Reverse Engineering Flash Memory for Fun and Benefit

Jeong Wook (Matt) Oh

oh@hp.com

oh.jeongwook@gmail.com

HP

NAND Flash technology

Flash Technology was invented circa 1980 by a Japanese inventor, Dr. Fujio Masuoka, while he was working for Toshiba. (1) Intel was the first company to produce the chips en masse. (1) In the 1990s, the technology was adapted from the industry and is now used everywhere. There are two different types of technology in Flash memory. First, NOR-based Flash is typically used as a replacement for old ROM technology. It has a long erase and write time, but it has a random read access capability for any memory location. In contrast, NAND-based Flash has a shorter erase and write time, but has other limitations. One limitation with NAND-based Flash is that it needs page-level access to the data. When reading or writing, NAND can't write by byte level, and the page size can vary from a few hundred bytes to a few thousand. (2)

In this paper, I am going to present a methodology for reverse engineering NAND Flash memory. I am most interested in NAND Flash technology when it is used for storage on embedded systems. Even if you can't perform random data access efficiently with NAND Flash, embedded devices can load up a whole NAND image to a DRAM and start up the operating system on the memory using an MTD (Memory Technology Device). I've found reverse engineering NAND Flash to be very beneficial when I was experimenting with many embedded devices. There are many different models of NAND Flash out there, and I'm using TSOP (Thin Small Outline Package) 48 type NAND Flash memory for my experiment here. This type of chip is very commonly used in many embedded devices on the market.

NAND Flash specification

The ONFI (Open NAND Flash Interface) is a joint working group of the companies involved with NAND Flash technology. It has published a series of industry standards, with specifications that are shared openly and revised over time to include new features. These resources are extremely useful for coming to grips with this technology. However, each chipset has its own specification, so whenever you work on your project, try to find the appropriate specification for your chipset. Mostly, the datasheets don't vary much for each of the NAND Flash chipsets, but I advise referring to the most accurate information you can find for your chipset.

Direct interaction over JTAG method

The Joint Test Action Group (JTAG) technique is the most common approach when reverse engineering modern embedded systems. While most vendors leave the JTAG interface for debugging and support, there is a growing trend for obfuscating or removing it for security purposes. If the target device is using NAND Flash memory for data storage, you can use a direct interaction method over JTAG.

De-soldering

The first step when interacting with NAND Flash memory is de-soldering the chip. You might use an SMT (Surface Mount Technology) re-work station for this process. (Figure 1)

Figure 1 SMT re-work station

The de-soldering process is very straightforward. The de-soldering station provides a hot air blower. Using the hot air, the solder alloy usually melts around 180 to 190 °C (360 to 370 °F) although I recommend setting the temperature slightly higher than that. Before applying high heat to the chip, you should put insulating tape around the target area. (Figure 2) This is for a couple of reasons: it protects other chips and stops the PCB (Printed Circuit Board) from burning; and it also prevents other small parts from being de-soldered accidentally. Direct the hot air over the pin areas evenly. At some point the chip will loosen and you can use tweezers or a similar tool to remove the chip from the board. You should be very careful not to burn yourself during this process.

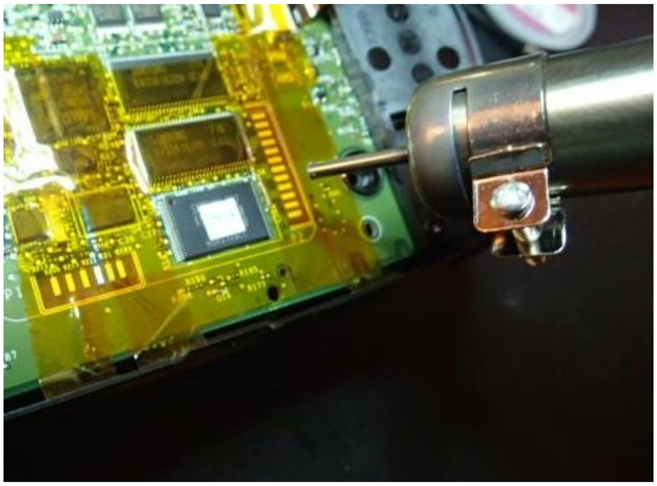


Figure 2 De-soldering in progress

Figure 3 Removed chip

NAND reader writer

Now you have a bare NAND Flash chip at hand. The next step is reading the bare metal image from the chip. There have been a lot of different approaches tried over time by the hobbyist community: Some use special Flash reader chipsets that can provide low-level access. However, the most reliable way I found to do this was bit-banging using the FTDI FT2232H chip set. This method was originally suggested by [Sprites Mod](). Bit-banging is a technique that allows you to directly interact with chips through software. FT2232H is a versatile chip that provides various ways to interact with chips through a USB interface.

FTDI FT2232H

FTDI FT2232H is a chip for USB communication. It provides USB 2.0 Hi-Speed (480Mb/s) to UART/FIFO IC. (3)To make my life easier, I just purchased an FTDI FT2232H breakout board and put female pin headers upon each of the port extensions. (Figure 4) The FTDI chip sets are pretty popular with hobbyists because of their versatility, so it would not be difficult to find a similar breakout board on the market.

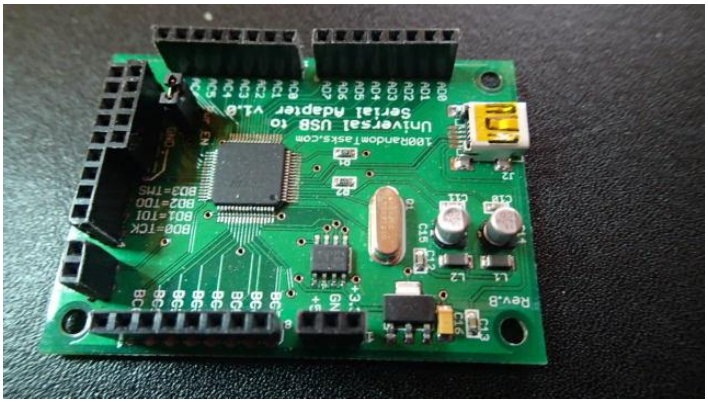


Figure 4 FTDI FT2232H Breakout board

FTDI FT2232H supports multiple modes. The 'MCU Host Bus Emulation Mode' is appropriate for our purposes in this case. In this mode, the FTDI chip emulates an 8048/8051 MCU host bus. By sending the commands shown in Table 1 and retrieving the results, the software can read or write bits through I/O lines. More details are available in a note published by FTDI.

Commands	Operation	Address
0x90	Read	8bit address
0x91	Read	16bit address
0x92	Write	8bit address
0x93	Write	16bit address
0x82	Set	High byte (BDBUS6, 7)
0x83	Read	High byte (BDBUS6, 7)

Table 1 FT2232H Commands

Connecting FT2232H with NAND Flash pins

Figure 5 shows the typical NAND Flash memory, its pin numbers and names.

Figure 5 Important NAND Flash memory pins and names

Figure 6 shows the connection between the FT2232H chip and NAND Flash memory. The connections are mostly based on information from [Sprites Mod](), but there is a slight modification between BDBUS6 and the CE (9) connection.

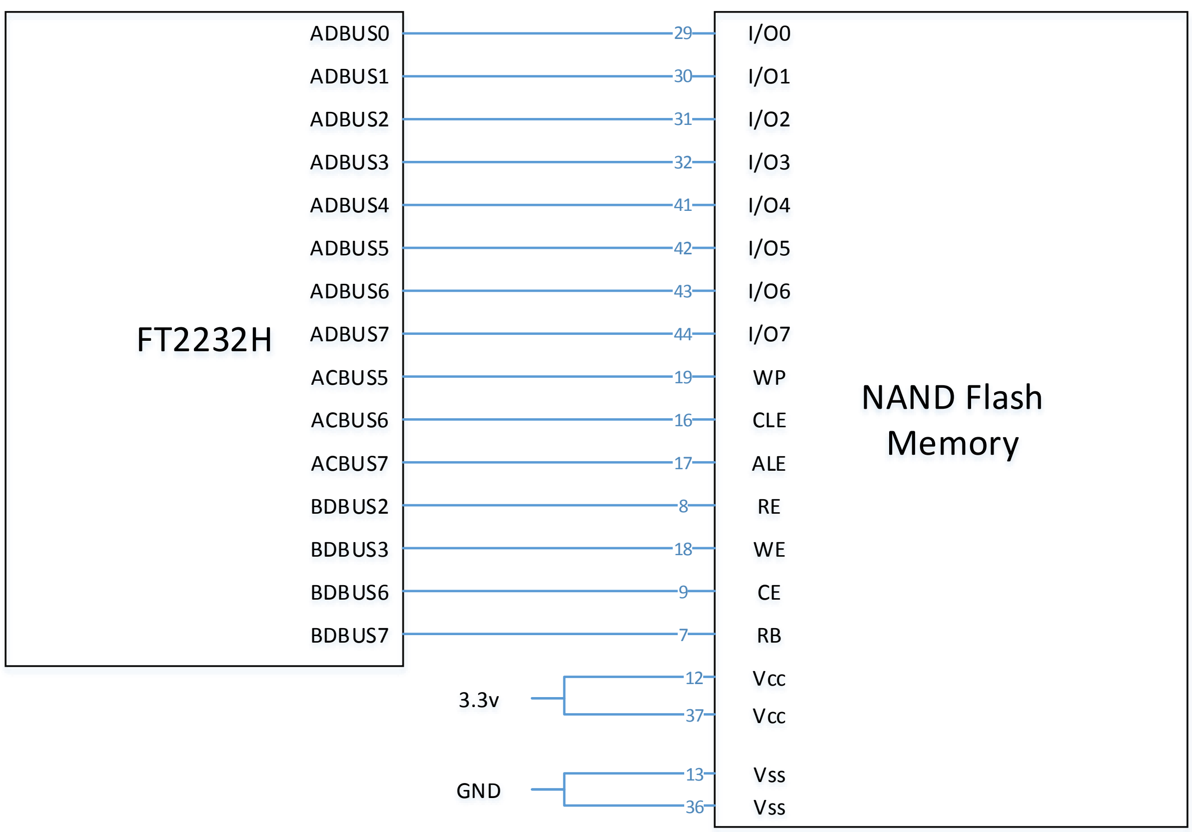


Figure 6 Connection between FT2232H and NAND Flash Memory

Table 2 shows you how to connect FT2232H pins with NAND Flash data lines. The ADBUS0 to ADBUS7 pins are used for data transfer and are connected to the I/O0 to I/O7 pins of the NAND Flash memory chip. The functions of FT2232H's pins are well explained in the datasheet. They are used for 8bit data transfer.

FT2232H	Use	NAND Flash	Pin number	Description
ADBUS0	Bit0	I/O0	29	**DATA INPUT/OUTPUT** Input command, address and data. Output data during read operations.
ADBUS1	Bit1	I/O1	30	
ADBUS2	Bit2	I/O2	31	
ADBUS3	Bit3	I/O3	32	
ADBUS4	Bit4	I/O4	41	
ADBUS5	Bit5	I/O5	42	
ADBUS6	Bit6	I/O6	43	
ADBUS7	Bit7	I/O7	44	

Table 2 FT2232H and NAND Flash Connections – Data Lines

Table 3 shows the connections for data type bit pins. CLE and ALE are used for command latch and address latch enabling purposes, which means that when new commands or addresses are transferred these lines should go high [1]. In this way, NAND Flash can differentiate between commands, addresses and data. WP should go high when write operations are under way. CLE, ALE and WP are on ACBUS and this bus is the 8 high bits when a 16bit operation is performed from the FTDI FT2232H chip. By setting these bits on and off, the software side can control what kind of data or operations are sent to the Flash memory.

FT2232H	Use	NAND Flash	Pin number	Description
ACBUS5	Bit13	WP	19	**WRITE PROTECT** Write operations fail when this is not high.
ACBUS6	Bit14	CLE	16	**COMMAND LATCH ENABLE** When this is high, commands are latched into the command register through the I/O ports.
ACBUS7	Bit15	ALE	17	**ADDRESS LATCH ENABLE** When this is high, addresses are latched into the address registers.

Table 3 FT2232H and NAND Flash Connections – Data Types Bits

The RE and WE pins are used for signaling readiness for the FT2232H chip's data read or write operations. When the FT2232H chip is ready to read data, it sends a falling signal on the BDBUS2 (RD#) pin and lets the other party know to send new data. When BDBUS3 (WR#) output is rising, it means new data is available from the FT2232H chip and it lets the NAND Flash chip fetch it. The BDBUS6 (I/O0) and BDBUS7 (I/O1) pins can be set and read using SET_BITS_HIGH (0x81), GET_BITS_HIGH (0x83) FT2232H commands. When RB is low, it means the Flash memory chip is busy processing data. CE bits are usually set to low, but when sequential row read operation is used, the pin needs to be set high after reading each block data.

FT2232H	Use	NAND Flash	Pin number	Description
BDBUS6	I/O0	CE	9	**CHIP ENABLE** Low state means the chip is enabled.
BDBUS7	I/O1	RB	7	**READY/BUSY OUTPUT** This pin indicates the status of the device operation. Low=busy, High=ready.
BDBUS2	Serial Data In (RD#)	RE	8	**READ ENABLE** Serial data-out control. Enable reading data from the device.
BDBUS3	Serial Signal Out (WR#)	WE	18	**WRITE ENABLE** Commands, addresses and data are latched on the rising edge of the WE pulse.

Table 4 FT2232H and NAND Flash connections –synchronization & control

Figure 7 shows a good example of a read operation. CLE and ALE go high which means the controller is sending commands and addresses. The RE changes phases when page data is read from the NAND Flash chip. The R/B line goes low during the busy state and back up to high when the NAND chip is ready.

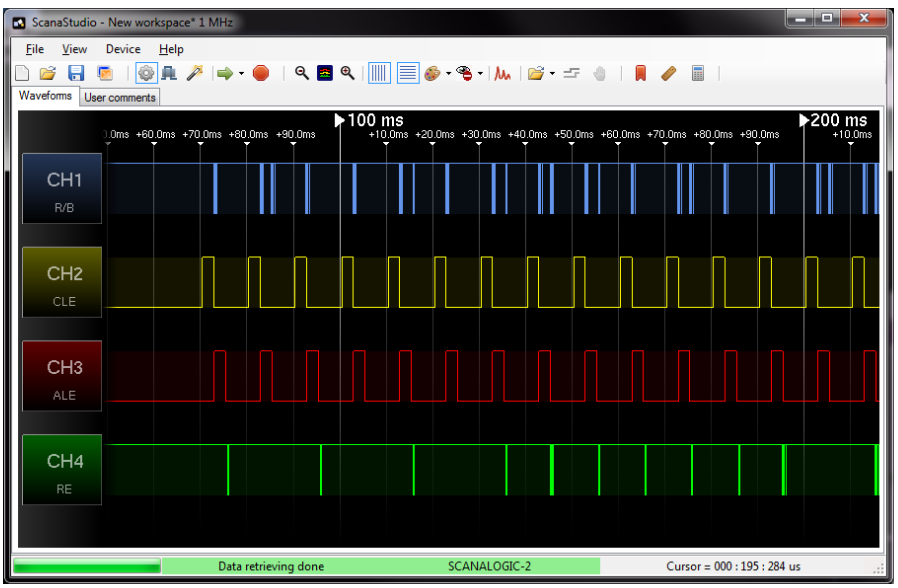


Figure 7 Repeated read operation

You also need to connect power lines to each side of the NAND Flash memory chip.

FT2232H	Use	NAND Flash	Pin number	Description
3v3	POWER	3v3	12	**POWER**
GND	GROUND	GND	13	**GROUND**
3v3	POWER	3v3	36	**POWER**
GND	GROUND	GND	37	**GROUND**

Table 5 FT2232H and NAND Flash Connections – Power

Besides these, the CE (Cheap Enable) pin (9) from the NAND Flash chip should be grounded. This means the chip is always enabled for normal operations.

NAND Flash chip command sets

Table 6 shows the basic command sets usually used by NAND Flash memory. There are more complicated commands available depending on the chipsets, but these basic commands are enough for essential operations like reading and writing data on the chip. Also, these commands tend to be the same across different models and brands. The pins and other descriptions presented here are mostly focused on small block NAND Flash models (512 bytes of data with 16 bytes OOB). Models with a large block size use a different set of commands, but the principle is same.

Function	1st cycle	2nd cycle
Read 1	00h/01h	-
Read 2	50h	-
Read ID	90h	-
Page Program	80h	10h
Block Erase	60h	D0h
Read Status	70h	

Table 6 Basic command sets for usual NAND Flash memory (small blocks)

Read operation

Every operation is done by page with Flash memory. To read a page, it uses the Read 1 (00h, 01h) and Read 2 (50h) functions. To read a full page with OOB data from small block Flash memory, you need to read it 3 times. The 00h command is used to read the first half of the page data (A area). The 01h command is used to read the second half of the page data (B area). Finally, to retrieve the OOB of the page (spare C area), the 50h command is used. Figure 8 shows the state of each pin when read operations are performed. CLE is set to high [1] when commands (00h, 01h, 50h) are passed. ALE is set to high [1] when addresses are transferred. R/B pin is set to low [0]) when the chip is busy preparing the data. RE and WE are used to indicate the readiness of the data operation on the I/O lines. When the WE signal is rising, new bytes (command and address in this case) are sent to the I/O pins. When the RE signal is falling, new bytes come from the NAND Flash memory chip if any data is available.

<table>
<tr><td>CLE</td><td>1</td><td colspan="4">0</td></tr>
<tr><td>ALE</td><td>0</td><td>1</td><td colspan="3">0</td></tr>
<tr><td>R/B</td><td colspan="2">1 (Ready)</td><td>R/B=0 (busy)</td><td colspan="2">1 (Ready)</td></tr>
<tr><td>RE</td><td colspan="3">1</td><td colspan="2">Falling for each bytes</td></tr>
<tr><td>WE</td><td colspan="2">Rising for each bytes</td><td colspan="3">1</td></tr>
<tr><td>I/O0~7</td><td>00h/01h
/50h</td><td>Start Address
A0 – A7 A9 – A25</td><td></td><td colspan="2">Data Output</td></tr>
</table>

Figure 8 Read operation pin states

Figure 9 shows a good example of how WE, CLE, ALE, and RE pin states change over time. First, the WE and CLE logic changes to send commands. Next, the WE and ALE change state to send addresses. Finally, RE is used to signal the reading of each byte.

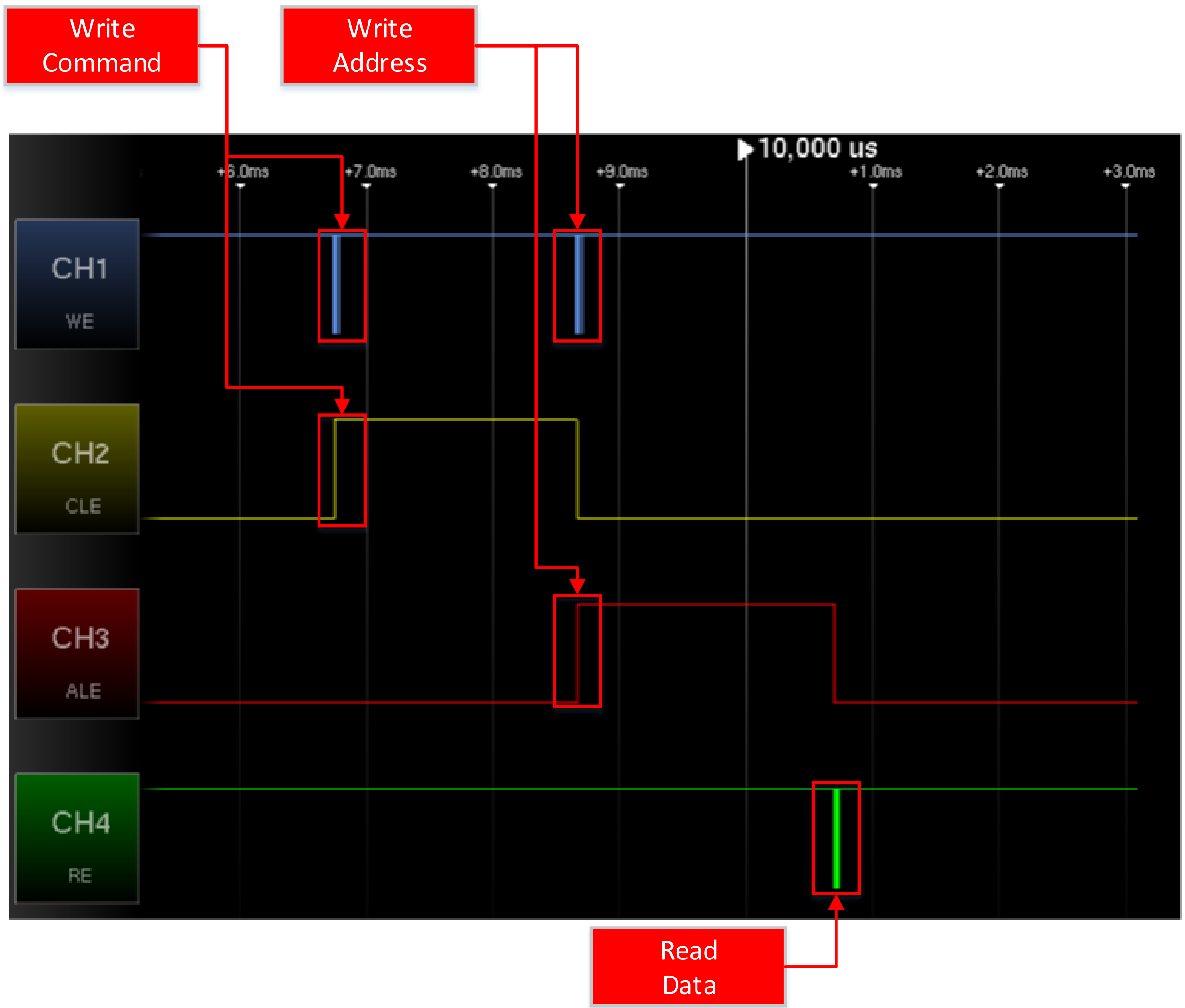


Figure 9 Reading data

From the FlashTool project, the code to read pages is implemented in a way similar to the examples shown in Figure 10 **Error! Reference source not found.**. The *readPage* method reads area A, B and the spare C area. The NAND_CMD_READ0 (00h), NAND_CMD_READ1 (01h) and NAND_CMD_READOOB (50h) commands are used to read each area.

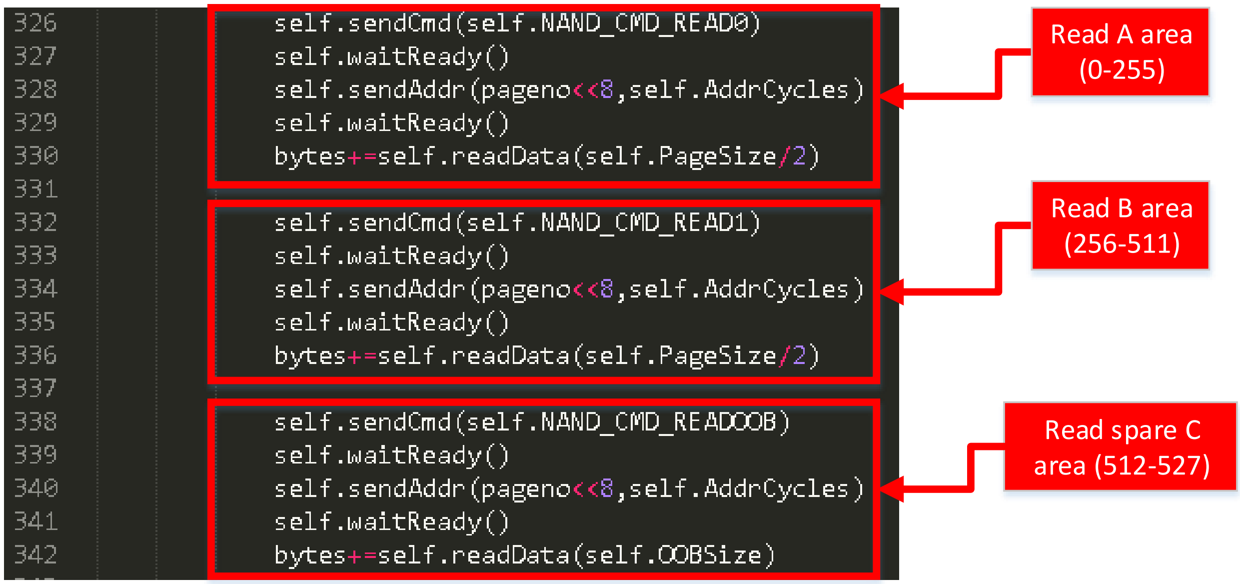


Figure 10 Reading a small block page

Write operation

Writing operations are done through sequence-in command (80h) and program command (10h). (Table 7) It uses a read status command (70h) to retrieve the result of the write operation. If the I/O0 is 0, it means the operation was successful.

<table>
<tr><th>CLE</th><td>1</td><td colspan="2">0</td><td colspan="4">1</td></tr>
<tr><th>ALE</th><td>0</td><td>1</td><td colspan="5">0</td></tr>
<tr><th>R/B</th><td colspan="4">1 (Ready)</td><td>R/B=0 (busy)</td><td colspan="2">1 (Ready)</td></tr>
<tr><th>RE</th><td colspan="6">1</td><td>Falling</td></tr>
<tr><th>WE</th><td colspan="4">Rising for each bytes</td><td>1</td><td>Rising</td><td>1</td></tr>
<tr><th>I/O0~7</th><td>80h</td><td>Address Input
A0 – A7 A9 – A25</td><td>Page + OOB data</td><td>10h</td><td></td><td>70h</td><td>I/O0=status</td></tr>
</table>

Table 7 Write operation pin states

Figure 11 shows the code that writes a page with a spare C area (OOB) from the FlashTool project. One thing to note is use of the NAND_CMD_READ0 (00h) at line 435, NAND_CMD_READ1 (01h) at line 446 and NAND_CMD_READOOB (50h) at line 457. Three commands are used for the reading operation, but they are also used for moving the writing pointer to the A, B and C areas. If a NAND_CMD_SEQIN (80h) command follows just after these commands, it just moves the pointer to each area. Additionally, there should be a NAND_CMD_PAGEPROG (10h) command to commit the writing operation.

Figure 11 Writing a small block page with spare C area

Figure 12 shows a good example of a writing operation. After a command and address are sent, WE fluctuates repeatedly to send bytes.

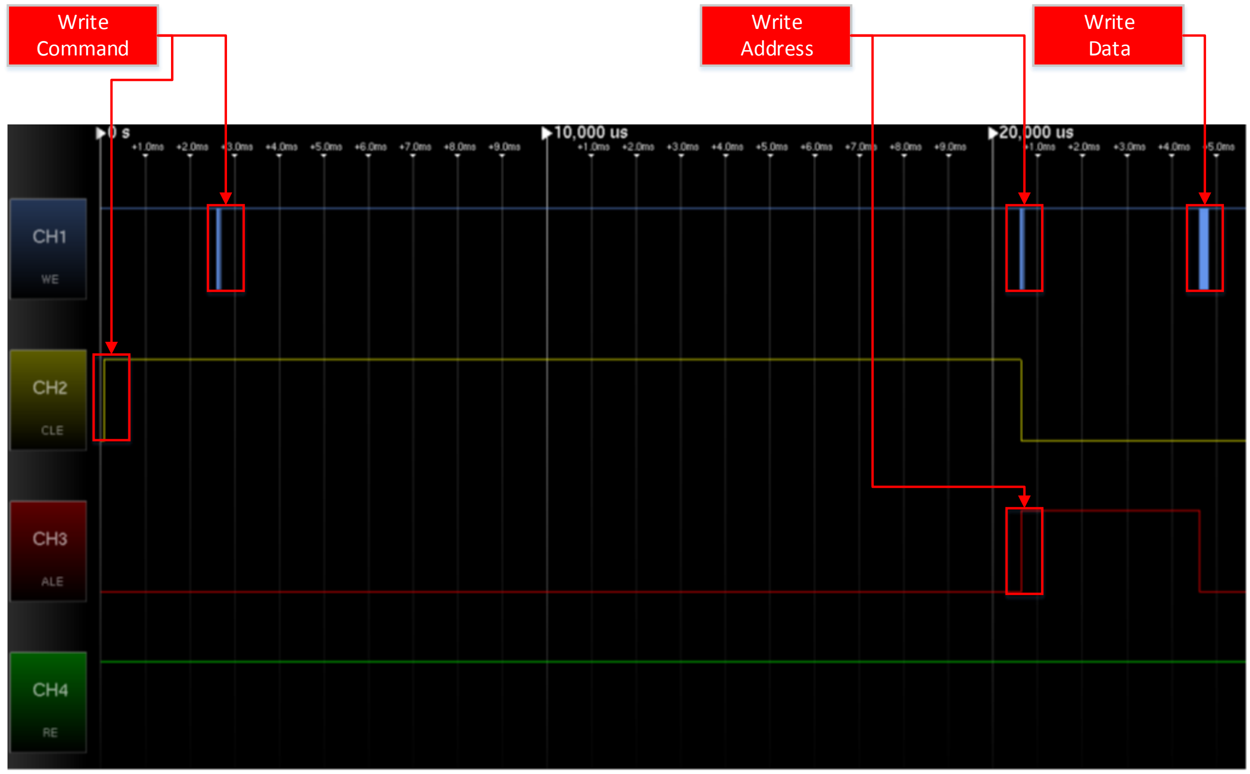


Figure 12 Writing Data

Reader writer

Figure 13 shows the final NAND Flash reader/writer assembled based on the connection information shown in Table 2. You can make a device like this even with a relatively low budget. You need an FTDI FT2232H breakout board, a USB cable, a TSOP48 socket, and wires.

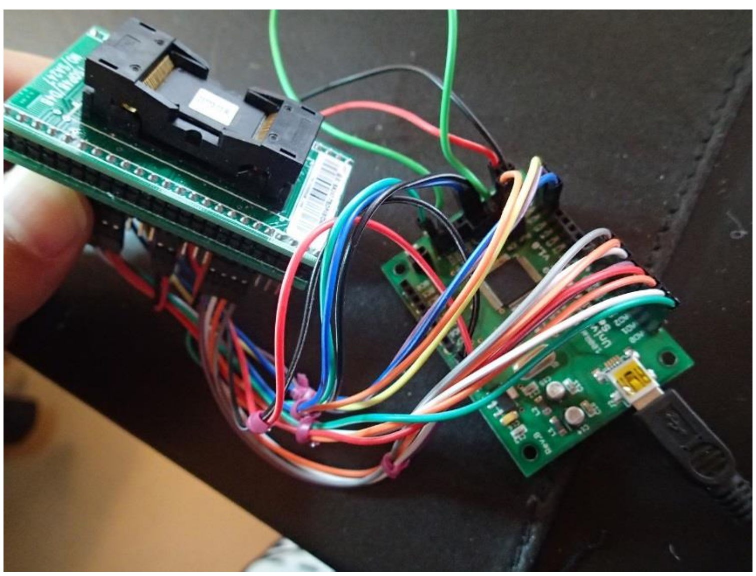


Figure 13 NAND Flash reader/writer

Place your NAND Flash chip inside the TSOP48 socket. (Figure 14) This socket is very useful as you can safely place your NAND Flash chip inside it and then directly interact with the extended pins without touching and possibly damaging any Flash memory chip pins.

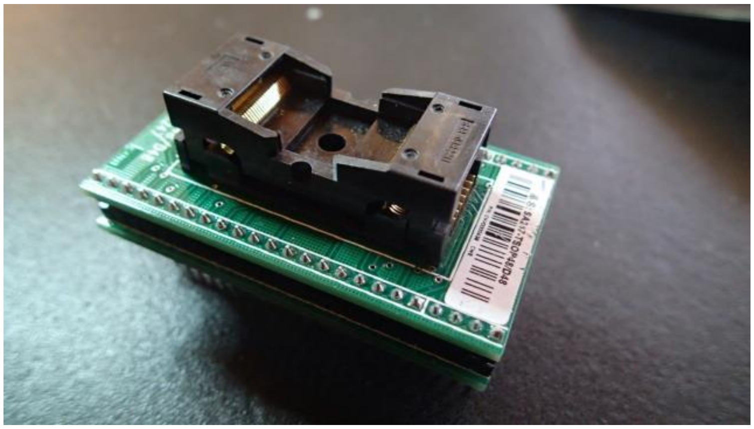


Figure 14 TSOP48 socket

When the NAND reader/writer is ready, just load the Flash memory. You should be careful to put the pin 1 location on the correct side of the socket. Usually the socket shows where pin 1 should be located. (Figure 15) When things are set, you can connect the reader/writer to the computer through a USB cable.

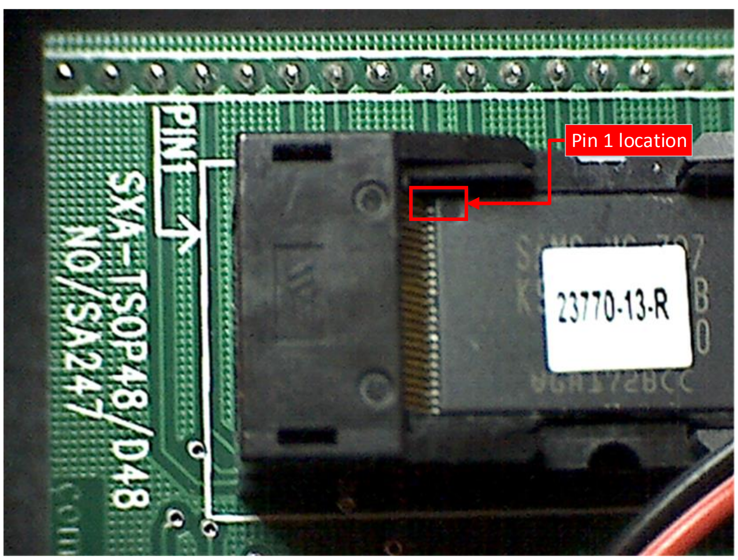


Figure 15 Pin 1 location

You need software to achieve bit-banging and there is a NANDTool open source project for this. I actually forked this project and created another experimental project here. Also, I ported whole C++ code to a Python project and made a FlashTool project. When the original project didn't support NAND

Flash programming, I put support in with some modifications to the original code. I use my project for this demonstration.

Download the FlashTool code from here first. You should install prerequisite packages like pyftdi and libusbx. With everything setup, you can query basic Flash information using the *–i* option. (Figure 16)

```
root@test:~# python FlashTool.py -i
Name:               NAND 64MiB 3,3V 8-bit
ID:                 0x76
Page size:          0x200
OOB size:           0x10
Page count:         0x20000
Size:               0x40
Erase size:         0x4000
Options:            0
Address cycle:      4
Manufacturer:       Samsung
```

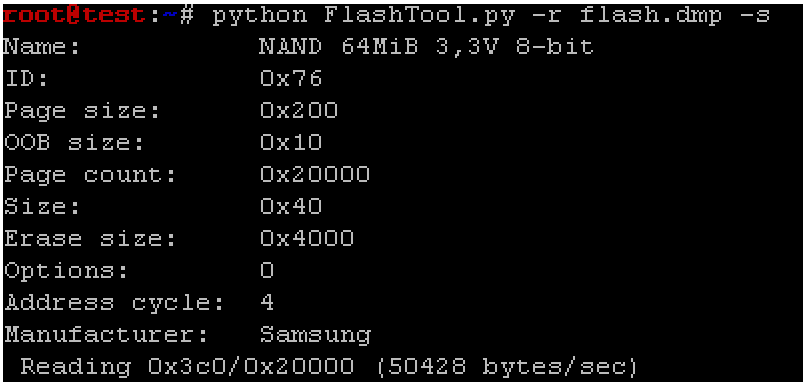
Figure 16 NANDTool -i (reading information)

You can also read the raw data (Figure 17). It takes some time to retrieve all the data depending on the size of the memory.

```
root@test:~# python FlashTool.py -r flash.dmp
Name:               NAND 64MiB 3,3V 8-bit
ID:                 0x76
Page size:          0x200
OOB size:           0x10
Page count:         0x20000
Size:               0x40
Erase size:         0x4000
Options:            0
Address cycle:      4
Manufacturer:       Samsung
 Reading 0x232/0x20000 (9586 bytes/sec)
```

Figure 17 Reading raw data

The FlashTool also supports sequential row read mode. When you can specify the *–s* option, it uses the mode and increases reading performance. The speed of reading is faster than normal page-by-page mode by 5-6 times in this case. (Figure 18)



```
root@test:~# python FlashTool.py -r flash.dmp -s
Name:            NAND 64MiB 3,3V 8-bit
ID:              0x76
Page size:       0x200
OOB size:        0x10
Page count:      0x20000
Size:            0x40
Erase size:      0x4000
Options:         0
Address cycle:   4
Manufacturer:    Samsung
 Reading 0x3c0/0x20000 (50428 bytes/sec)
```

Figure 18 Sequential Row read mode (-s)

Working with a bare metal image

NAND Flash memory is a physical device and there's always the chance that it will be affected by the randomness of nature. NAND Flash uses a spare column to save meta-data on each page. A page is the minimum element of data operation in NAND Flash as NAND Flash can't perform byte-by-byte operations. If you modify a byte from the page, it should rewrite the whole page with modified data. To counteract random failures, Flash memory uses two concepts; ECC (Error Correction Code) and bad blocks. This information is saved in the spare column of each page, which is also called the OOB area. (Figure 19)

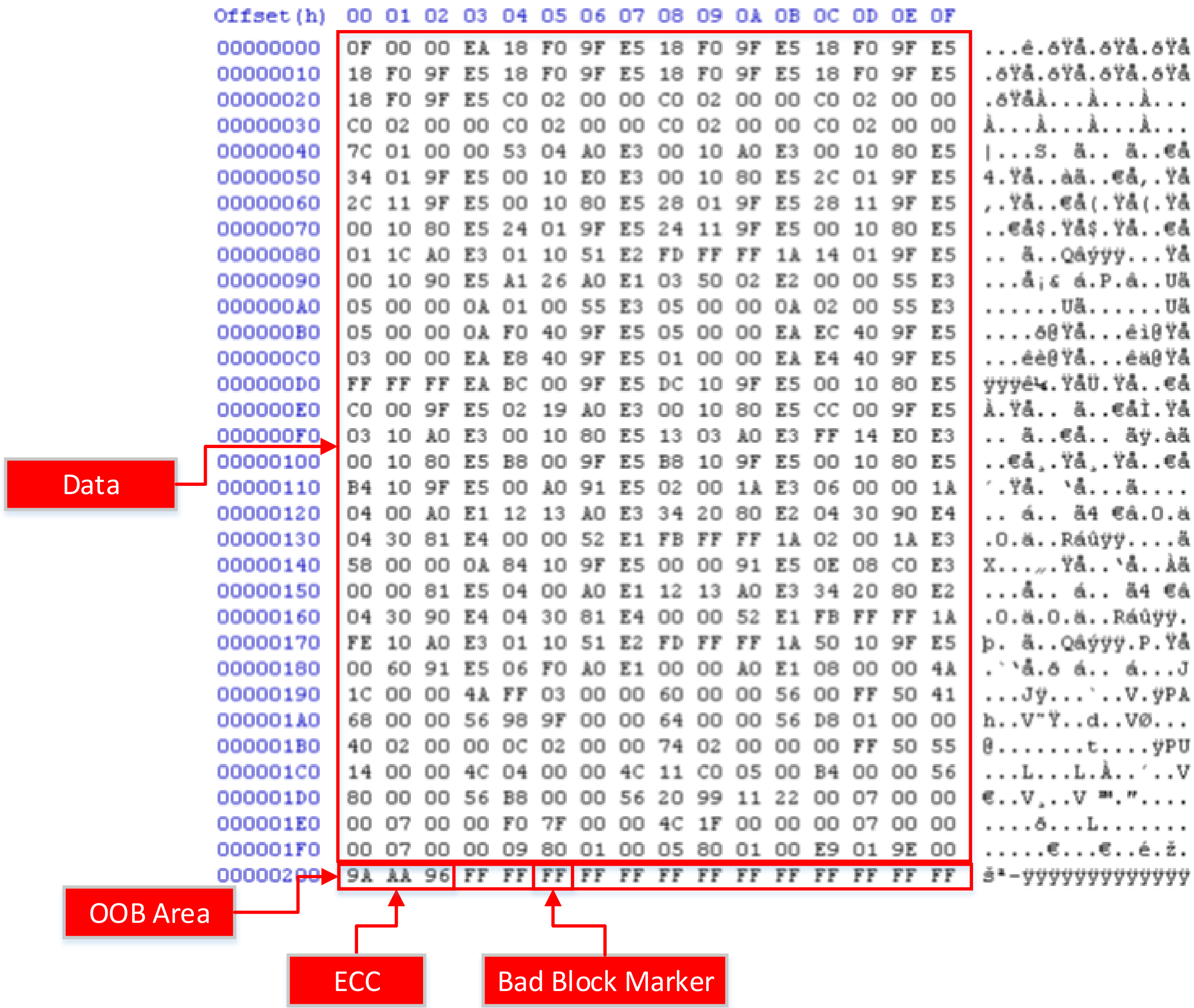


Figure 19 Data & OOB area

ECC

The ECC is a way to correct one bit of failure from a page. Failure can always occur with data on memory. A checksum can be useful to detect these errors. With ECC, besides detecting errors, it can correct them if they are minor. It uses the concept of Hamming code, invented by Richard Hamming in 1950. It was originally used for correcting errors with punch cards. (4)

Modern Flash memories use a different ECC algorithm with Hamming code as its root. Even similar chipsets from the same vendor might have slightly different ECC algorithms. But the differences are minor and are generally just tweaks of XOR or shifting orders or methods. The problem is that you need to figure out the correct algorithms to verify the validity of each page and to generate ECC later for page modification.

I'll show the ECC algorithm used by the chip sets I worked on (Samsung K9F1208). Figure 20 shows the table representation of bits on a page with a size of 512. Each bit is represented by a cell and each row is one byte. From this matrix, first, you can calculate various checksums across bits.

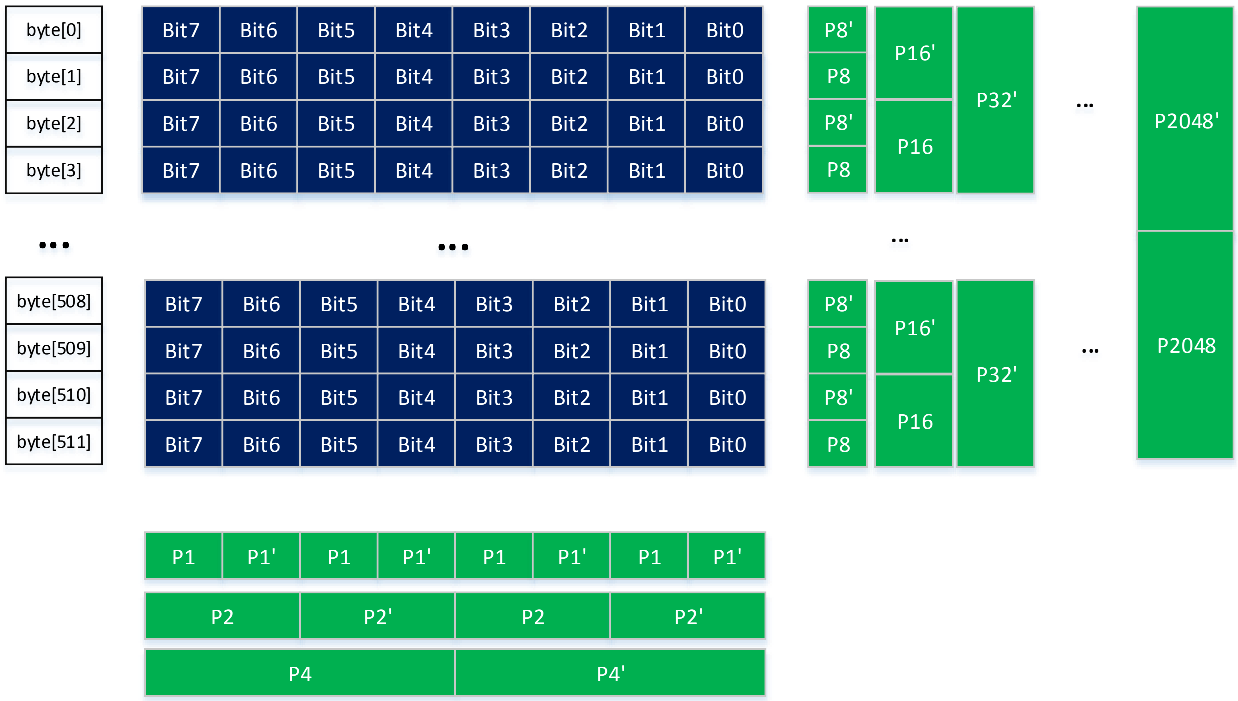


Figure 20 ECC calculation table

For example, P8' checksum is calculated by XOR-ing all the bits in red in Figure 21.

Figure 21 P8' calculation

Figure 22 shows the example of calculating P16'. It uses bits from byte[0], bytes[1], byte[4], byte[5] and so on until byte[508] and byte[509] for checksum calculation. Other column checksums like P8, P16', P16, P32', P32, P2048' and P2048 are calculated in same manner.

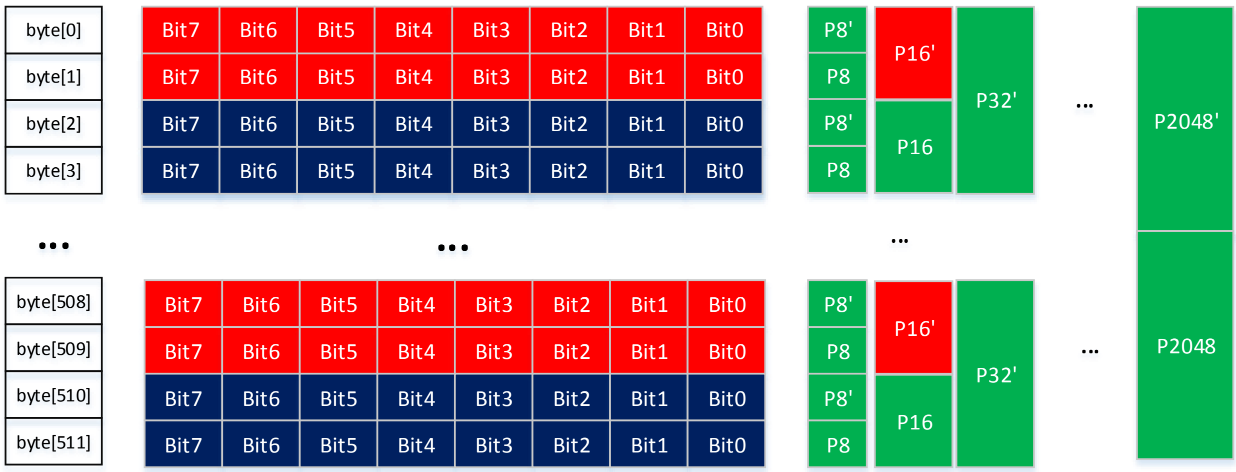


Figure 22 P16' calculation

Figure 23 shows the example code that implements this algorithm.

```
if i & 0x01 == 0x01:
    p8 = xor_bit ^ p8
else:
    p8_ = xor_bit ^ p8_

if i & 0x02 == 0x02:
    p16 = xor_bit ^ p16
else:
    p16_ = xor_bit ^ p16_

if i & 0x04 == 0x04:
    p32 = xor_bit ^ p32
else:
    p32_ = xor_bit ^ p32_

if i & 0x08 == 0x08:
    p64 = xor_bit ^ p64
else:
    p64_ = xor_bit ^ p64_

if i & 0x10 == 0x10:
    p128 = xor_bit ^ p128
else:
    p128_ = xor_bit ^ p128_

if i & 0x20 == 0x20:
    p256 = xor_bit ^ p256
else:
    p256_ = xor_bit ^ p256_

if i & 0x40 == 0x40:
    p512 = xor_bit ^ p512
else:
    p512_ = xor_bit ^ p512_

if i & 0x80 == 0x80:
    p1024 = xor_bit ^ p1024
else:
    p1024_ = xor_bit ^ p1024_

if i & 0x100 == 0x100:
    p2048 = xor_bit ^ p2048
else:
    p2048_ = xor_bit ^ p2048_
```

Figure 23 Code for calculating row checksums

The column checksums are calculated over the same bit locations over all the bytes in the page. For example, Figure 24 shows how P2 can be calculated by taking bits 2,3,6,7 from each byte.

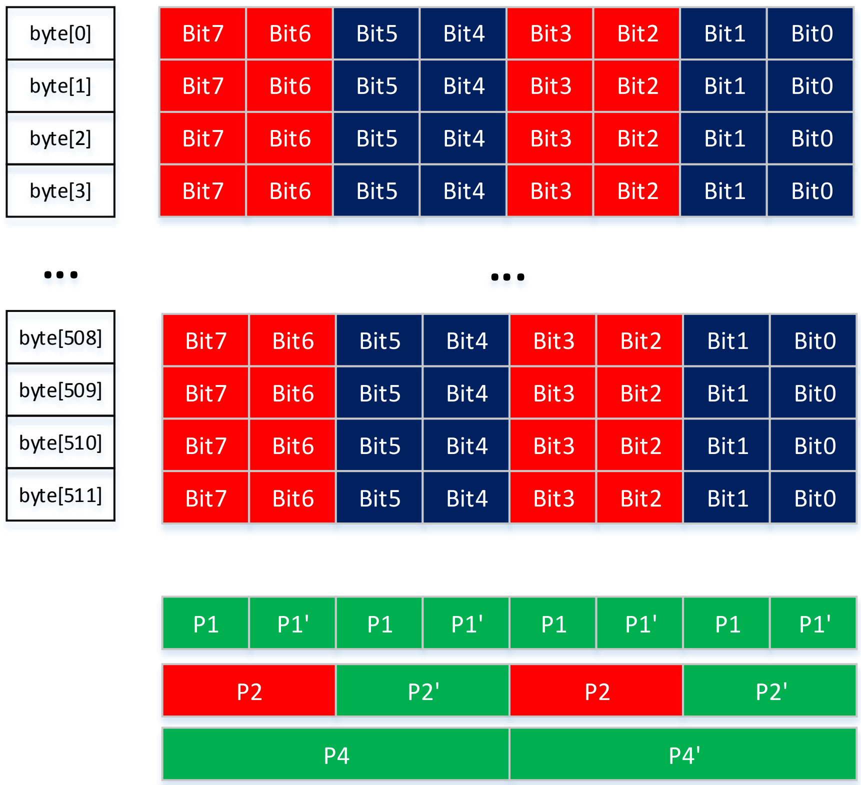


Figure 24 P2 calculation

Figure 25 shows the code that calculates column checksums.

```
p1 = bit7 ^ bit5 ^ bit3 ^ bit1 ^ p1
p1_ = bit6 ^ bit4 ^ bit2 ^ bit0 ^ p1_
p2 = bit7 ^ bit6 ^ bit3 ^ bit2 ^ p2
p2_ = bit5 ^ bit4 ^ bit1 ^ bit0 ^ p2_
p4 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ p4
p4_ = bit3 ^ bit2 ^ bit1 ^ bit0 ^ p4_
```

Figure 25 Row checksum calculation code

Finally, you need to calculate 3 ECC values based on the checksums calculated. The row and column checksum methods are very similar for different NAND Flash memory models, but ECC calculations tend to be slightly different across different models. The code in Figure 26 shows the algorithm used for the specific model I worked on.

```
ecc0 = (p64 << 7) + (p64_ << 6) + (p32 << 5) + (p32_ << 4) + (p16 << 3) + (p16_ << 2) + ( p8 << 1) + ( p8_ << 0)
ecc1 = (p1024 << 7) + (p1024_ << 6) + (p512 << 5) + (p512_ << 4) + (p256 << 3) + (p256_ << 2) + (p128 << 1) + (p128_<< 0)
ecc2 = (p4 << 7) + (p4_ << 6) + (p2 << 5) + (p2_ << 4) + (p1 << 3) + (p1_ << 2) + (p2048 << 1) + (p2048_ << 0)
```

Figure 26 ECC calculation code

Bad blocks

'Bad blocks' is a generic concept that is also used in hard disk technology. With Flash memory, if errors are more than the ECC can handle, it marks the entire block as bad. Those blocks are isolated from other blocks and are no longer used. To mark bad blocks, the first or last pages are used for marking, according to the ONFI standard. Some vendors use their own scheme for marking bad blocks. Figure 27 shows one of the examples for checking bad blocks from the DumpFlash project. If the 6^{th} byte from the OOB data of the first or second page for each block has non FFh values, it is recognized as a bad block. This scheme is used by multiple vendors including Samsung and Micron.

```python
def IsBadBlock(self,block):
    for page in range(0,2,1):
        block_offset = (block * self.BlockSize ) + (page * (self.PageSize + self.OOBSize))
        self.fd.seek( block_offset + self.PageSize + 5 )
        bad_block_byte = self.fd.read(1)

        if not bad_block_byte:
            return self.ERROR

        if bad_block_byte == '\xff':
            return self.CLEAN_BLOCK

    return self.BAD_BLOCK
```

Figure 27 Example bad block check routine

```
C:\mat\Analysis\NAND Flash\Tool>c:\python27\python DumpFlash.py -b flash.dmp
Opening flash.dmp
Check bad blocks:
Bad block: 3622 (at 0x3a5cc00)
Bad block: 3626 (at 0x3a6d400)
Bad block: 3978 (at 0x4019400)
Checked 4096 blocks and found 3 errors
```

Figure 28 Using DumpFlash tool to find bad blocks

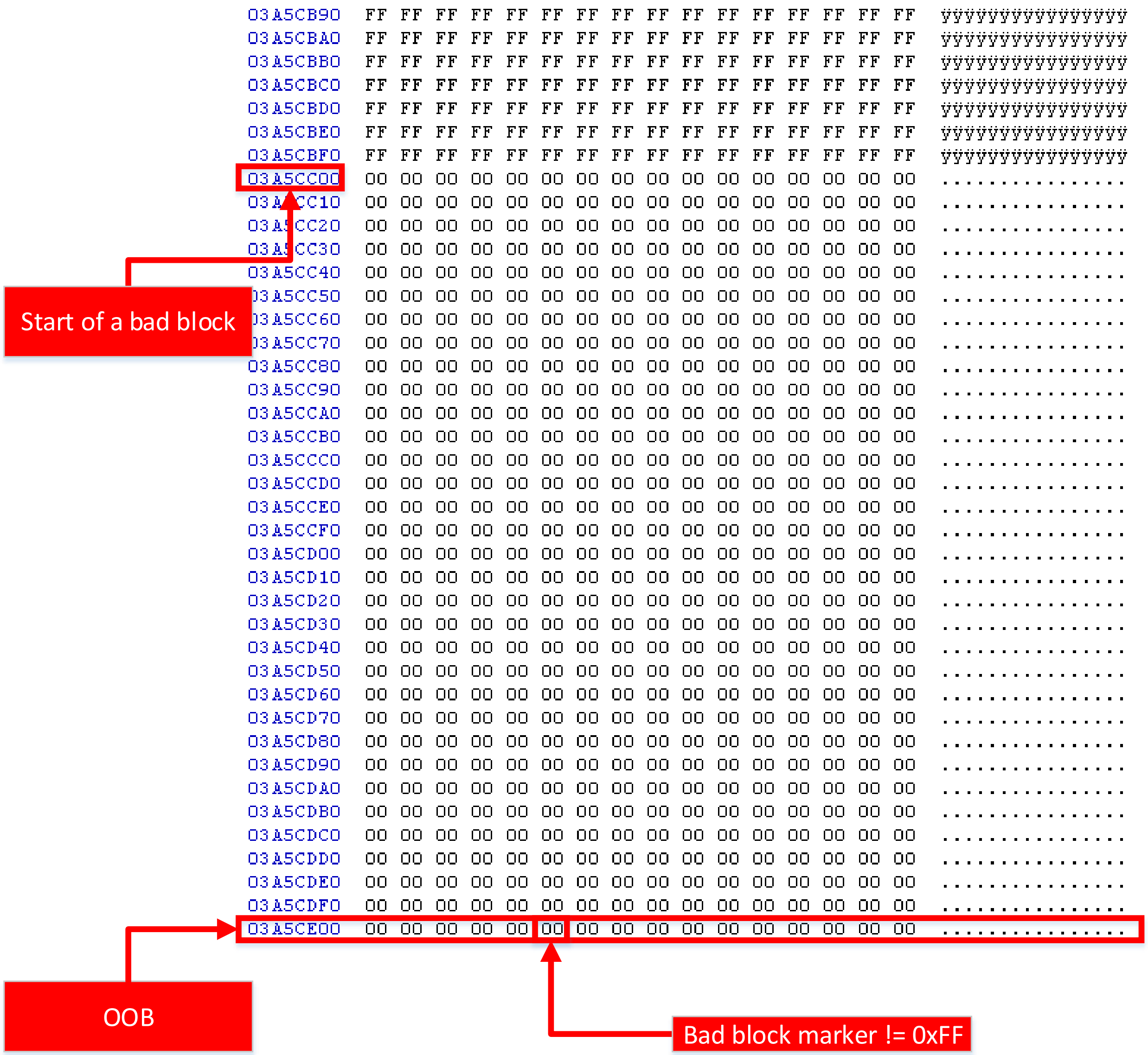


Figure 29 How a bad block is marked

Reverse engineering Flash memory data

When the NAND Flash memory is used for booting up embedded systems, the structure usually looks similar to Figure 30. The first block is always loaded first to address 0x00000000 during the boot-up process. After that U-Boot code and images follow. When the boot-loading code and U-Boot images are read only, the JFFS2 file system is used for reading and writing.

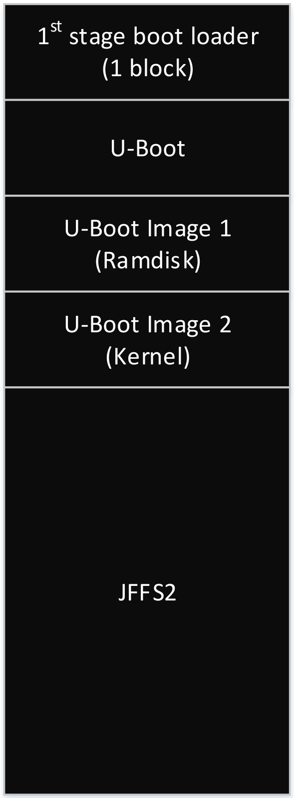


Figure 30 An example of Flash memory layout

1st stage boot loader

This boot loader does low level initialization. (Figure 31)

```
ROM:00000178 loc_178                               ; CODE XREF: ROM:00000158↑j
ROM:00000178                 TST             R10, #2
ROM:0000017C                 BEQ             loc_2EC
ROM:00000180                 LDR             R1, =0x56000080 ; S3C2410X_MISCCR
ROM:00000184                 LDR             R0, [R1]
ROM:00000188                 BIC             R0, R0, #0xE0000
ROM:0000018C                 STR             R0, [R1]
ROM:00000190                 MOV             R0, R4  ; R4: bytes to send to bus
ROM:00000194                 MOV             R1, #0x48000000 ; S3C2410X_BWSCON
ROM:00000198                 ADD             R2, R0, #0x34
ROM:0000019C
ROM:0000019C loc_19C                               ; CODE XREF: ROM:000001A8↓j
ROM:0000019C                 LDR             R3, [R0],#4
ROM:000001A0                 STR             R3, [R1],#4 ; R0: bytes to send to bus
ROM:000001A4                 CMP             R2, R0
ROM:000001A8                 BNE             loc_19C
ROM:000001AC                 MOV             R1, #0xFE ; '¦'
ROM:000001B0
ROM:000001B0 loc_1B0                               ; CODE XREF: ROM:000001B4↓j
ROM:000001B0                 SUBS            R1, R1, #1
ROM:000001B4                 BNE             loc_1B0
ROM:000001B8                 LDR             R1, =0x560000B8 ; S3C2410X_GSTATUS3
ROM:000001BC                 LDR             R6, [R1]
ROM:000001C0                 MOV             PC, R6
```

Figure 31 Low level initialization of the system

It also loads up the next level boot loader. Figure 32 from the image I worked on shows very interesting strings like the name of the first boot loader and some log messages on the next level boot loader.

```
ROM:00000DF5 aNandBootloader DCB "Nand Bootloader(ADAM) 3.2.4",0xA
ROM:00000DF5                 DCB "    ",0
ROM:00000E16                 DCB    0
ROM:00000E17                 DCB    0
ROM:00000E18                 DCB  0xA
ROM:00000E19 aLoadingUBoot   DCB "Loading U-BOOT ",0xA
ROM:00000E19                 DCB " ",0
ROM:00000E2B                 DCB    0
ROM:00000E2C                 DCB  0xA
ROM:00000E2D                 DCB  0xA
ROM:00000E2E aUBootExit      DCB "U-Boot EXIT",0xA,0
```

Figure 32 Strings from the first stage boot loader

U-Boot loader

After the first stage boot loader, there is a next level boot loader that can perform various complicated operations. U-Boot loader is a very popular choice amongst embedded systems. The kernel image and actual file system are placed with them.

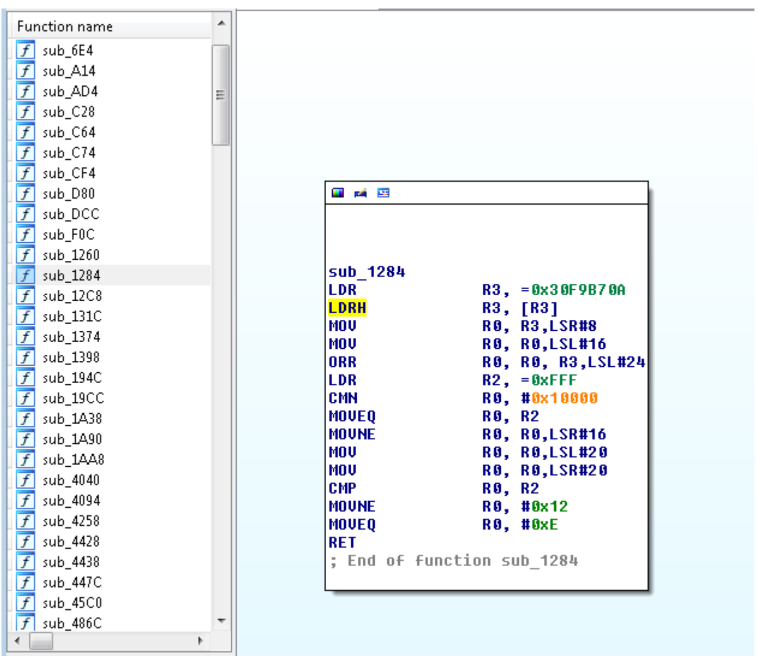


Figure 33 U-boot boot code

U-Boot images

The U-Boot image usually follows the U-Boot loader code. If the first 4 bytes of a block starts with the U-boot magic DWORD 0x56190527, then it’s probably a U-Boot image. Figure 34 shows the image header definition that contains the magic value.

```c
#define IH_MAGIC    0x27051956  /* Image Magic Number       */
#define IH_NMLEN            32  /* Image Name Length        */

/*
 * Legacy format image header,
 * all data in network byte order (aka natural aka bigendian).
 */
typedef struct image_header {
    uint32_t    ih_magic;   /* Image Header Magic Number    */
    uint32_t    ih_hcrc;    /* Image Header CRC Checksum    */
    uint32_t    ih_time;    /* Image Creation Timestamp */
    uint32_t    ih_size;    /* Image Data Size      */
    uint32_t    ih_load;    /* Data  Load  Address      */
    uint32_t    ih_ep;      /* Entry Point Address      */
    uint32_t    ih_dcrc;    /* Image Data CRC Checksum  */
    uint8_t     ih_os;      /* Operating System     */
    uint8_t     ih_arch;    /* CPU architecture     */
    uint8_t     ih_type;    /* Image Type           */
    uint8_t     ih_comp;    /* Compression Type     */
    uint8_t     ih_name[IH_NMLEN];  /* Image Name       */
} image_header_t;
```

Figure 34 U-Boot image header structure

For example, Figure 35 shows a typical U-Boot image header. The important value in retrieving the whole image file is the image length. The header size is 0x40 and image length is 0x28A03B in this case. This makes the total image size 0x28A07B.

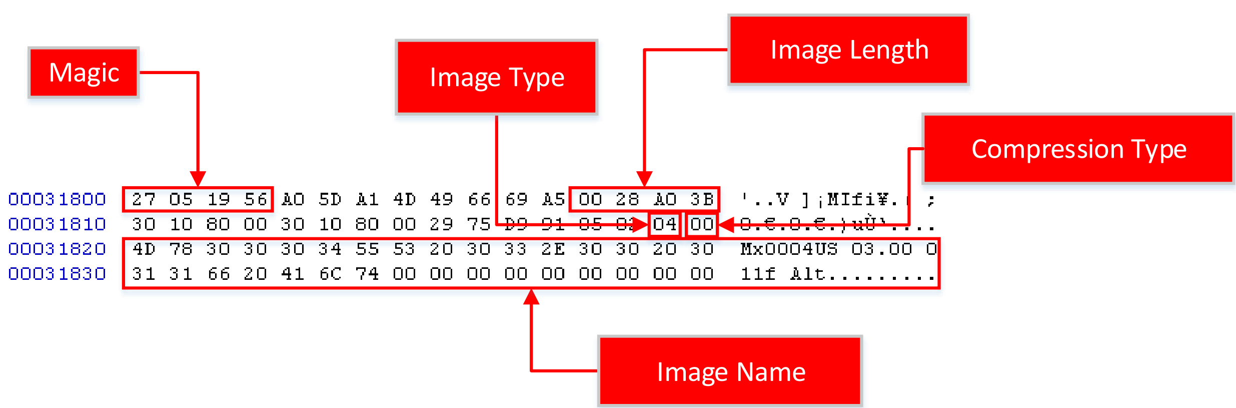


Figure 35 Typical U-Boot Image header

For my example, one page is 0x200 bytes, so the page count of the U-Boot image is 0x28A07B/0x200 = 0x1450. There are additional 0x28A07B%0x200 = 0x7B bytes above these pages. One page on the NAND dump image is 0x210 because of the extra OOB size (0x10). So the physical address of the image end is similar to the following:

page count = 0x1450

extra data = 0x7B

*page count * (page size + oob size) + extra data*

= 0x1450 * (0x200 + 0x10) + 0x7b

= 0x29E57B

The start address of the image is 0x31800 and if you add up this to the size of the image on the NAND image (0x29E57B), it becomes 0x2CFD7B.

You can extract this image by running the following command using the *–r* option designating the start and end addresses of the data.

```
python DumpFlash.py -r 0x00031800 0x002CFD7B -o Dump-00031800-UBOOT.dmp flash.dmp
```

Interestingly, IDA supports loading U-Boot images. (Figure 36)

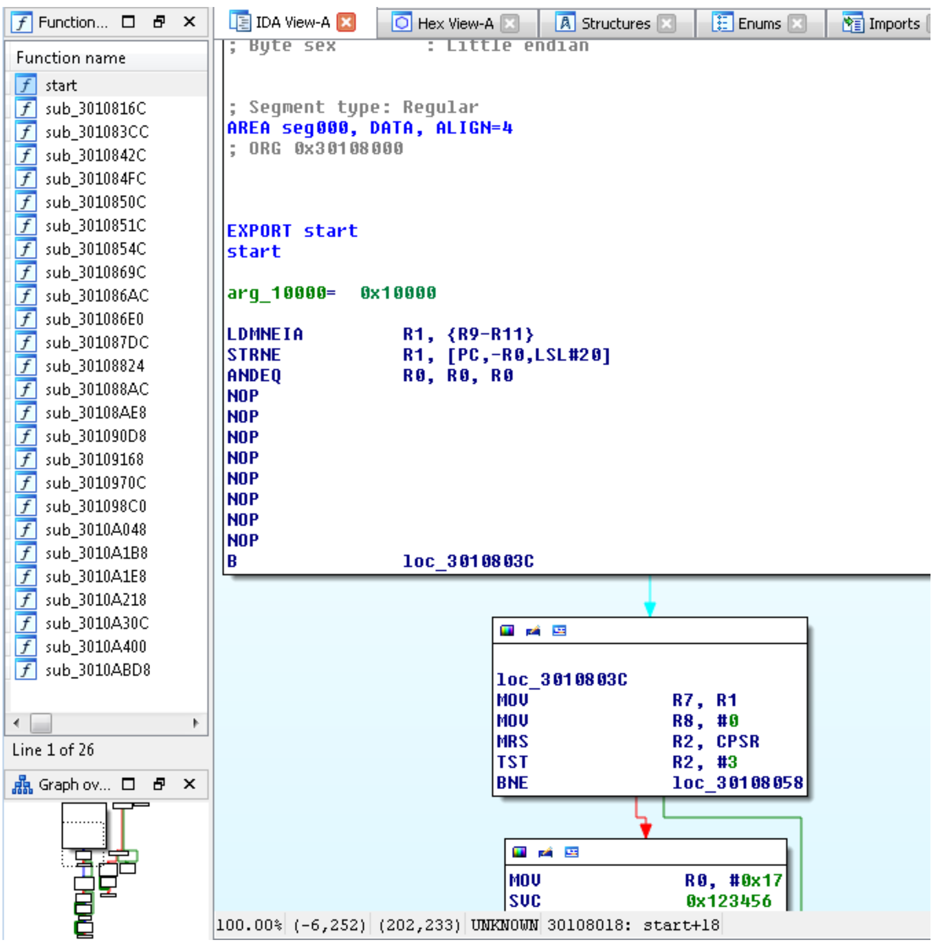


Figure 36 U-Boot Image Disassembly

However, manually parsing the image still helps us to understand the internals, and IDA doesn't do well with multi-file images. Figure 37 shows the U-Boot header and multi-file length fields after that. The

DWORD 0x00000000 marks the end of length fields. For this image it has two images inside it with lengths of 0x000E9118 and 0x001A0F17.

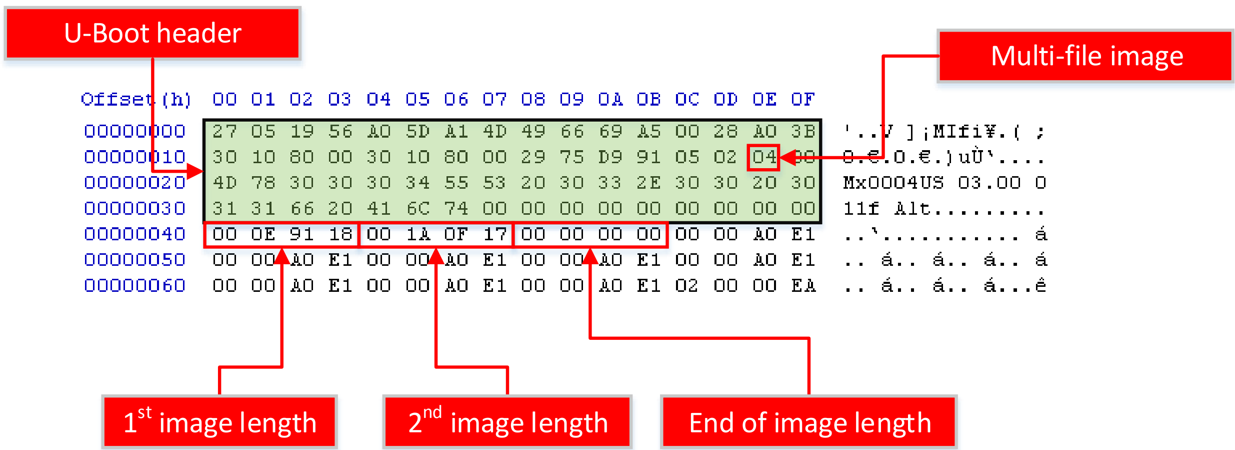


Figure 37 Multi-file image

You can also use the *mkimage* command to check the content of the U-Boot file. (Figure 38)

```
root@test:~# mkimage -l Dump-00031800-UBOOT.dmp
Image Name:   Mx0004US 03.00 011f Alt
Created:      Thu Jan  8 13:01:25 2009
Image Type:   ARM Linux Multi-File Image (uncompressed)
Data Size:    2662459 Bytes = 2600.06 kB = 2.54 MB
Load Address: 30108000
Entry Point:  30108000
Contents:
   Image 0: 954648 Bytes = 932.27 kB = 0.91 MB
   Image 1: 1707799 Bytes = 1667.77 kB = 1.63 MB
```

Figure 38 mkimage result

Ramdisk image

When image 0 looks like a code file, image 1 has more interesting contents. By just fiddling around with it you can identify that it is gzip compressed. After decompression, if you run the file command on the file, it looks like Figure 39, which shows that the file is an ext2 file system file.

```
root@test:~# file 02.decompressed.img
02.decompressed.img: Linux rev 1.0 ext2 filesystem data, UUID=42ba98f4-ee44-494e
-bddf-22d139c313b8
```

Figure 39 File command result on the 02.decompressed.img

You can mount the file on the Linux system using MTD. First, load MTD related kernel modules. (Figure 40)

```
root@kali:~# modprobe mtdram total_size=65536
root@kali:~# modprobe mtdblock
```

Figure 40 Loading MTD modules

You can use *dd* to copy the image to the MTD block device. (Figure 41)

```
root@test:~# dd if=02.decompressed.img of=/dev/mtdblock0
16384+0 records in
16384+0 records out
8388608 bytes (8.4 MB) copied, 0.0928797 s, 90.3 MB/s
```

Figure 41 Using dd to copy image

After copying the image to the MTD device, you can mount it using the *mount* command. (Figure 42)

```
root@test:~# mount /dev/mtdblock0 /tmp/mtd -t ext2
root@test:~# ls -la /tmp/mtd
total 51
drwxr-xr-x  17 root root  1024 Jan  8  2009 .
drwxrwxrwt  10 root root  4096 Jun 10 08:46 ..
drwxr-xr-x.  2 root root  2048 Jan  8  2009 bin
drwxr-xr-x.  2 root root  1024 Jan  8  2009 boot
drwxr-xr-x.  5 root root  4096 Jan  8  2009 dev
drwxr-xr-x.  3 root root  1024 Jan  8  2009 etc
drwxr-xr-x.  2 root root  1024 Jan  8  2009 home
drwxr-xr-x.  2 root root  1024 Jan  8  2009 initrd
drwxr-xr-x.  3 root root  1024 Jan  8  2009 lib
lrwxrwxrwx.  1 root root    11 Jan  8  2009 linuxrc -> bin/busybox
drwx------   2 root root 12288 Jan  8  2009 lost+found
drwxr-xr-x.  5 root root  1024 Jan  8  2009 mnt
drwxr-xr-x.  2 root root  1024 Jan  8  2009 proc
drwxr-xr-x.  2 root root  1024 Jan  8  2009 root
drwxr-xr-x.  2 root root  2048 Jan  8  2009 sbin
drwxr-xr-x.  2 root root  1024 Jan  8  2009 sys
drwxr-xr-x.  4 root root  1024 Jan  8  2009 usr
drwxr-xr-x.  2 root root  1024 Jan  8  2009 var
```

Figure 42 Mounting the device

Kernel image

With the image I worked on, I found another U-Boot image. The basic image information is shown in Figure 43.

```
root@test:~# mkimage -l Dump-00349800-UBOOT.dmp
Image Name:   Mx0004US 01.00 011
Created:      Mon Mar 31 11:30:37 2008
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    953052 Bytes = 930.71 kB = 0.91 MB
Load Address: 30108000
Entry Point:  30108000
```

Figure 43 mkimage information for second U-Bootimage

IDA loads up this image without any issues. The only problem is that the code shown by IDA is the bootstrapping code that decompresses following the gzipped kernel image. To identify the start of the kernel image, you can search for the gzip image magic value (0x8b1f) as shown in Figure 44.

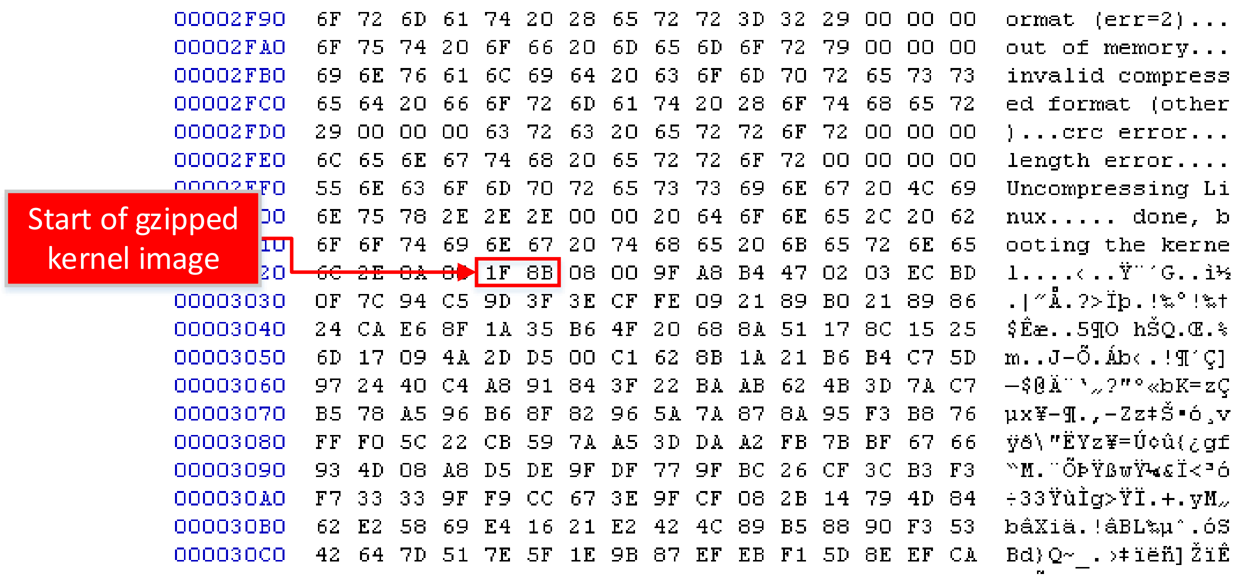


Figure 44 Start of compressed image

After you take out the image starting from the gzip magic bytes, you can decompress the image using any decompression utility that supports the gzip format. After it is decompressed you can load up the image using IDA. (Figure 45)

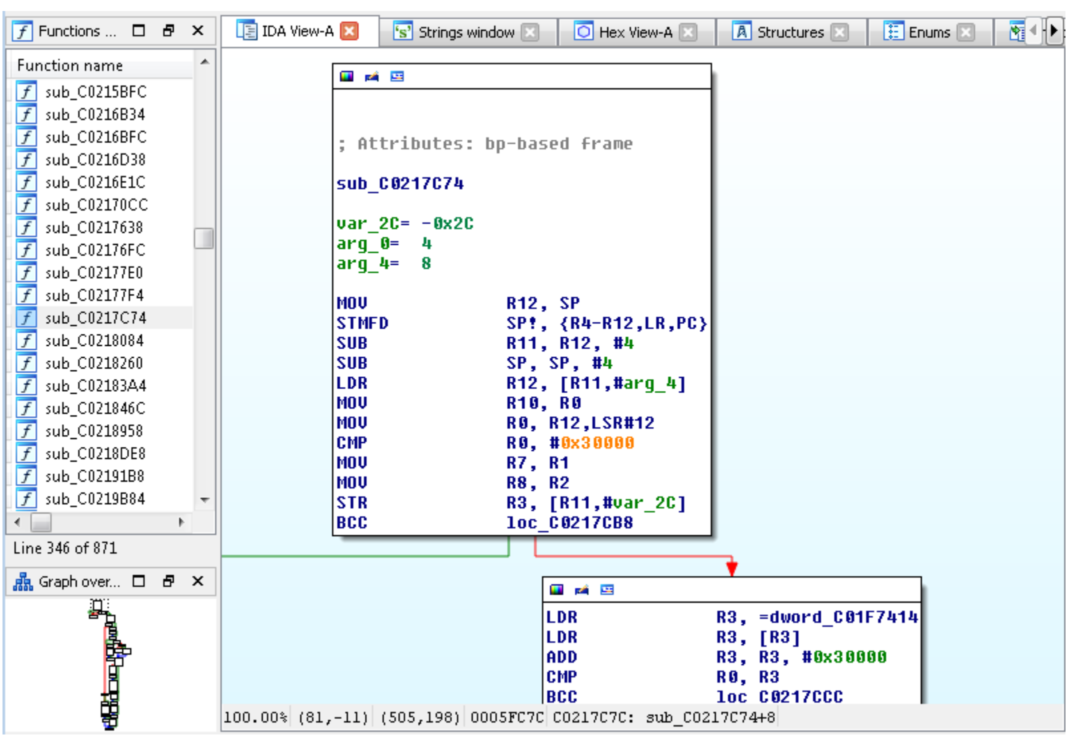


Figure 45 Kernel Image Disassembly

JFFS2

From the whole layout, the JFFS2 file system is at the core of the data analysis. The boot loaders are usually based on very generic code. Many interesting custom files are placed under the JFFS2 file system. Identifying the JFFS2 file system from the raw NAND Flash image is relatively easy. Usually JFFS2 puts specialized *erasemarkers* inside the spare column of each page. The *erasemarkers* are inserted when the NAND Flash memory is formatted with JFFS2 file system tools. This indicates that the block is used by JFFS2 and doesn't need additional initialization. Ideally, the *erasemarker*s would be located at every first page of each block. But, in reality it can present in every few blocks if the file system was created with a block size different from the real NAND Flash memory block size. This doesn't prevent JFFS2 from working correctly, but might challenge performance.

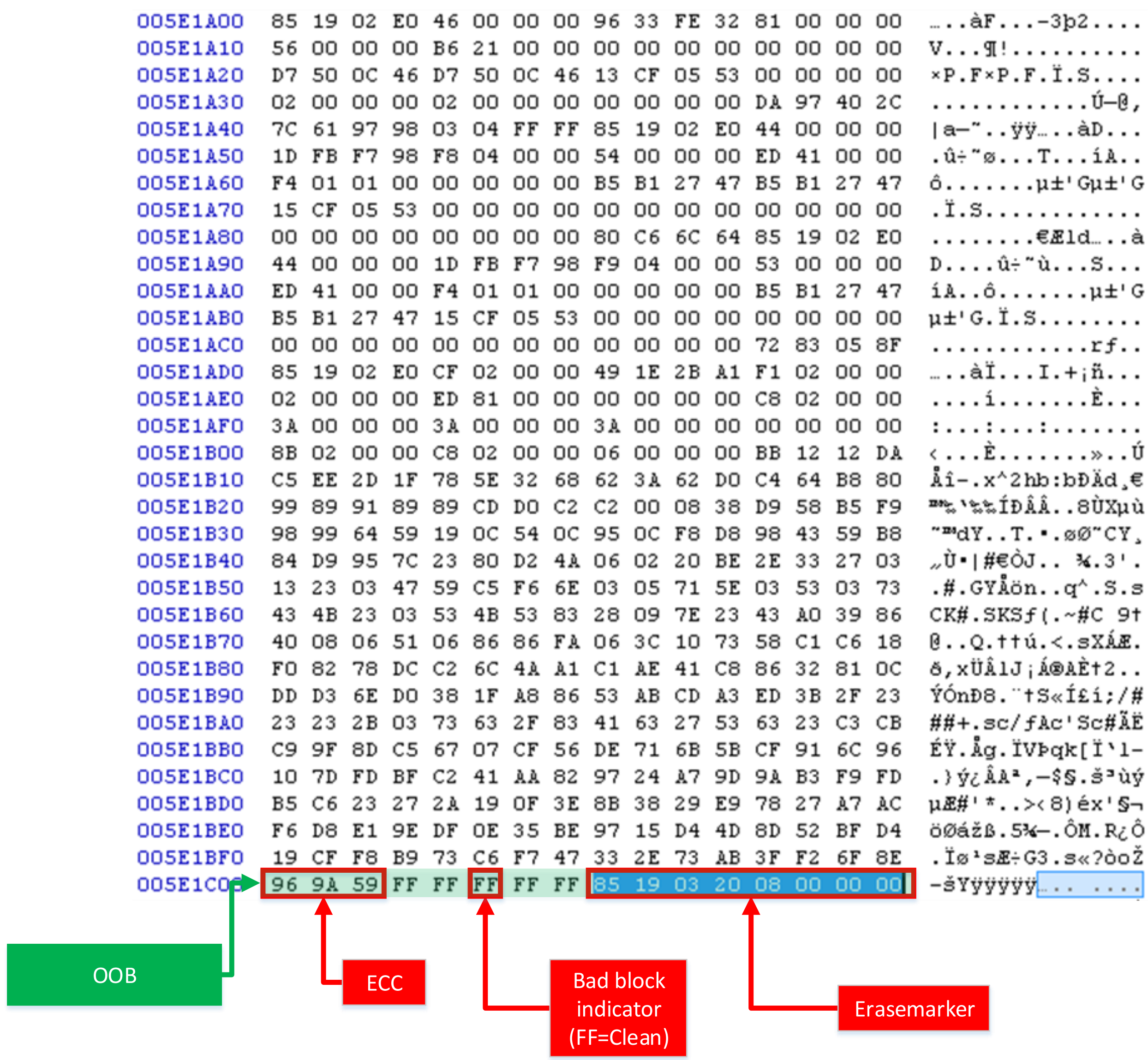


Figure 46 JFFS2 Erase Marker location from a page and spare column bytes

After identifying the start of the JFFS2 file system, you can extract the whole image. You need to verify if any bad blocks are present in the middle, check ECC for each block and remove the spare column from the original bytes. To assist with this process, I released a tool called [DumpFlash.py](). To extract part of the Flash memory, you just pass the start and end addresses after the *–r* option. You can put an output file name after the *–o* option. The following command dumps out the JFFS2 file system (*at address 0x0262c200 ~ 0x03084600)* bytes from the *flash.dmp* file. (Figure 47)

python DumpFlash.py -r 0x0262c200 0x03084600 -o jffs2.dmp flash.dmp

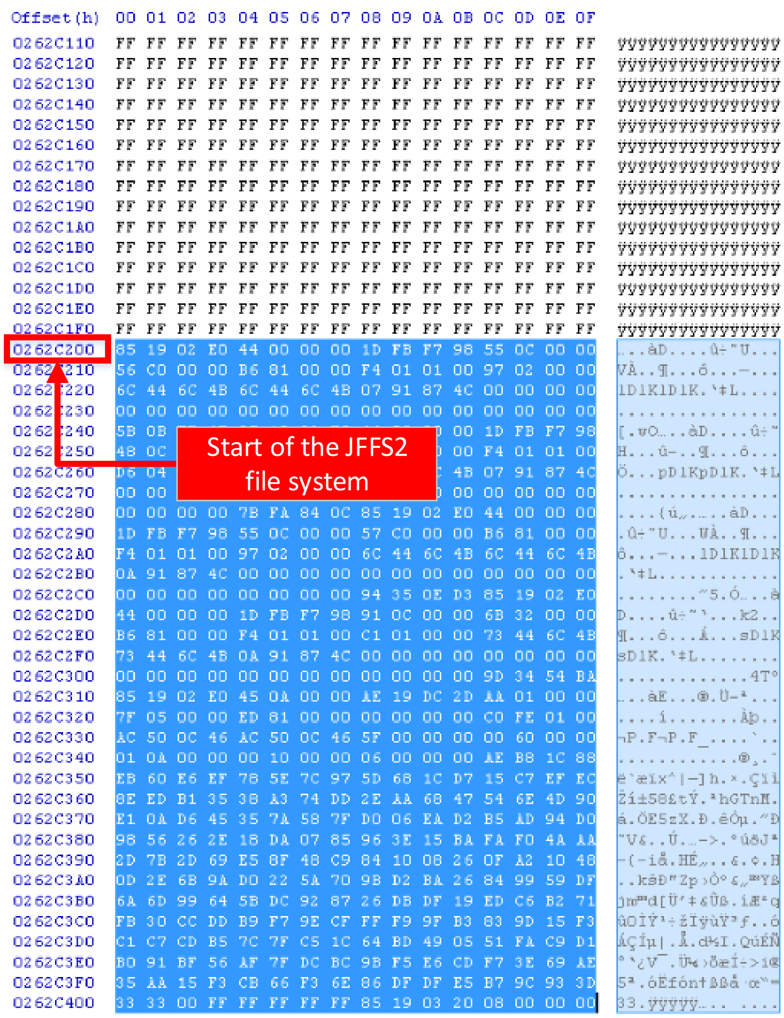


Figure 47 Example of start address of a JFFS2 file system

Mounting the JFFS2 file system using MTD

Now you can mount the JFFS2 raw image on the Linux operating system. First, you need to create an MTD device. Load related Linux kernel modules like *mtdram*, *mtdblock* and *jffs2* first. (Figure 48) This creates an MTD device on the system.

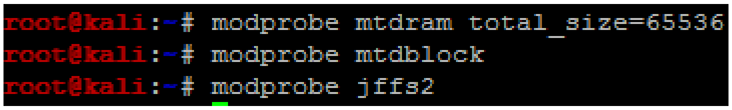


Figure 48 Loading related kernel modules

Use the *dd* utility to initialize the data of the MTD block device and mount the device to an arbitrary location. (Figure 49)

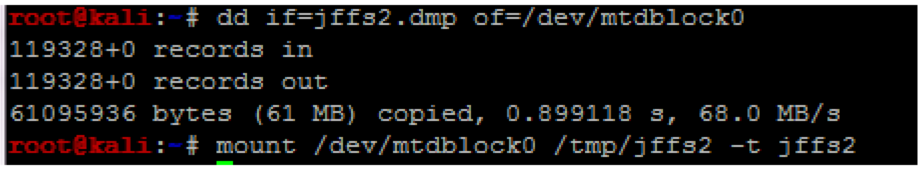


Figure 49 Mount MTD block device

After successful mounting, you can navigate and modify the file system on the fly. (Figure 50)

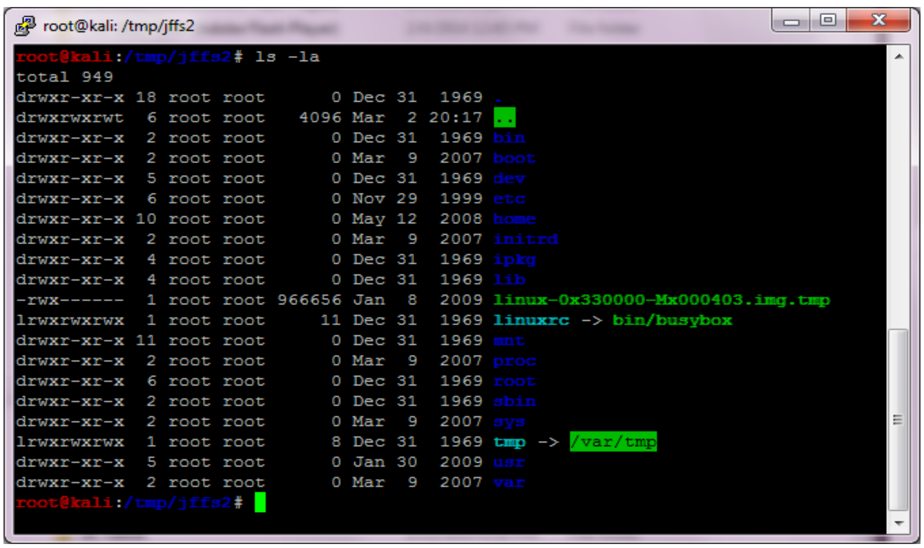


Figure 50 Mounted JFFS2 file system

Low level JFFS2 analysis

JFFS2 is a journaling file system. A journaling file system is one that keeps logs of changes to the file system. This is very useful for embedded systems as it means they can be turned off any time without any proper shutdown process without breaking the whole file system. You might lose some changes, but the integrity of other major file systems is not affected. Journaling makes the file system more resistant to corruption due to sudden shutdown. The fact that JFFS2 keeps file system changes can be very useful from a forensic point of view.

To automate the process of analyzing the JFFS2 file system, I created the DumpJFFS2 project that can handle the low level nature of the JFFS2 file system file. Using this tool, you can dump out the whole file system without mounting. Based on the source code, you can even create your own custom logic to analyze the low level JFFS2 file system.

Modifying data and reattaching

The good thing with this JFFS2 mounting technique is that you have write access on the file system. You can try to modify and patch any files on the system and take the JFFS2 raw image from the MTD device. The dumped image is a valid JFFS2 file that can be mounted again. You can program the NAND flash with this modified JFFS2 data.

```
root@test:~# dd if=/dev/mtdblock0 of=mtdblock0.dmp bs=512
131070+0 records in
131070+0 records out
67107840 bytes (67 MB) copied, 2.90779 s, 23.1 MB/s
```

Figure 51 Dumping mtdblock device raw image

Writing to NAND Flash

After you make changes to the JFFS2 file system image, you need to place the OOB data before writing to the Flash memory. The following command reconstructs a flat NAND Flash image from a memory image of the JFFS2 file system. It reads the *mtdblock0.dmp* file dumped from the MTD device and adds OOB data automatically, writing it to the *mtdblock0.oob.dmp* file. It calculates ECC for each page and adds the JFFS2 *erasemarker* for each block.

python DumpFlash.py -R -o mtdblock0.oob.dmp mtdblock0.dmp

Using this flat image, you can finally write it back to the original NAND Flash memory chip. With the NAND reader/writer connected to a USB port, run following command:

python FlashTool.py -w mtdblock.mod.oob.dmp -R 0x12820 0xffffffff

The *-s* option designates the start page number. The option 0x12820 designates the address of 0x12820 * (0x200 + 0x10) in this case (page size=0x100=512, spare column=0x10=16). The actual location it writes is 0x262C200. This is the location from where I extracted the JFFS2 image.

Figure 52 shows what this NAND Flashing process looks like.

```
root@test:~# python FlashTool.py -w mtdblock0.oob.dmp -R 0x12820 0xffffffff
Name:             NAND 64MiB 3,3V 8-bit
ID:               0x76
Page size:        0x200
OOB size:         0x10
Page count:       0x20000
Size:             0x40
Erase size:       0x4000
Options:          0
Address cycle:    4
Manufacturer:     Samsung
 Writing 0x1285b/0x20000 (6941 bytes/sec)
```

Figure 52 Writing the full image to NAND Flash

Re-soldering

After modifying raw data and writing it back to the Flash memory, it is time to re-solder the chip onto the target system. The re-soldering process is not much different from standard SMT soldering. Originally SMT was developed for automatic soldering of PCB components. So the chips are usually small and the pitch of the pins is also relatively small. This makes soldering them to the PCB manually challenging, but it is not extremely difficult when you get accustomed to it. There are many different methodologies developed by many hobbyists. The method I used was just placing the chip on the pin location and heating the pins using the soldering iron. This lets the solder residue (Figure 53) left from the previous de-soldering process melt again. The chip is soldered again using this same solder. Sometimes adding a small amount of solder paste onto each pin helps the chip to reattach to the board. If this method doesn't meet your requirements, you can remove any solder residue first and start with new solder or solder paste. Various detailed techniques can be found on the Internet.

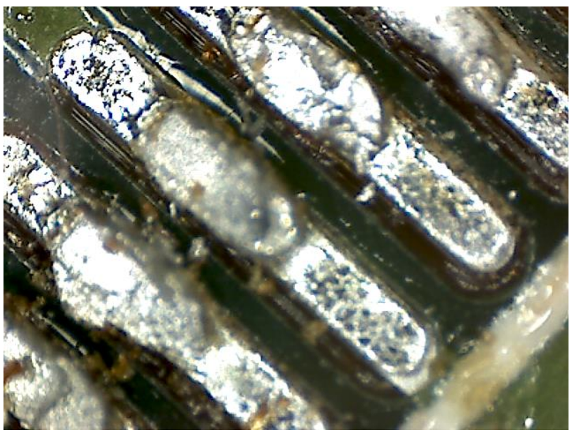


Figure 53 Solder residue

There are many pitfalls with SMT soldering and one of the big issues is bridging. The pitch for the NAND flash TSOP48 model is 0.5 mm, which is extremely small. This means the solder can easily go over multple pins and create shorts. (Figure 54) - be careful to ensure this doesn't happen.

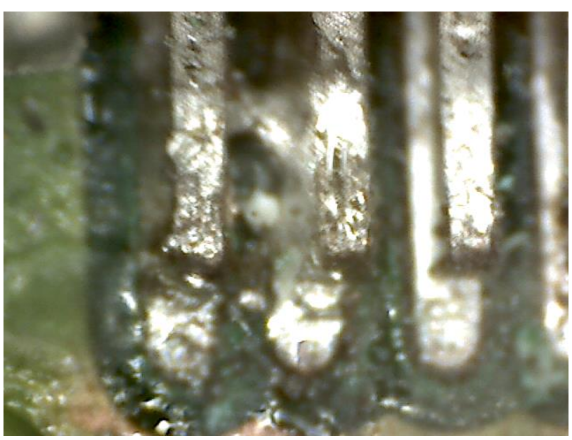


Figure 54 Bridge

One of the other big problems with re-soldering is possible damage to the board. (Figure 55) With the de-soldering process, excessive heat is applied and it can damage the PCB board. With this in mind, you should be extra careful when you re-solder the chips. One good thing with Flash memory, is that many pins are not actually used. If the damaged patterns are not used, then the chips will still operate normally. You should check with the chip datasheet to see if any damaged patterns are actually used by the chip.

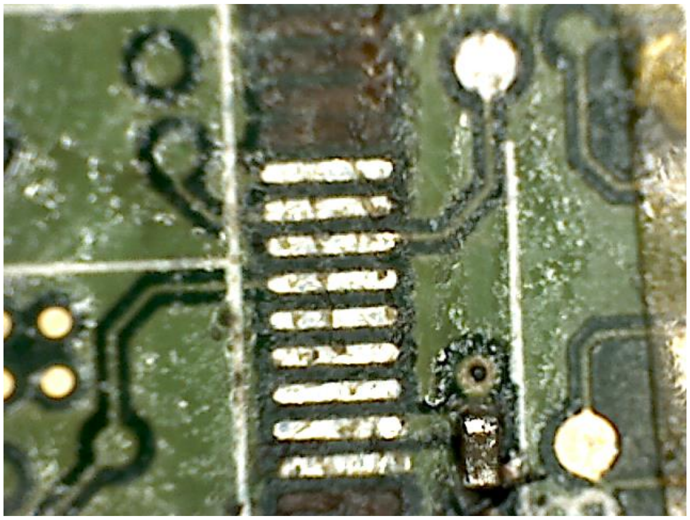


Figure 55 Damaged circuit board

For my case, the circuit for pin 48 was damaged but luckily the pin is never used by the chip. So everything worked fine after re-soldering. The truth is that the pins that are not used have a greater tendency to be damaged as they are not connected to any circuitry on the system. They are just glued to the board without any connection to other components and it makes them more vulnerable to heat.

Tools

FlashTool – Python Implmentation of Flash reader/writer software

- https://github.com/ohjeongwook/DumpFlash/blob/master/FlashTool.py
    - Write support
    - Fast sequential row read mode support
    - More experimental code coming.

Enhanced NandTool (forked from original NandTool): NandTool with writing support

- https://github.com/ohjeongwook/NANDReader_FTDI
    - Write support

DumpFlash.py: Flash image manipulation tool (ECC, Bad block check)

- https://github.com/ohjeongwook/DumpFlash/blob/master/DumpFlash.py

DumpJFFS2.py: JFFS2 parsing tool

- https://github.com/ohjeongwook/DumpFlash/blob/master/DumpJFFS2.py

Conclusion

Interacting directly with Flash memory is useful when JTAG can't be used. This situation is becoming more and more likely these days as some vendors obfuscate or remove JTAG interfaces to protect their intellectual property. As a security researcher, you have a need for accessing the internals of embedded systems. By directly interacting with a low level Flash memory interface, you have the benefit of accessing data that can't otherwise be retrieved. The entire process can be time consuming, but the benefit is clear. The de-soldering method is referred to as a destructive method in reverse engineering hardware. But, it is still possible to re-solder the chip to the system using SMT soldering methods. There is a higher chance of damaging the circuit board than when working on a fresh, new PCB board, but the chance for success is still high enough. Also, there are many factors to consider when extracting, modifying and reconstructing a bare metal image with your modification like ECC, bad blocks and JFFS2 *erasemarkers*. You might try to modify code from many places like boot loaders, the kernel and the JFFS2 root image. Thus, you can start on your way to researching embedded systems, even when JTAG connections are not feasible.

Lastly, many USB thumb drives and other devices also use NAND Flash memory for storage and they don't have any JTAG points at all by design. Even though the data format saved on the memory will be totally different from what is presented here, it could be beneficial to perform forensic analysis on these devices using this method.

References

1. **[Online] http://www.forbes.com/fdc/welcome_mjx.shtml.**

2. [Online] http://www2.electronicproducts.com/NAND_vs_NOR_flash_technology-article-FEBMSY1-feb2002-html.aspx.

3. [Online] http://www.ftdichip.com/Products/ICs/FT2232H.htm.

4. [Online] http://www.techradar.com/us/news/computing/how-error-detection-and-correction-works-1080736.