically knows that is always responsible for the lower 4KB and MEX3 knows that it is responsible for the upper 4KB of the 8K request they get)

The LBA32K modulo 128 is used to lookup something at 0x823168.

Then the LBA32K is further divided and modulo’d by 3760, into LBA32div3760 and LBA32mod3760, which are first used to lookup in a bitfield, and later on in a Hashmap for the FTLaddr (Flash-Translation-Layer-Address), this is the core of the mapping from the LBA32K address to where it is stored in the Flash.

At first I thought that the LBA32K address is mapped to the FTLaddr, which would make it 32K-aligned/sized. But later on, I realized that the FTLaddr is actually 4KB aligned/sized. It turned out that there are 8 tables (4 tables are used by MEX2, the other 4 tables by MEX3) which translate from LBA8K→FTLaddr, and the FTLaddr is different for different 8KB-blocks inside the same 32KB block, so the FLTaddr is actually 4KB sized. And those 8 tables actually correspond to the 8 flash channels.

Now the situation gets a bit difficult, because the flash is organized in several dimensions (Zone, Channel, PBPN, FTLbit), and depending on the order in which you look at those dimensions, the sizes are very different. (The plan to figure out which dimension means what is to try Erasing and seeing which dimensions are affected and which are not affected)

The FTLaddr is then devided by 512 to get a 16-MB size, the rest is remembered as PBN64K as 32K offset into the 16MB blocks. The 16MB could be the Erase-Size.

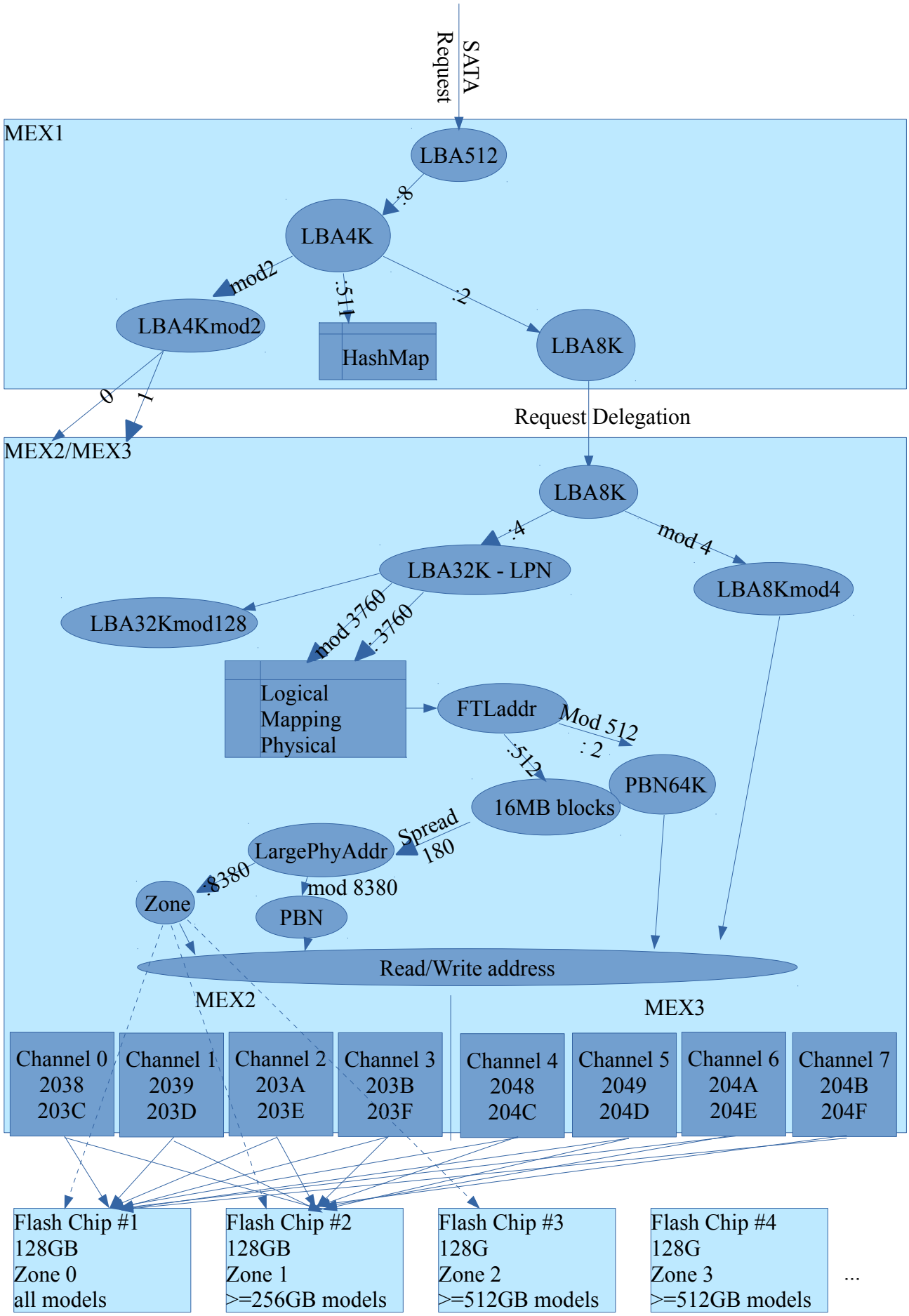
Now this 16MB is spread by180 16MB blocks, which is about 2.8 GB of space for the SA area at the beginning of every physical Flash chip on the PCB. This results in the “LPA, the LargePhysicalAddress”. The LPA is then divided by 8380, which is the Zone (which physical chip on the PCB), and the rest is the PBN (Physical Block Number), which is the physical 16MB Block inside the Physical Chip.

To read/write the physical chip, the PBN64K is added again to the 16MB-sized PBN (Physical Block Number) and together they form the PBPN (PhysicalBlockandPageNumber).

So with the Zone you address the physical chip, with the PBPN you address the 8KB page inside the chip, and with the bitfield you select which of the 512 Blocks in the 8KB block you want to read.

So I was missing the last 3 bits between the 8KB and 64KB. I guessed that they might be given by 8 Flash controllers. It first turned out that all 8 flash controllers have access to the same data, but I later found out that this must been a special case, and those 3 bits are actually into which flash channel to use.

On the other hand, we have 2 chips which are together 250 GB, so one of them must hold a little more than 125 GB. Since the LPA is divided by 8380, we would get 134080 MB / 8380 = 16 MB, so a PBN should be 16MB in size, not 8KB. If we would take 8KB*256*83800*2, we would get only 16 GB capacity per chip. That’s what I meant with the dimensions ...


SATA Request
MEX1
LBA512
:8
LBA4K
mod2
:511
:2
LBA4Kmod2
HashMap
LBA8K
0
1
Request Delegation
MEX2/MEX3
LBA8K
:4
mod 4
LBA32K - LPN
LBA8Kmod4
LBA32Kmod128
mod 3760
:3760
Logical
Mapping
Physical
FTLaddr
Mod 512
:2
:512
PBN64K
16MB blocks
Spread
180
LargePhyAddr
:8380
mod 8380
Zone
PBN
Read/Write address
MEX2
MEX3
Channel 0
2038
203C
Channel 1
2039
203D
Channel 2
203A
203E
Channel 3
203B
203F
Channel 4
2048
204C
Channel 5
2049
204D
Channel 6
204A
204E
Channel 7
204B
204F
Flash Chip #1
128GB
Zone 0
all models
Flash Chip #2
128GB
Zone 1
>=256GB models
Flash Chip #3
128G
Zone 2
>=512GB models
Flash Chip #4
128G
Zone 3
>=512GB models
...

TRIM SUPPORT

I tried trimming the SSD, and discovered that only 4KB (= 8 x 512-Byte blocks) aligned TRIM commands have any effect on the SSD. Sending 8 TRIM commands for 1 block each instead of one large TRIM command for the same 8 512-Byte blocks does not have any effect. Sending a TRIM command for unaligned 16 blocks only affects the aligned 4KB inside the TRIM region, the unaligned start and the rest at the end of the TRIM region is ignored. The reason for this is that the FTL inside EVO840 primarily works on 4KB and 8KB blocks, so the SSD simply does not care about anything smaller. So if you want to send a random TRIM command that definitely deletes at least some data, you would have to make it at least 15 blocks large. Please keep in mind that most partitioning tools created unaligned partitions by default in the past, so even if you align your TRIM commands within a partition, they might be unaligned on the disk.

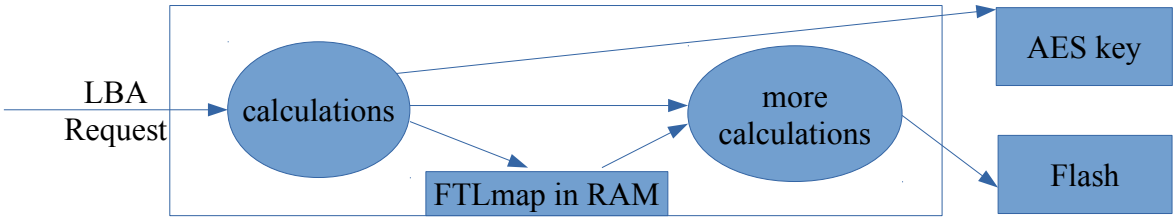
FTLmap (Flash Translation Layer Mapping)

After I had a rough overview of how the FTL system works, I started to develop a tool that should be able to dump and decode any given sector directly from flash, without using the firmware (dumpflash.pl)

At first I tried it with sector 0, since sector0 is most likely already loaded by the operating system, and because it is the easiest to start with. When I had it working, I continued with sector 1, …

I monitored what actually happens when a sector is read from flash, and recorded all the relevant parameters (everything that is SToRed). Then traced the values that were stored to the places where they were calculated by the firmware, and tried to learn how all the values are calculated, the dependencies, …

In a simplified diagram the FTL mapping looks like this:



The FTL takes the LBA number (which 512-Byte block should be read/written to/from the SSD), and calculates the address in the FTLmap. The FTLmap contains the wear-leveling mapping for every 4KB page of the SSD, so every 4KB can be anywhere on the SSD. Every entry in the FTLmap is 64 Bits / 8 Bytes, so the FTLmap for a 250 GB SSD (4883997168 512-byte blocks) takes up 466 MB, which is taking up most of the RAM that the SSD has (512 MB). The address from the FTLmap together with some more bits from the original LBA request is then calculated into the parameters for accessing the right page of Flash. The AES key is directly derived from the LBA, it does not depend on the data in the FTLmap.

Then I tried to calculate all the values for the parameters that are sent to the flash PHY. (And I just retrieved the values from the FTLmap from RAM through JTAG). When I got it working for all the parameters for sector 0, I was able to retrieve the encrypted data of sector 0 and together with the secret keys and the initialisation vector I was able to decode it. I was able to retrieve the first 4 KB of the SSD. Then I tried the other sectors, there I saw that I was also to get sector 8, but for all the other sectors (1 to 7), I got the wrong data, although I sent the correct parameters in. I compared what I sent to what the firmware sent, and they were identical. After some thinking, I had the idea that perhaps I sent the correct parameters to the wrong address, and that turned out to be the problem. Long time ago, I had tried to read from the various channels, and all seemed to return the same data, so I thought that I could use any channel to retrieve the data. I currently guess that I just read coincidentally identical copies. But it actually mattered, which channel I used, and all of the 8 channels deliver different data. And which channel you use is defined by the address where you send the parameters to. So I figured out how the correct channel needs to be calculated (it is determined by the LBA address), and then I was able to read all the sectors. … I thought. After some more tests where I tried to read random sectors from the whole SSD, it turned out that I could only read approximately the first 100 MB of the SSD. So analyzed where the problem was and it turned out that the FTLmap did not had any valid values beyond the first 117,5 MB. So I tried to trace read requests to sectors within the first 117 MB and beyond that (always directly after powercycling the SSD), and it turned out that the SSD did 8 read requests from FLASH before it actually read the sector it should read, which contained the requested data. So the firmware filled up the FTLmap on demand! So it turned out that the visible LBA space is seperated into 2030 blocks where each block contains 3760 times 32KB, which is exactly 117.5MB. And all those blocks are only loaded from the NAND Flash to RAM on demand, on the first access there. Why is it that complicated? Most likely to speed up the startup process of the SSD, and also to reduce the wear of the flash. If it would have to load all the contents of the FTLmap into RAM during the startup process, it might take 2 seconds longer, and not only write access but also read accesses are wearing out the flash cells, which is to be avoided. Another potential reason could be that they need a smaller size for atomic updates to the FTLmap for write processes.

So if you want to dump the whole FTLmap from RAM, you should first send read requests for at least 1 sector every 117.5 MB to the SSD, to get the whole FTLmap loaded into RAM before you can dump it. That’s 2030 requests on the 250 GB SSD. Doing this might also be a good idea to do at bootup time if you need a reliable low latency from the SSD, or if you want to prevent cache timing sidechannel leaks.

Which part of the FTLmap is currently loaded into RAM is stored in a bitarray at `0x849e81a0` for MEX2 and `0x933E81A0` and `9x953741A0` for MEX3 in RAM
A currently open question is where the FTLmap is actually stored on the Flash, and how it is wear-leveled. This needs further investigation.

SAFE MODE UART

The UART interface that is available in SAFE Mode has 3.3Volt FTDI RX/TX pins, and uses 115200 8N1 configuration.
Unfortunately, it is not designed to be used with a Terminal, but it is designed to be used with a special purpose SSD debugging application, which likely only Samsung has.

It has a dozen commands, the only one I analyzed a bit further is the “r” command, which can be used to read data from memory.
I found a tool called “HDD Serial Commander” http://www.hddserialcommander.com/ which should provide a GUI for the commands. (The commands have to be inserted into a SQLite database there)
From the CPU side, you can interface the UART the following way:
UART Base address: 0x20503000
Reading a byte from UART: [0x20503018] : Serial Byte IN
Writing a byte to UART: [0x20503014]: Serial Byte OUT (you must write at least 16 bits to this register, but only 8 bits will be written on the serial line)
After every read/write operation, the firmware code waits approximately 50 CPU cycles in a loop.
You can see the communication patterns here: http://www2.futureware.at/cgi-bin/ssd/logs?log=debugmex1-safe-press-r-UART_REGISTER.log (search for UART on that page)

After some more decompiling, I think I fully understand the UART protocol, there are 3 commands:
“rr” reads 32 bits from memory, it takes the memory address as 8 hexadecimal characters. “rw” writes 32 bits to memory. The rw command only uses the normal STR command, it does not have any flash-erasing super-powers.
“~” initiates a firmware update, followed by 1 Megabyte of the firmware file, followed by the “~” character again.
It seems to me that the UART protocol does not support any other commands, but perhaps I overlooked something.
(Well, with SATA you can upload your own software to RAM and then with the write commands, you could overwrite any of the code areas and execute your own code there)

GPIOs (General Purpose Input Output)

There are a bunch of GPIOs in the controller. Their input value can be read from the register 0x20501004:
The Direction register is 0x20501000. If the bit is set, it outputs the value from 0x20501004 to the GPIO, if the bit is cleared, it reads the value from the GPIO.
The meaning of the value in 0x20501008 is also related, but not clear yet.
20501008 is only written to, the following values: 0, 0x4005004, |=0x4000, |=0x1000 (Bit 12), |=1, |=0x5500, |=0x5000, |=0x50 (which is UART related), 0x30000. It is cleared directly after the direction is set to input and directly before the value is sampled.
Perhaps 0x20501008 is controlling PINMUXing or some other GPIO configuration?

Bit 2 (0x4): UART RX
Bit 3 (0x8): UART TX
Bit 4 (0x10): I2C – SCL (I2C Clock)
Bit 5 (0x20): I2C – SDA (I2C Data)
Bit 6+7 (0xC0): SATA COMINIT
Bit 9 (0x200): Pin11 from the SATA connector
Bit 10 (0x400): ? (this seems to be an input from the SATA core)
Bit 11 (0x800): ? (this seems to be an input from the SATA core)
Bit 12 (0x1000): ? Is set to Output by Stage 2, but never used, perhaps PINMUXed
Bit 13 (0x2000): ? Is set to Output by Stage 2, but never used, perhaps PINMUXed
Bit 15 (0x8000): I2C ??? perhaps providing power to something
Bit 16 (0x10000): ? Seems to be a configuration pin that is read just once
Bit 17 (0x20000): SAFE-Mode Pin (the one next to the JTAG pins)
Bit 18+19 (0xC0000): ? Seem to be configuration pins that are read just once

DMA

The DMA controller (or one of the DMA controllers, it could be 8 or 9(=8+1) or perhaps even more, later on it turned out that there are actually 3 DMA controllers, one for each core) can be found at 0x10010060: To do a DMA transfer, you should wait for bit 4 in 0x10010060 to be cleared, to make sure you do not interrupt a running transfer. Then you write 0 to 0x10010060 to signal that you want to initiate a new transfer. The DMA controller always transfers in blocks of 8 bytes, so you have to divide the size in bytes by 8 and subtract one and write the result to 0x10010064. The source address needs to be written to 0x10010068 and the target address needs to be written to 0x1001006C. When you are done with it, OR 0x10010060 with the value 0x40004003 to initiate the transfer. You can wait for [0x10010060] & 0x10 (bit 4) to be cleared again to wait for the transfer to be finished if you want. Afterwards you should write 0 to 0x10010060 to signal that the DMA controller is free again.
In SAFE mode, this can be used to dump arbitrary memory in a fast way:
You do a DMA transfer to copy the desired RAM to 0x85833000 by initiating the DMA transfer through JTAG with OpenOCD:

```
my $acht=($size>>3)-1;
$openocd->write("halt\n");
$openocd->write("mdb 0x10010060\n");
$openocd->write("mww 0x10010060 0\n");
$openocd->write("mww 0x10010064 $acht\n");
$openocd->write("mww 0x10010068 $source\n");
$openocd->write("mww 0x1001006C 0x85833000\n");
$openocd->write("mww 0x10010060 0x40004003\n");
$openocd->write("mdb 0x10010060\n");
$openocd→write("resume\n");
```

Afterwards you should sleep 1 second to let the DMA transfer complete. We don't want to actively wait for the DMA transfer to finish since this would interrupt the CPU too long, which could cause timeouts on the SATA bus. Just asynchronously starting the DMA transfer and waiting a second for it to be completed works fine.
And afterwards you can read up to 400 MB out of the RAM starting from 0x85833000 by simply reading from the SATA device, like "dd if=/dev/sda of=memdump.dat bs=512 count=10000"
I am still trying to find also the opposite way to write large blocks to arbitrary memory, but I couldn't figure out what the SATA writes are doing exactly. Later on it turned out that writing works in the same way, I just had problems with my test-setup.

Flash

I finally found the Flash Interface, it works the following way:

The base addresses for the flash channels are (MEX2:) 0x203C0000, 0x203D0000, 0x203E0000, 0x203F0000, (MEX3:) 0x204C0000, 0x204D0000, 0x204E0000, 0x204F0000.
The following is written for the base addresses of the first channel:
[203C3C**00**]:= 6(=WRITE) 7(=READ) – Flash Command (the other commands are documented below)
[203C3C**04**]:= 11(=DATA RW) 23(=SA RW) 1(=RW) – this parameter is important to be set correctly, but I don't understand the meaning yet.
[203C3C**08**]:= 89(=DATA) 8A(=DATA) 0x10000007(=WRITE) 0x10000087(=SA) 0x10A5(=READ), 0x10A7(=WRITE) – The FTLbit is added to 0x89 for the parameter for data access, the other bits are not understood yet.

[203C3C**54**]:=PBPN (Phyiscal Block and Page Number), the lower 8 Bits are the Page Number the upper bits are the Block number
[203C3C**58**]:= 0xFF, 0xFF00, 0xFFFF (This is a bitfield, every bit represents a 512Byte block. 0xFF and 0xFF00 are 4KB blocks, 0xFFFF is an 8KB block)
[203C3C**5C**]:= 0x**80**4E8200 (80=DATA), 0x**84**938000(84=SA=Service Area) Memory address of a 8KB block where the data is read/written to, which sub-blocks are actually read/written to is specified by the bitfield at address 58 above
* The following 3 commands are only used for READ-UPDATE-WRITE cycles for the Service Area:
* [203C3C**9C**]:=PBPN+0x100 (reading it from an address and writing it again to the address +0x100 seems to be a wear-level algorithm here)
* [203C3C**A0**]:=0xFFFF (Bitfield again)
* [203C3C**A4**]:=Mem+0x4000
[20380**00**]:=0x3F (This is a combination of 0x30 and 0xF, the **F** is the "destination", which points to 203C**3C**00 in the base addresses above (destination<<10). The 0x40 bit here also has some meaning.
[20380**04**]:=0601, 0701, 2601, 2701 (Zone selects the physical Flash chip on the PCB first chip 0=>0yyy, second chip 1=>2yyy) (Read: 7 Write: 6)
[20380**14**]:=1 (starts the Flash-Transaction, this bit is cleared automatically when the Transaction is finished)
When the Flash Transaction is finished, an IRQ is issued, and the IRQ handler does:
[2038000**C**]:=0x7FFF8000
But this does not seem to be necessary.

Now regarding the status register [2038000**C**]: When everything is ok, it has the value 0xFFFF0000. When you have read a sector it often gets the value 0x7FFF8000, which needs to be acknowledged by writing 0x7FFF8000 again (or more or less whichever value it had in it). When the register has the value 0x7FFF000 after any request, then the Channel seems to be dead. I could not find any way to resurrect a channel got into the 0x7FFF000 state, it seems to ignore all write requests to all the channel registers from then on. Unfortunately, the channels seldom go into the 0x7FFF0000 state with my own reading code. If anyone can find a solution for that, please let me know. My broken SSD actually has channel status values like 0x0FFF0000 and 0xFFFD0000, which I never saw on any of the working SSDs I have. So I guess that the bits marked with the bitmask 0x700F0000 are showing hardware problems.

The Flash interface supports the following commands:

0x03: Unknown command

The function of this command is unknown. After this command, a sleep(600) or sleep(100) is done. Later on the result of this command can be retrieved with command 0x0B.
No Parameters
No Return value

0x05: ERASE command

This function is used in the firmware update routine. It erases a whole block of several pages at once.
Parameters:
[0x04]:=0x03
[0x08]:=0xA4
Return value in [0x0C]

0x06: Write-Page

This command can write 1-16 512-Byte blocks from RAM to FLASH. The firmware usually uses 8 blocks, which make 4096 Bytes, and 16 blocks which make 8192 Byte. The Flash must be erased before it can be written to.

0x07: Read-Page

This command can read 1-16 512-Byte blocks from FLASH to RAM.

0x0A: Integrity check

This command is used in the firmware update code. After the command we are sleeping for 100,000 cycles. It most likely scans the communication between the SSD controller and the NAND flash for bit errors. It is used to check all the channels, whether they work correctly.
Parameters:
[0x08]: 0x20 (Bit)
It seems there is a return value at 0x20300050 + channel, which has the value 0xEC if the check succeeded or something else if the zone is bad. This is likely a hint for the Error Correction.

0x0B: Status

This is some kind of status command, which tells whether an operation has completed or not.
It is relevant after the commands 0x03, 0x10, 0x11 and 0x17
Return value:
[0x0C]: Return value, most interesting bit is bit 15: ([0x0C]>>15)&1

0x0E: Unknown page-oriented command

This command is similar to the read and write commands 0x06 and 0x07. This command can run arbitrarily long, so it is handled with a timeout, and the status is checked every 500 cycles.
[0x04]: 0x01
[0x08]: 0x10A4
[0x58]: Bitfield, which of the 512-byte pages should be affected
[0x54]: PBPN…
[0x5C]: memAddr
[0x174]:= 0 (this it to overwrite any errors that happened earlier)
Return value:
[0x174]: 0xBEAF1234 is a possible return value likely signaling an error

0x10: Set-Value

This command can write a single byte into the Info page. See http://www.onfi.org/~/media/onfi/specs/jesd230b.pdf for more information about the Info page.
Parameters:
[0x18]: 0x01 – the number of bytes to be transferred
[0x1C]: Address in the info page, which byte should be changed
[0x20]: Value to be set
In case 2 bytes are transferred:
[0x24]: Address of the second byte
[0x28]: Value of the second byte
No Return value

0x11: Get-Value

This command can read a single byte from the Info page
Parameters:
[0x18]: 0x01 – the number of bytes to be transferred
[0x1C]: Address in the info page, which byte should be read
Return value:
[0x20]: The byte value is returned here

0x12: Unknown command

The function of this command is unknown.
Parameters:
[0x18]: 0x01 – the number of bytes to be transferred
[0x1C]: Likely an address: Seen value: 0x02/0x01, 0x30, 0x85/0x01, 0x8D/var, 0xA0/0x00, 0xA0/0x01, 0xA9/0x19E7
[0x20]: Unknown input parameter
No Return value

0x14: Unknown command

[0x10]: Parameter

0x15: Unknown command

The function of this command is unknown.
Parameters:
[0x10]: Unknown input parameter
No Return value

0x17: Unknown command

The function of this command is unknown. After this command, a sleep(600) or sleep(100) is done. Later on the result of this command can be retrieved with command 0x0B.
No Parameters
No Return value

0x18: Unknown command

The function of this command is unknown. It is likely doing the opposite of 0x19, since it is called before a read-page command and 0x19 is called directly afterwards.
No Parameters
No Return value

0x19: Unknown command

The function of this command is unknown. It is likely doing the opposite of 0x18
No Parameters
No Return value

0x1C: Unknown command

The function of this command is unknown.
Parameters:
[0x10]: Unknown input parameter

No Return value

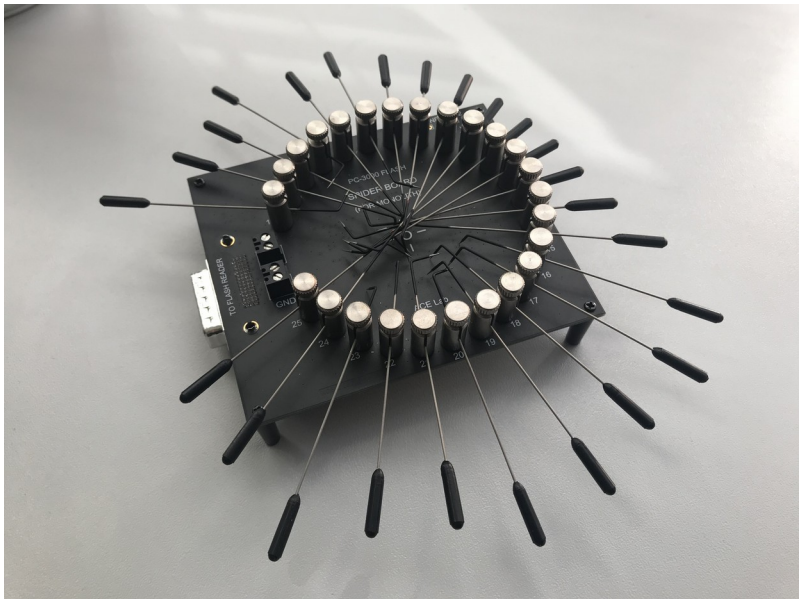
0x1D: Unknown command

The function of this command is unknown.
Parameters:
[0x18]: 0x01 – this likely signals that a single byte is transferred
[0x1C]: 0x30, 0x66 – Different fixed values are sent as parameter
No Return value

Flash Physical Interface

After I had understood most of the Flash interface and the Commands, I got interested in how those commands and parameters actually map to the physical NAND flash chips on the board. I found out that the commands the firmware sends to the Flash controller must get translated to completely different commands that are sent from the Flash controller to the NAND flash chips.
I read through the standardisation documents of the ONFI Flash standards, version 3.2, which define the BGA316 format: http://www.onfi.org/~/media/onfi/specs/onfi_3_2-gold.pdf?la=en
The standard offers several options that could be used or not:
* Pin Reduction
* 16 CE_n and 32 CE_n BGA packaging
For the Pin Reduction feature (to reduce the insanely amount of traces on the PCB), the Eni and Enu pins are specified. If Pin Reduction would be used, those Eni pins would have to be connected from one package to another, but I could not find any such connections when I probed the PCB. So I believe that Pin reduction is not used.
Regarding 16 vs. 32 CE_n pins, the voltage levels of the pins gave the clear picture, that there are only 16 CE_n pins.
So we have a 16 CE_n BGA without Pin Reduction. The disadvantage of this option is that it seems to be completely symmetrical, so you cannot know which side is which side, and whether the chip is upside down, or not. So you have to be extra careful with any chip-off recovery strategies.

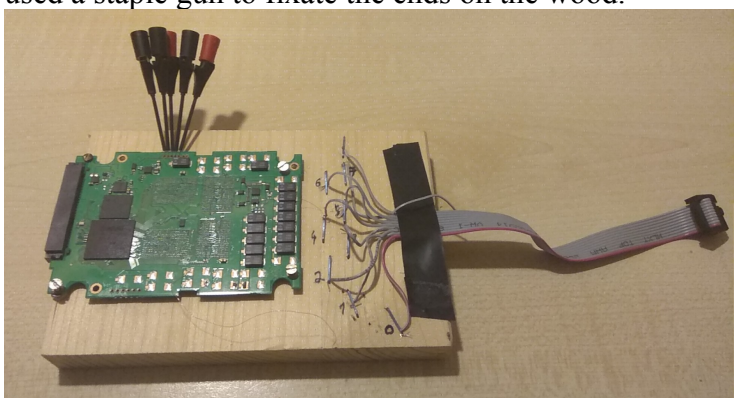
The next big question was, how the physical flash chips are actually addressed from the firmware. Since I suspected that either one of the flash chips or the traces on the PCB were broken on my broken SSD, I wanted to figure out how the specific chips are addressed, so that I could find out, which chip is actually broken.
I had the idea to try a cheap 24 Mt 8CH logic analyzer that I had bought and never used before to connect to the bare balls of the unpopulated Flash footprint on the PCB, try to send commands to all the NAND chips, and measure whether the CE (Chip-Select) lines are activated to send commands there.
But how to connect to those tiny little balls or pads?
The commercial options I found are the Spider from AceLabs (that was introduced on the market after I started this project).
It is designed to connect to the small pins of a monolith SD or micro SD card. Unfortunately it turned out to be too small for a 2.5” SSD. (It looked much larger on the photo)



The real professional solution from my point of view is a Flying Probe machine that is used in electronic factories for testing and programming. Some of the companies that produce such machines are Acculogic, Seica, Spea.

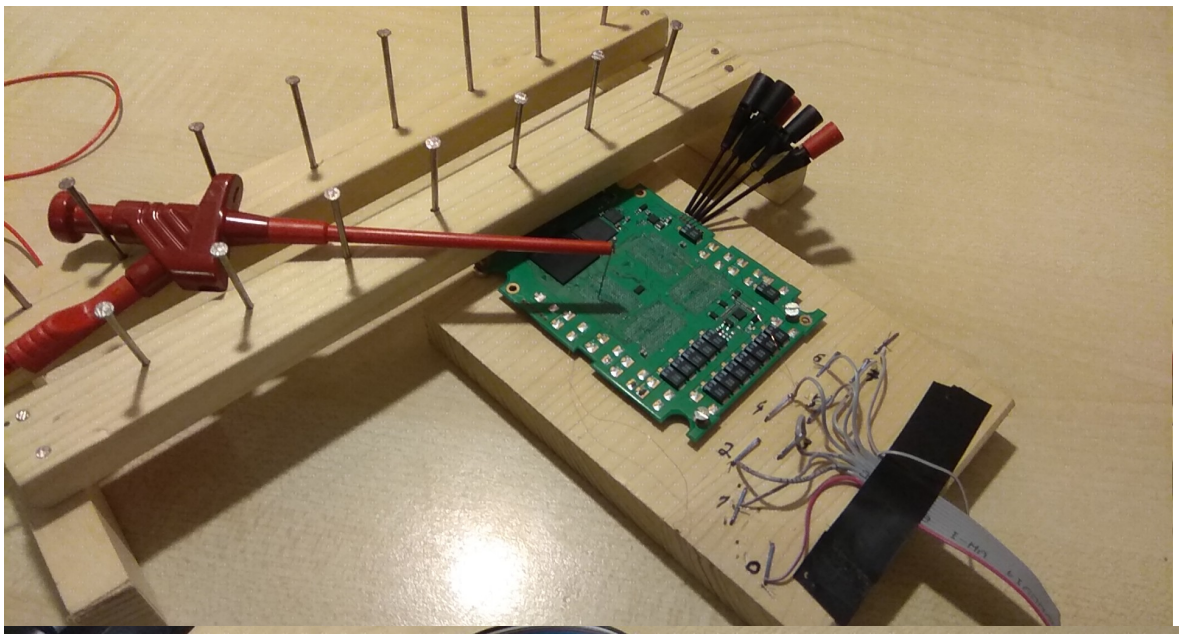
At first I tried with sharp probes. That worked well for just probing a voltage level, but holding the probes for minutes in your hand while controlling the computer and looking at the screen-results at the same time didn't quite work. So I developed several solutions:
To connect to the BGA balls, I first tried soldering enameled copper wire to the BGA balls. What surprised me was that the BGA balls have a very low melting point (less than 200°C I think, I wasn't able to measure it exactly), and the other solder tin on the larger SMD pads had a very high melting point (or perhaps the heat just spread too much, I don't know). I first practiced soldering on the border balls, which are all unconnected.
For the logic analyzer, I have been using PulseView from the sigrok project to interface with it.

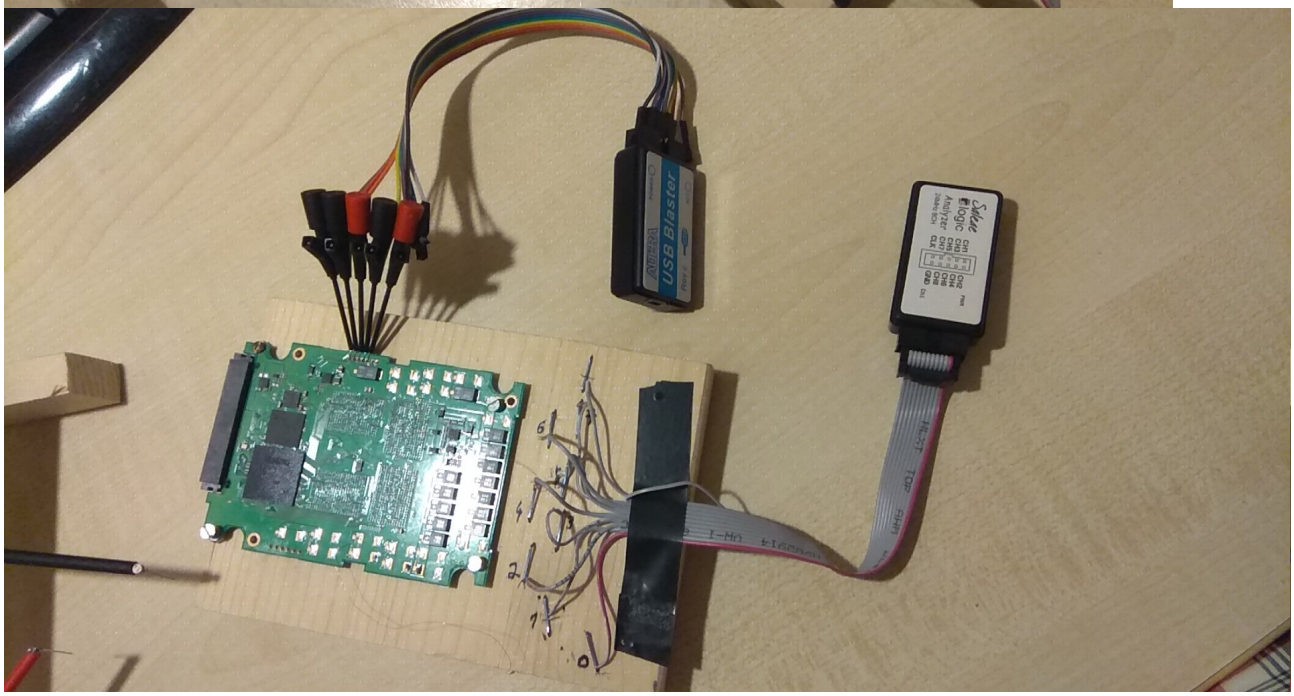
I developed a sigrok plugin that beeps whenever a given signal changes it's value. That way I do not have to look on the screen anymore and can just listen to the signals on the BGA balls, and concentrate on probing.
Since the board of the SSD bent slightly on pressure, I built a wooden fixture and screwed the SSD with spacers on the board. Then I got some Micro-Probes that I could use to interface with the JTAG pins. Then I took a 10-Pin cable, removed the connector and spliced it up on one end, and used a staple gun to fixate the ends on the wood.

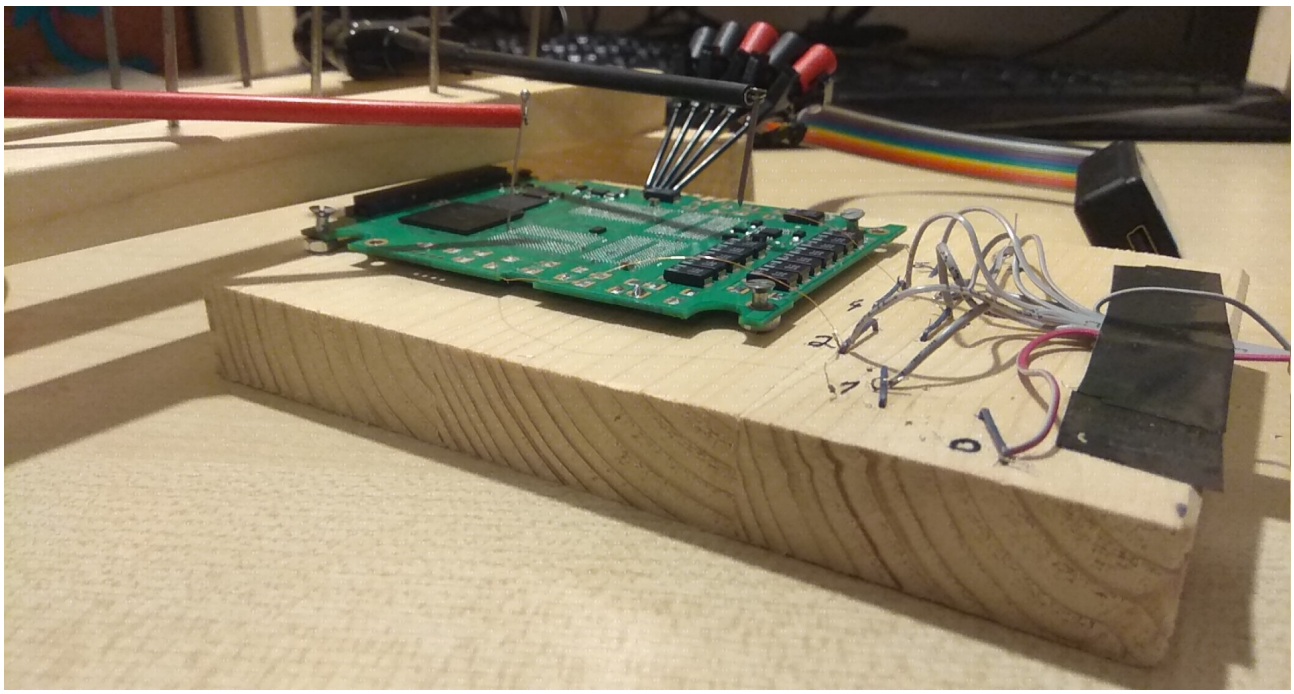


With a crayon I documented which of the 8 logic analyzer channels is where.
Then I found an old wooden game of my children. I took flexible test clip pinch grippers and attached needles in the front, and layed those on the game, so they became my flying probes.

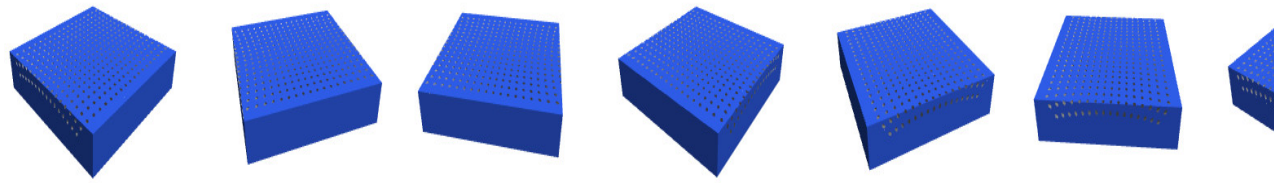
Then I combined both the fixture and the flying probes, and I was able to solder several copper wires permanently and also to directly measure individual BGA balls on demand.



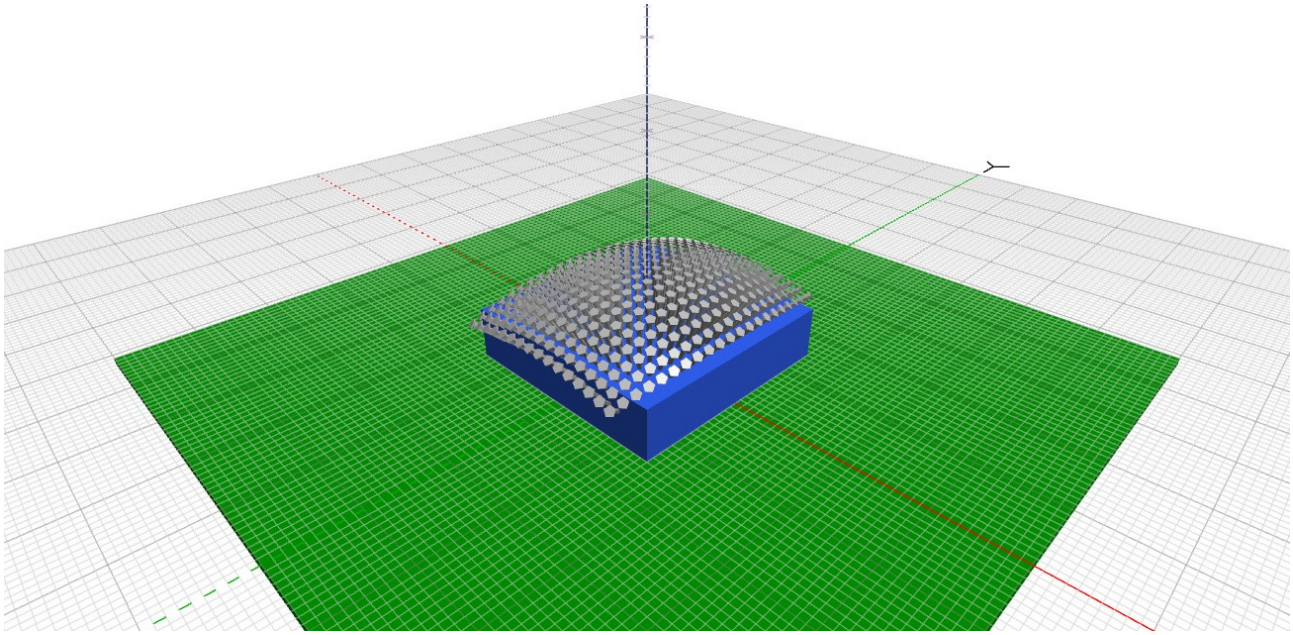

USB Blaster



Another idea I have to interface the BGA balls is special nails holder to make access easier:

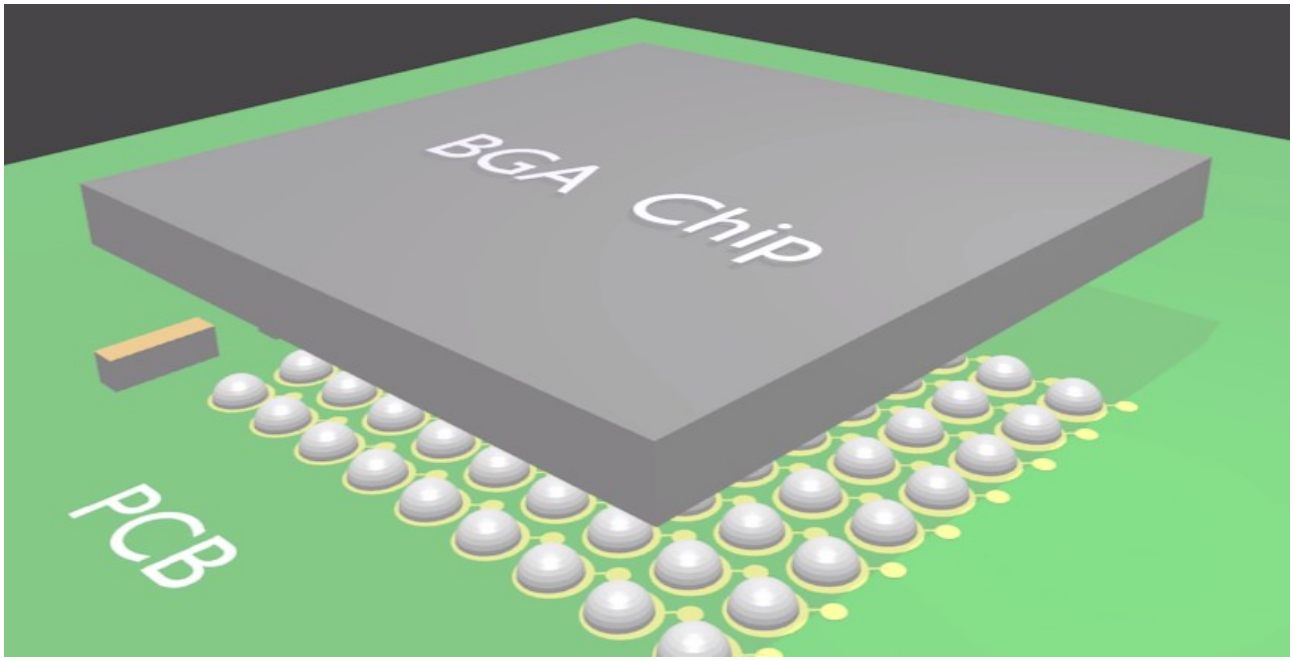


It is designed so that you can put it on a BGA footprint, put nails into it, and then you have a connection:



This way you physically convert the small BGA balls to larger nail heads that are easier to work with.

For the other direction of interfacing the BGA chips in an easier way, my idea is to find a spongy electrically conductive material (or perhaps a flex material?) and put small balls of that material on PCB, so that you can then easily place the BGA chip on the adapter:
I created a video of my idea here:
http://www2.futureware.at/~philipp/ssd/BGAadapter.mp4
http://www2.futureware.at/~philipp/ssd/BGAadapter.avi



Power management

For a long time I had trouble to use the flash interface in a reliable and predictable way. Sometimes it worked, sometimes it didn't, and it took a long time to understand the various preconditions and failure modes. The first problem I had was that I often just got 0x20 when I read some values in the 0x203C0000 address space. So I called this the "0x20 error". Later on, I noticed that I cannot even write something there, I always get back 0x20 when it is in that state. I have setup an automated flash-dump test, which tried to power on the SSD, try a read command, to make sure that all tables are initialized and the flash interface has been used, then to halt all the cores and I tried my own code to dump data from the NAND flash chips. But it reproducably failed with the 0x20 error. So I got the idea to try to monitor a read-request, to set breakpoint directly after the flash request, since I assumed that the flash must be fully enabled at that point of time, and then I tried to read from the 0x203C0000 adddress space, and there the error was gone, and I could read and write and read what I had written in that address space. So I added some code to single-step through the firmware directly after the read request, to try to read from an address in the 0x203C0000 range, and to verify whether it is 0x20 or anything else, after every single-step. And as soon as it would read 0x20, it would stop and tell me at which instruction it stopped. 10 Minutes later I found where the 0x20 error happened, and instruction there was doing [0x20500004] := 0x300 , and it was part of a function that wrote various values and bits to the memory range 0x20500000 – 0x20500030. I searched through the firmware for other write accesses to 0x20500004 and quickly found the counterpart, that wrote [0x20500004] := 0x70F
So it became clear that the firmware turns off the flash interface when it is idle, and turns it on again when new commands arrive. This is likely to save power, and it has the downside that it slows the speed of the SSD, since it takes time to power up again.

Encryption

The controllers contains a Random Number Generator which is used to generate random numbers for the AES keys. The RNG can be found at 0x2050e000 and it can supply up to 32 Bytes at a time! Theoretically we could create a firmware that would provide random numbers on a SATA block device this way.
And the AES controller can be found at 0x20104000.
To encrypt an AES block that is located at 0x20104090 so that the result is written to 0x20104100, you have to write [0x20104000]:=[0x20104000] | 0x10000
To wait for the encryption/decryption to be finished you have to wait on the bit 0x100000 in the same register to clear.
The AES key is written to 0x20104050, the Initialisation-Vector is written to 0x20104070 in RAM.
The keys are stored in a slightly obfuscated way in the service area in the big Flash chips, the MEX processor does not contain any state itself. So if your MEX controller is broken, you can replace it with a good one, or you could put the FLASH chips on a good donor PCB. You only have to match the right firmware version!
The AES modes ECB(1), CBC(2) and XTS(3) are supported by the hardware, but only XTS(3) seems to be used.

Encryption Control

The encryption is controlled by a ranges table, which defines which parts of the SSD is to be encrypted with which key. The AES core seems to support loading up to 8 different AES keys at the same time, while up to 20 different ranges (of which 16 can be used at the same time) can use those 8 keys. A factory new SSD seems to use 3 ranges by default:

Ranges 0x800200F4

Enabled	KeySlotID	LBA-Start	LBA-End	Size:
1	**0x0**	**0x1D24F970**	**0xFFFFFFFF**	Big, likely unused
1	**0x0**	**0**	**0x1D1C5970**	250 GB → data
1	**0xA**	**0x1D1C5970**	**0x1D24F970**	289 MB → Shadow MBR
0	**0xFF**	**0**	**0**	
… this table has 18 or 20 lines				

So the second entry defines which key is used for the actual user-data. If you enable/use OPAL encryption, this entry is split up into smaller regions (e.g. boot sector, and partitions can be encrypted with different keys).
The first entry uses the same AES key (since it's the same KeySlotID), but it seems to be at the end of the physical flash and beyond that.
The third entry is the non-user-visible system area, which uses a different AES key. (The number 0xA is a magic number in the code, it gets protected from the user-partitions).
The KeySlotID 0xFF means undefined.

The KeySlotID field references the actual AES keys:

KeyMaterial 0x825E14

KeySlotId	Enabled	Key1	Key2
0x0	**1**	**32 Bytes AES-256 Key**	**32 Bytes AES-256 Key**
0x1	**0**	**0**	**0**
0x2	**0**	**0**	**0**
0x3	**0**	**0**	**0**
0x4	**0**	**0**	**0**
0x5	**0**	**0**	**0**
0x6	**0**	**0**	**0**
0x7	**0**	**0**	**0**
0x8	**0**	**0**	**0**
0x9	**0**	**0**	**0**
0xA	**1**	**32 Bytes AES-256 Key**	**32 Bytes AES-256 Key**

(Why are there always two 32-Byte long AES-256 keys? Because the XTS mode needs 2 keys)
Then there is a table that references the ranges:

RangeIdArray 0x800200E4

	0	1	2	3	4	5	..	14	15
	00	**01**	**FF**	**FF**	**FF**	**FF**	**...**	**FF**	**02**
PHY Core:	↓	↓	↓	↓	↓	↓	↓	↓	↓
20100380									
20100384									
20100340									

The first (0) references Range 0, which is the big chunk at the end of the SSD.
The second (1) references Range 1, which is the user-data.
Then there are 13 empty ranges (2..14).
The last (15) references Range 2, which is the System-Area.
These 16 positions are then referencing PHY addresses, where most likely the KeySlotId and the LBA-Start and LBA-End are stored into the PHY.

So the encryption core can (when given an LBA address) decide in hardware, which of the 8 AES keys that are loaded are to be used for encryption/decryption.

Additionally there are 2 KeySlotArrays with 8 Byte values each, which are referencing the KeySlots as well:

FirstKeySlotArray 0x800200D4

10	0	FF	FF	FF	FF	FF	FF

SecondKeySlotArray 0x800200CC

?	?	?	?	?	?	?	?

It’s not clear to me yet what those 2 are needed for, but the code is somewhat shoveling around and comparing those two.

So the encryption/decryption of the data happens transparently on the way from SATA<→RAM, and likely only gets turned on/off by the SATA command handlers, and it must get fed with the requested LBA. The actual encryption is done by the hardware then, and the firmware never sees the user data in plaintext.

Since the the encryption is effectively managed by the hardware, the lifecycles of the aboves tables in RAM is very short: In the firmware, those tables are initialized with 0x00, 0xFF, during powerup the tables are filled from structures from the NAND flash, then the necessary values are written to the encryption PHY, and then the tables are overwritten with zeros again, since during normal operation the tables are not needed. If the SSD receives unlock/lock or other keymanagement functions through SATA, those tables are likely recreated again in RAM and cleared again as soon as the command is finished.

Results

I analyzed parts of the initialization code that is executed during the powerup of the SSD.
I found several places where it reads values from memory, where different values were read on the GOOD and the BAD SSD. So I tried to change those values in memory on the BAD SSD to the values that the GOOD SSD had.
Unfortunately I haven't found a way yet to debug the initialization during bootup, so I only had the possibility to boot the SSD up normally, then start debugging, halting all cores to prevent any problems, (both the cores interfering with my tests and me interfering with what the other cores are doing which could cause them to create even more problems in the SSD)
But it turned out that some of those values in memory are read-only, some actually get their values from the initialization, and some do not have the wanted effect. But I haven't analyzed them all yet, there are more memory places to come.
To overcome the problem with this read-only memory and with the problem that I cannot debug the initialization process at the powerup, the idea is to analyze the firmware updating procedure, to find a way to change the first 4 bytes of the firmware to be an endless loop, then I could power it up, start the debugger, halt the core, change the 4 bytes back again to the normal instruction and then continue single-stepping the initialization.

Diagnostics

To analyze a non-starting SSD and figure out what exactly the problem is, I developed an analysis tool, which connects through JTAG to the SSD, reads a few memory values, and provides an automatic diagnosis of the values. I guess that it should work for other firmware versions, and it should be portable to other SSDs from Samsung, and perhaps it could be made useful for other SSDs as well.
You can get it at https://github.com/thesourcerer8/SSDdiag/
Please post the results of running it with your SSD as issues on the GitHub page.

Bugs

At the beginning of the project, I was told the story of a different SSD/HDD vendor who had a bug in the firmware where the Grown-Defect-List had a Buffer-Overflow which wasn't a problem as long as there weren't too many bad sectors, but as soon as there were too many bad sectors, it would overwrite some other vital data areas on the memory, and then the whole SSD/HDD could not start anymore, and millions of drives were affected.

It could be this story: http://www.msfn.org/board/topic/128807-the-solution-for-seagate-720011-hdds/
So I was looking specifically for memory corruption, heap spraying and off-by-one errors in the code. (I would need more memory dumps of different SSDs to be able to spot them in the data)
And surprisingly to me, I found not just one but several off-by-one errors in the code.
The SSD 840 Pro datasheet claims that there should be 32 NCQ slots (https://en.wikipedia.org/wiki/Native_Command_Queuing), but the code first compares whether it is <32, then it acts on the data, and only afterwards it branches depending on the comparison done earlier, so effectively there are 33 NCQ slots. It could be that the 33rd NCQ slot is written over some memory that should belong to something else, I don't know.
Several other similar off-by-one errors were found, which brought me to the idea that the following could be the underlying problem in the Ansi C sourcecode of the firmware:

int i=0;

while(i++<32) mem[i]=0;

Looks right, isn't it? And what about this:

while(++i<32) mem[i]=0;

There is one case where the off-by-one element in a memset() is pointing to unmapped memory addresses, which would resulted in a Data Abort when my single-stepper pre-fetched that memory, so I first thought that there is a bug in my single-stepper, but later it turned out to be an off-by-one in the firmware. Interestingly, even later I found that there is actually a protection mechanism against that kind of off-by-one element in the code that I overlooked earlier, but it does not protect enough in all cases.

All the CPU cores initialize various memory regions during the initialisation routine

MEX2 (the second ARM core) does the following in the initialisation:

for(i=0x80061858; i<0x80060000;++i) *i=0;

Since the end address is before the start address, this should initialize memory, but it doesn't!

The same problem exists in the initialisation code of 2 similar ARM cores, MEX2 and MEX3, each with their own address ranges.

Either the 2 addresses are exchanged, or one of the addresses is wrong.

Therefore some RAM that should be initialized isn't and potentially contains garbage values.

Another bug is that the Controller counts the milliseconds in a 32-Bit register, which will overflow every 49.71 days. (Windows 95 and Windows 98 had the same problem and crashed shortly after the overflow) I do not fully understand the codepaths around it, but it seems that it can crash by sleeping too long after an overflow. Another problem is that the machine-hours in the SMART report might be wrongly calculated with such an overflow, but I haven't checked that code yet.

I have reported the bugs I have found to Samsung, but I never got any qualified answer from them.

Your help is needed

Ok, so much for what I did, now about what you can help:

Please visit the reverse engineering platform, take a look at the traces, and when you understand something, please add your comments there:
http://www2.futureware.at/cgi-bin/ssd/logs
It seems that nobody did that yet, so I am thinking about closing the comment system again.

One helpful thing for reverse engineering has been numerology: If something is done 3 times, it is most likely for the 3 CPU cores, if something is done 4 or 8 times, it is most likely for the 8 Channels to Flash (The 8 channels contain AES (encryption/decryption), DMA (direct memory access), ECC (error correction code) and Flash interfacing in a pipeline, and interestingly, they are grouped in 2 groups of 4 channels each), 16 is likely the number of hashtable entries, but each of the 8 channels also has something 16 times, if something is done 32/33 times it is most likely done for NCQ SATA request buffers. 64 seems to be SATA response buffers. If something is done 80 times, it is related to something that is done for every channel.

The firmware contains strings like "<MESSAGE>%s</MESSAGE>" which are a sign that the firmware was developed in the C programming language. I was not able to identify the C standard library. Perhaps anyone has an idea, which C standard library is used? The compiler seems to be a GCC variant, it is detected by RetDec as "android-ndk-r8 (4.4.3 armv5te, armv7a(arm-linux-androideabi-gcc))"
The firmware does not seem to contain a complete C library, I only found memset, bzero, memcpy, sprintf in it. I found an interesting malloc function, which provides aligned allocation. Unfortunately, it always allocates the space necessary for alignment, and later on it only allocates space for the actual data when there is enough space left. This could be seen as a memory leak, but it unlikely is a real problem. So theoretically, if you called that function once, asking to allocate 1 Byte that is 1GB aligned, it would first allocate the space needed for aligning it to 1 GB, most likely exceeding the buffer by far, afterwards it would fail allocating 1 Byte, but the original alignment would cause any further allocation tries to fail.

If you have any ideas what else I could try out, please let me know.

I will update this document in the future, if I find out any new things.

Error Handling

There is one function that is called from a huge amount of other functions, therefore I think it is a generic error handler. It takes the calling address as the last parameter, it likely logs all the timestamp (in machine hours, since there is no RTC (Real-Time-Clock)), and the location of the

error to Flash. After every call to this error handler I saw a while(1); endless loop to prevent any further execution of the firmware after the error. Interestingly my SSD does not end up in such an error handler, but it seems to wait for inter-core-communication instead.

Easier tasks

* Please analyze the 2 command-parsers, two for the SATA commands (one in SAFE mode, one in normal mode). (The UART debug commands are fully analyzed). Which commands are available, how are they called, which parameters do they have, what do they do? Which commands are dangerous (writing stuff), which are read-only? I found a debugging routine which generates XML about various things, it seems that it is called by the generic error handler, but I couldn't find out yet where it puts its output to. It seems that some of those exception-handling routines are never called by the firmware.

* Search for a way to permanently write to the various memories (perhaps the method can be found in the firmware update command 0x92, perhaps it can be found in the UART debug commands. I think the way to go here is to modify the firmware-update check through JTAG or UART rw commands to ignore any tampering with the firmware, to then load a modified firmware.

* Search for memory patterns that look like a Grown-Defect-List (as if I knew how it looks like)

* Search for the wear-leveling algorithm and any related data structures (I also dont have any good idea how that looks like yet, I am not sure whether I could recognize it if I saw it)

* What does mex3 do with the read accesses to the 0x8yyyyyyy memory region? Are those verifying the results of the ECC verification?

* Improve the sequencing algorithm, perhaps visualize it

* Identify how mex1 wakes up mex2 and mex3

Harder tasks

* Please analyze the firmware updating procedure. I would need a firmware change that has the first 4 bytes changed to an endless loop, so that I can reliably debug the initialisation of the firmware. My current guess is that there is a 16 or 32 bit checksum at the end of the firmware header which protects the whole firmware. This guess was wrong, SHA-256 is used instead. Although I dont' believe in the security of elliptic curves, I doubt that there are security issues in the firmware-update process

* Identify the various PHYs that are used in the controller. Find datasheets for potentially used PHYs and verify the access behaviour with the register documentation in the datasheets.
Ok, I did that, the result is a huge memory map documentation here:
http://www2.futureware.at/~philipp/ssd/MemoryMap.pdf

* Find a way to debug the initialisation of the SSD (I am wondering how Samsung is doing that without a SRST line)

ARM CPU Register Documentation

ARM cores have a lot of CPU registers which can be read with the MCR instruction.

At first I tried to read them manually, but I soon gave up and automated it:

I copied the ARM documentation, and wrote a script that parses all the example read commands from the official ARM documentation website, executed those read commands through OpenOCD against all 3 cores, and then writing out the whole documentation with annotations of the actual values, down to the bitfields. In case the 3 cores would have provided different values, the script would have explained that, but it unexpectedly turned out that all 3 cores had the same values. (There is one register that should contain the Core-ID which I would have expected to be different (perhaps that's a design issue?), and I expected references to the CoreSight modules, but I found none)

http://www2.futureware.at/~philipp/ssd/cortexr4regsgood.html

Table 4.3. MIDR Register bit assignments
Bits
Name Function
Value: **65(0x41)** [31:24] Implementer
Indicates implementer:
0x41 = ARM Limited.
Value: **1** [23:20] Variant
Identifies the major revision of the processor. This is the major revision number n in the rn part of the rnpn description of the product revision status.
Value: **7** [19:16] Architecture
Indicates the architecture version:
0xF = see feature registers.
Value: **1044(0x414)** [15:4] Primary part number
Indicates processor part number:
0xC14 = Cortex-R4.
Value: **4** [3:0] Revision
Identifies the minor revision of the processor. This is the minor revision number n in the pn part of the rnpn description of the product revision status.
Note
If an MRC instruction is executed with CRn = c0, Opcode_1 = 0, CRm = c0, and an Opcode_2 value corresponding to an unimplemented or reserved I
value of the MIDR.
To access the MIDR Register, read CP15 with:
MRC p15, 0, , c0, c0, 0 ; Read MIDR

Table 4.4. CTR Register bit assignments
Bits
Name
Function
Value: **0** [31:28]
- Always b1000.
Value: **0** [27:24] CWG
Cache Write-back Granule:
0x0 = no information provided. See maximum cache line size in c0, Current Cache Size Identification Register.

So in the end you can read your individualized ARM documentation that documents your specific chip, and you don't have to switch between the documentation and calculate the bit fields by hand anymore. I guess that this tool might be valuable for a lot of ARM developers...

http://www2.futureware.at/~philipp/ssd/cortexr4regs.pl
The only problem I found out later is that the MPU registers are actually a whole array, and you need to initialize one register with the region index first before reading/writing the other registers, so my tool only provides the values for the currently set index.

Appendix

The following is a list of references and notes, and documentation snippets I collected

References

Some more links that might be interesting:
http://blog.acelaboratory.com/pc-3000-ssd-samsung-family.html
http://www.acelaboratory.com/catalog/vosstanovlenie-dannih-sdd-flash.php
https://www.reddit.com/r/ReverseEngineering/comments/2uwhls/samsung_ssd_firmware_deobfuscation_utility/
https://github.com/radare/radare2/issues/3190
https://github.com/arduino/OpenOCD/blob/master/tcl/interface/sysfsgpio-raspberrypi.cfg
AHCI - OSDev Wiki
SSD
fmadio | Recover Bricked SSD with JTAG
False Positive Strings · Issue #3190 · radare/radare2
Novena headers to linux GPIO mappings - Studio Kousagi Wiki
Definitive GPIO guide - Studio Kousagi Wiki
cheatsheets/radare2.md at master · pwntester/cheatsheets
ARM Information Center
BeagleBoardOpenOCD - eLinux.org
openOCD/coresight-trace.txt at master · kevinmehall/openOCD
raspberrypi/armjtag/rpi2 at master · dwelch67/raspberrypi
Telecomm.& Memory Ic Supplier,Original Condition! Offer Sample K90kgy8s7m-cck0 - Buy K90kgy8s7m-cck0,Samsung Memory Ic,Computer Ic Product on Alibaba.com
OpenOCD - Open On-Chip Debugger / Mailing Lists
boot - What is the booting process for ARM? - Stack Overflow
ARM CoreSight Architecture Specification v2.0 - IHI0029D_coresight_architecture_spec_v2_0.pdf
google/binnavi: BinNavi is a binary analysis IDE that allows to inspect, navigate, edit and annotate control flow graphs and call graphs of disassembled code.
depcb/screenshot.png at master · unixdj/depcb
pyOCD/cmsis_dap_core.py at master · mbedmicro/pyOCD
mongodb-labs/disasm: Interactive Disassembler GUI
BinaryAnalysisPlatform/qira: QEMU Interactive Runtime Analyser
Hard Disk Firmware Hacking (Part 1) | MalwareTech
SSD firmware hacking.
HDD GURU FORUMS • View topic - How to find TRST & SRST for JTAG?
ARM Processors: How to debug: CoreSight basics ... | ARM Connected Community
The HDD Oracle. • View topic - Datasheets for SSD and Flash Drive ICs
Microsoft Word - dg_sata_ip_appnote1_en.doc - dg_sata_ip_appnote1_en.pdf
IDLE - serialata10a.pdf
Welcome to the Python Mapper documentation! — Python Mapper documentation
SATA Storage Technology - SATA Storage Technology.pdf
Free ip cores - ASICS.ws - Your Partner for IP Cores, ASIC/FPGA Design, Synthesis and 14.pdf
HDD GURU FORUMS • View topic - STUart - Seagate sectors over UART driver
HDD Serial Commander
The HDD Oracle. • View topic - HDDs / PCBs Suppliers list :

radare/radeco: radare decompiler tool based on radeco-lib
Screenshots - SD Computing Service

OPENOCD Commands

drscan # not necessary
dump_image <file> <address> <size>
halt
reset halt
soft_reset_halt
resume
step
poll
reg
armv4_5 core_state
armv4_5 reg
cortex_a8 cache_info
dap apid
dap apsel
dap apsel 0
dap apsel 1
dap apsel 2
dap apsel 3
dap baseaddr
dap info
dap info 0
dap info 1
dap info 2
dap info 3
arm disassemble 0x80e88158 10
bp 0x80e88160 4 hw
mdw 0 (memory read word)
#mww 0x2020001C 0x10000 (memory write word
nand probe 0
nand list
nand dump 0 voltcraft_dso-3062c_nand_dump_normal.dd 0 0x4000000

set _TARGETNAME $_CHIPNAME.cpu.0
target create $_TARGETNAME cortex_a -chain-position $_CHIPNAME.dap -coreid 0 -dbgbase 0x80010000
set _TARGETNAME $_CHIPNAME.cpu.1
target create $_TARGETNAME cortex_a -chain-position $_CHIPNAME.dap -coreid 1 -dbgbase 0x80012000
set _TARGETNAME $_CHIPNAME.cpu.2
target create $_TARGETNAME cortex_a -chain-position $_CHIPNAME.dap -coreid 2 -dbgbase 0x80014000

GDB Commands:

monitor scan_chain
info registers
p/x $pc
x/i $pc
x

ARM ISA:

This is a list of the ARM instructions and and a short description what they do. Those instructions can be enhanced with conditional argument, for example: B (Branch)+EQ (Equal)=BEQ (Branch if Equal), or STR (Store)+EQ (Equal)=STREQ (Store if Equal)
ADC Add with carry Rd: = Rn + Op2 + Carry
ADD Add Rd: = Rn + Op2
AND AND Rd: = Rn AND Op2
B Branch R15: = address
BIC Bit clear Rd: = Rn AND NOT Op2
BL Branch with link R14: = R15, R15: = address
BX Branch and exchange R15: = Rn,
T bit: = Rn[0]
CDP Coprocessor data processing (coprocessor-specific)
CMN Compare negative CPSR flags: = Rn + Op2
CMP Compare CPSR flags: = Rn - Op2
EOR Exclusive OR Rd: = (Rn AND NOT Op2)
OR (op2 AND NOT Rn)
LDC Load coprocessor from memory Coprocessor load
LDM Load multiple registers Stack manipulation (Pop)
LDR Load register from memory Rd: = (address)
MCR Move CPU register to coprocessor register cRn: = rRn {<op>cRm}
MLA Multiply accumulate Rd: = (Rm * Rs) + Rn
MOV Move register or constant Rd: = Op2
MRC Move from coprocessor register to CPU register Rn: = cRn {<op>cRm}
MRS Move PSR status/flags to register Rn: = PSR
MSR Move register to PSR status/flags PSR: = Rm
MUL Multiply
MVN Move negative register Rd: = 0xFFFFFFFF EOR Op2
ORR OR Rd: = Rn OR Op2
RSB Reverse subtract Rd: = Op2 - Rn
RSC Reverse subtract with carry Rd: = Op2 - Rn-1 + Carry
SBC Subtract with carry Rd: = Rn - Op2-1 + Carry
STC Store coprocessor register to memory Address: = CRn
STM Store multiple Stack manipulation (push)
STR Store register to memory <address>: = Rd
SUB Subtract Rd: = Rn - Op2
SWI Software Interrupt OS call
SWP Swap register with memory Rd: = [Rn], [Rn] := Rm
TEQ Test bit-wise equality CPSR flags: = Rn EOR Op2
TST Test bits CPSR flags: = Rn AND Op2

THUMB ISA:

ADC Add with carry V – V
ADD Add V V V (1)
AND AND V – V
ASR Arithmetic shift right V – V
B Unconditional branch V – –
Bxx Conditional branch V – –
BIC Bit clear V – V
BL Branch and link V – –
BX Branch and exchange V V –
CMN Compare negative V – V
CMP Compare V V V
EOR EOR V – V
LDMIA Load multiple V – –
LDR Load word V – –
LDRB Load byte V – –
LDRH Load half-word V – –
LSL Logical shift left V – V
LDSB Load sign-extended byte V – –
LDSH Load sign-extended half-word V – –
LSR Logical shift right V – V
MOV Move register V V V (2)
MUL Multiply V – V
MVN Move negative register V – V
NEG Negate V – V
ORR OR V – V
POP Pop registers V – –
PUSH Push registers V – –
POR Rotate right V – V
SBC Subtract with carry V – V
STMIA Store multiple V – –
STR Store word V – –
STRB Store byte V – –
STRH Store half-word V – –
SWI Software interrupt – – –
SUB Subtract V – V
TST Test bits V – V

TODOs

Implement [PC, #123] recognition for singlestepper

Maximum JTAG Scan Interface Timing: 12 MHz

Serial Number Identification:
Good=S1DFNEAD802015A
Bad =S1DBNSBFA80438H

The address 0x40825088 is only reachable for mex1, not for mex2 or mex3. Why?

ARM Semihosting!!!

arm7_9 dbgrq [enable|disable]

goodlogs/debugmex1init.log:0x0000007c->0x00000080 ARM Supervisor 0xf57ff04f UNDEFINED INSTRUCTION svcmi 0xf07ff5
goodlogs/debugmex1init.log:0x00000084->0x00000088 ARM Supervisor 0xf57ff06f UNDEFINED INSTRUCTION svcvs 0xf07ff5
goodlogs/debugmexall.log: 0x00000078 0xf57ff04f UNDEFINED INSTRUCTION svcmi 0xf07ff5
goodlogs/debugmexall.log: 0x00000080 0xf57ff06f UNDEFINED INSTRUCTION svcvs 0xf07ff5
goodlogs/debugmex1init.log:0x000001d0->0x000001d4 ARM Supervisor 0xf57ff04f UNDEFINED INSTRUCTION svcmi 0xf07ff5
goodlogs/debugmex1init.log:0x000001d8->0x000001dc ARM Supervisor 0xf57ff06f UNDEFINED INSTRUCTION svcvs 0xf07ff5
0x0001b69e->0x0000e918 Thumb Supervisor 0xf7f3e93c UNDEFINED OPCODE stcllo p3, c15, [sb], 0x3dc
goodlogs/debugmex3.log: 0x00018648->0x000192b4 Thumb Supervisor 0xf000ee34 UNDEFINED OPCODE lsls r0, r6, 3 ; adds r4, 0xee
rasm2 -aarm -b16 -d 0xf000ee34
rasm2 -aarm -b32 -d 0xf57ff04f