Reverse Engineering Flash Memory for Fun and Benefit

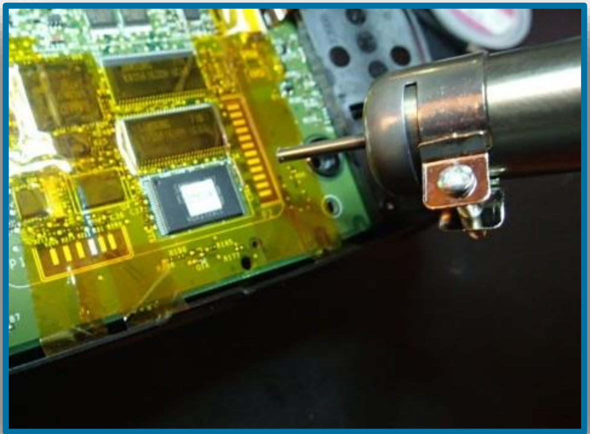
Jeong Wook (Matt) Oh / 2014

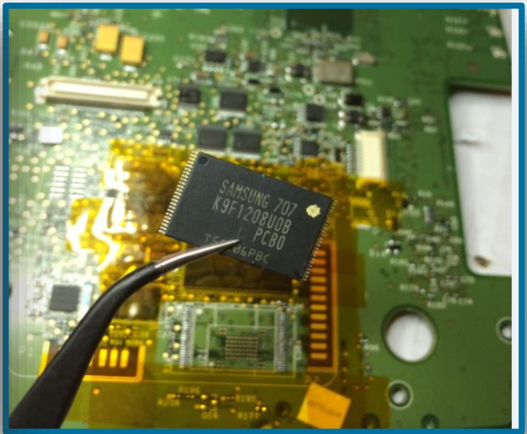
oh@hp.com

oh.jeongwook@gmail.com

De-soldering

De-soldering





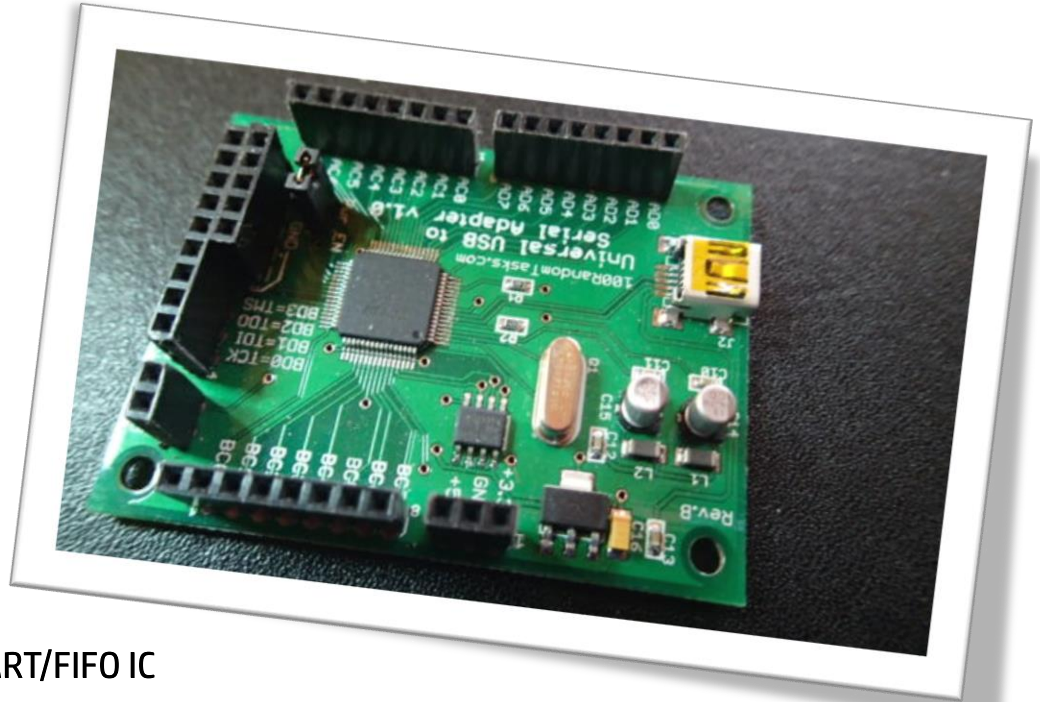
Use an SMT Rework station with an hot air blower:

- The solder alloy melts at around 180 and 190°C (360 and 370°F), but I recommend setting the temperature slightly higher

Note: Before applying high heat to the chip, put insulating tape around the target area

FTDI FT2232H & NAND Flash Memory

FTDI FT2232H breakout board



A chip for USB communication

- Provides USB 2.0 Hi-Speed (480Mb/s) to UART/FIFO IC

Note: Put female pin headers on each port extension

MCU Host Bus Emulation Mode

FTDI FT2232H supports multiple modes

- Use ‘MCU Host Bus Emulation Mode’ for this case

The FTDI chip emulates an 8048/8051 MCU host bus

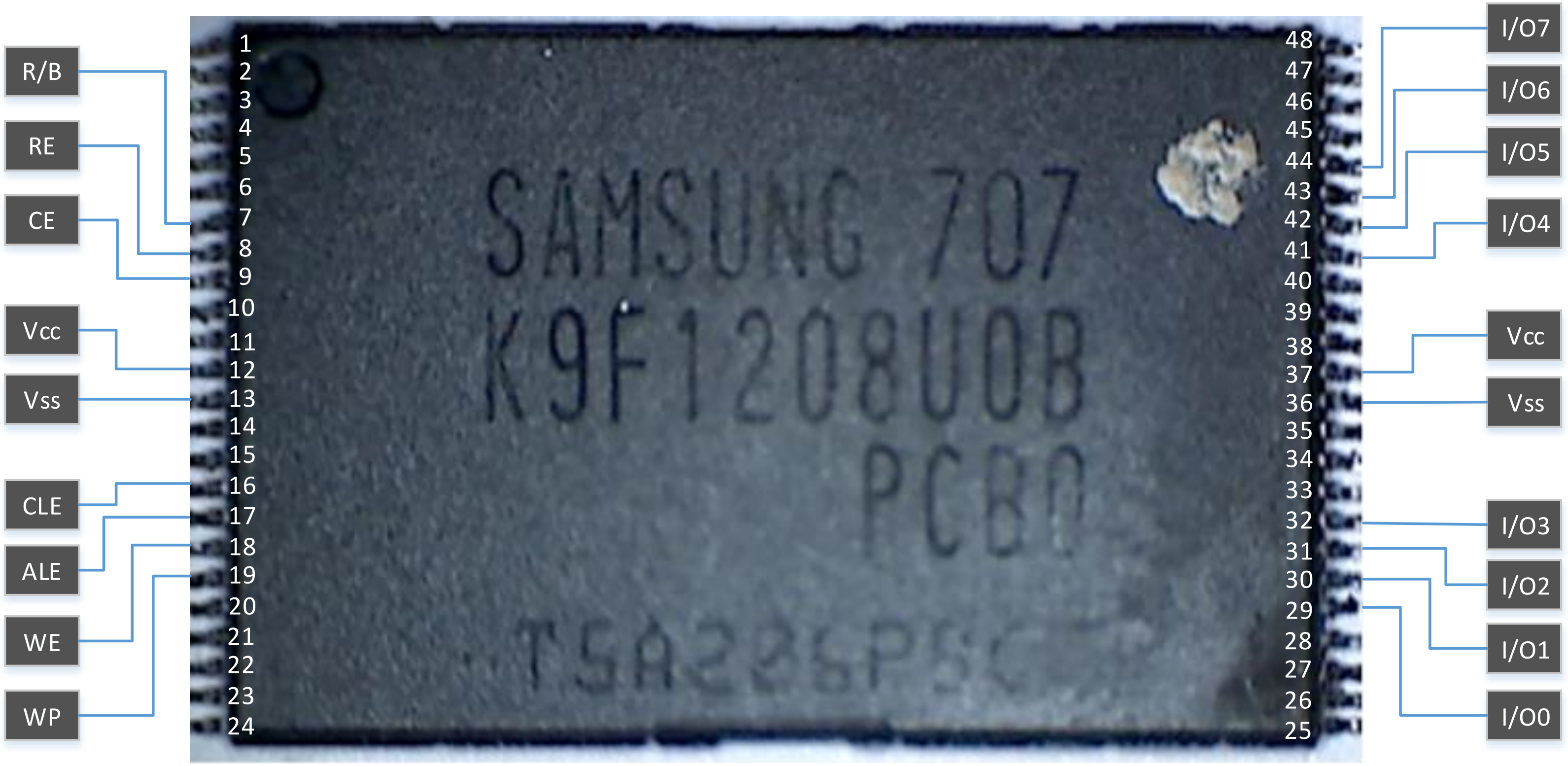
FT2232H Commands

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)

By sending commands and retrieving results, the software reads or writes bits through I/O lines.

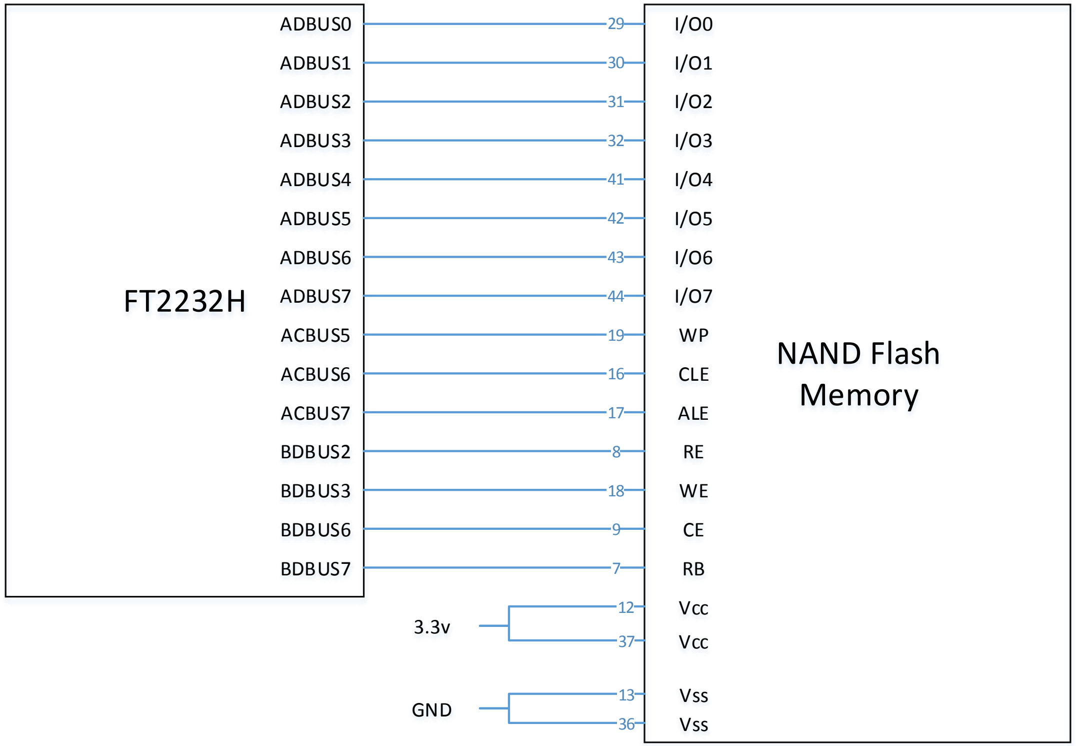
- See FTDI's note for more detail

NAND Flash memory pins and names



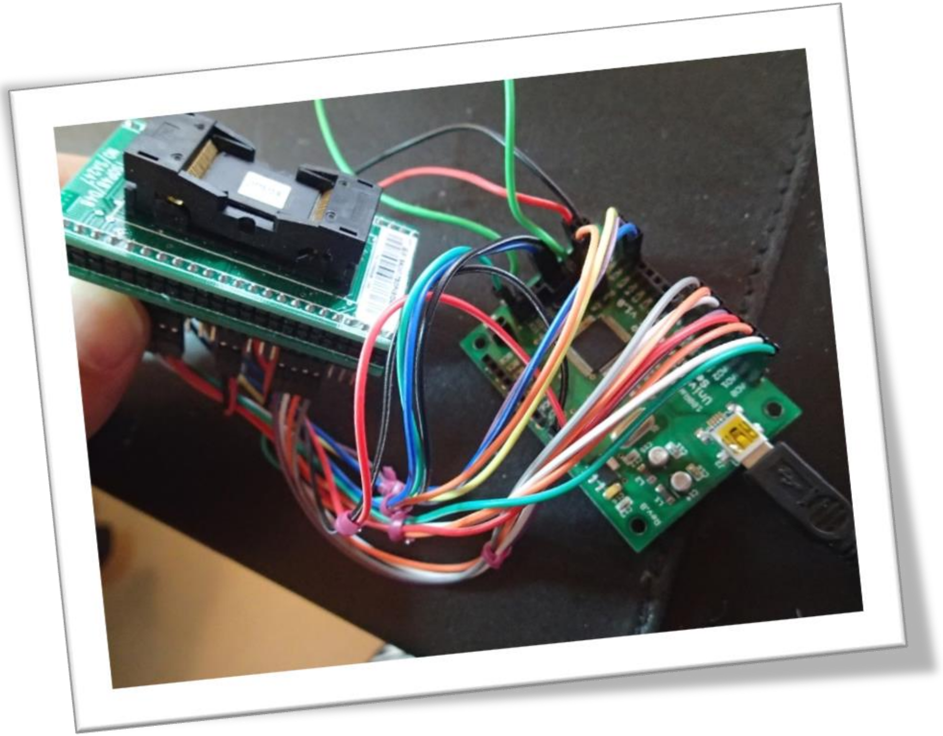
Connection between FT2232H and NAND Flash Memory

The connections are mostly based on the information from Sprites Mod , but there is a slight modification between BDBUS6 and CE (9) connection.

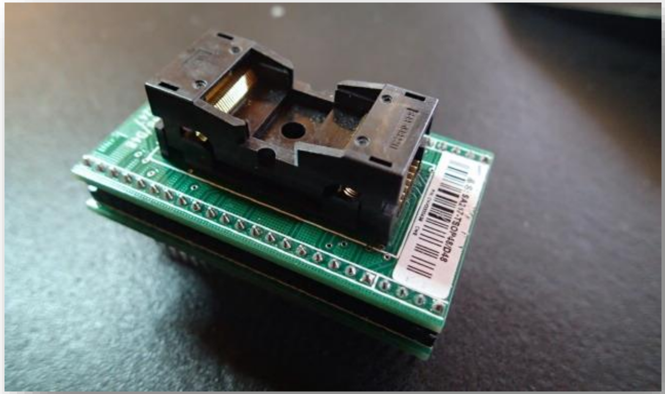


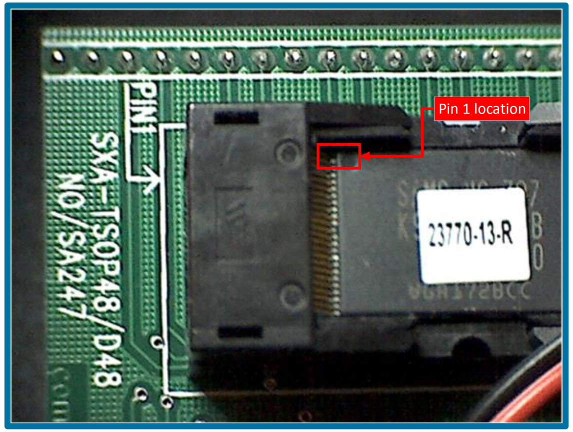
NAND Flash reader/writer

You need an FTDI FT2232H breakout board, a USB cable, a TSOP48 socket and wires



TSOP48 socket





Place your NAND Flash chip inside the TSOP48 socket:

- This socket is very useful
- Use it to directly interact with the extended pins and avoid touching and possibly damaging any Flash memory chip pins

Data Lines

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	

Low byte

- 0x90,0x91,0x92,0x93 commands can be used to set values

Data Control Lines

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

High byte

- 0x91, 0x93 can be used to set values

I/O and Strobe Lines

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.

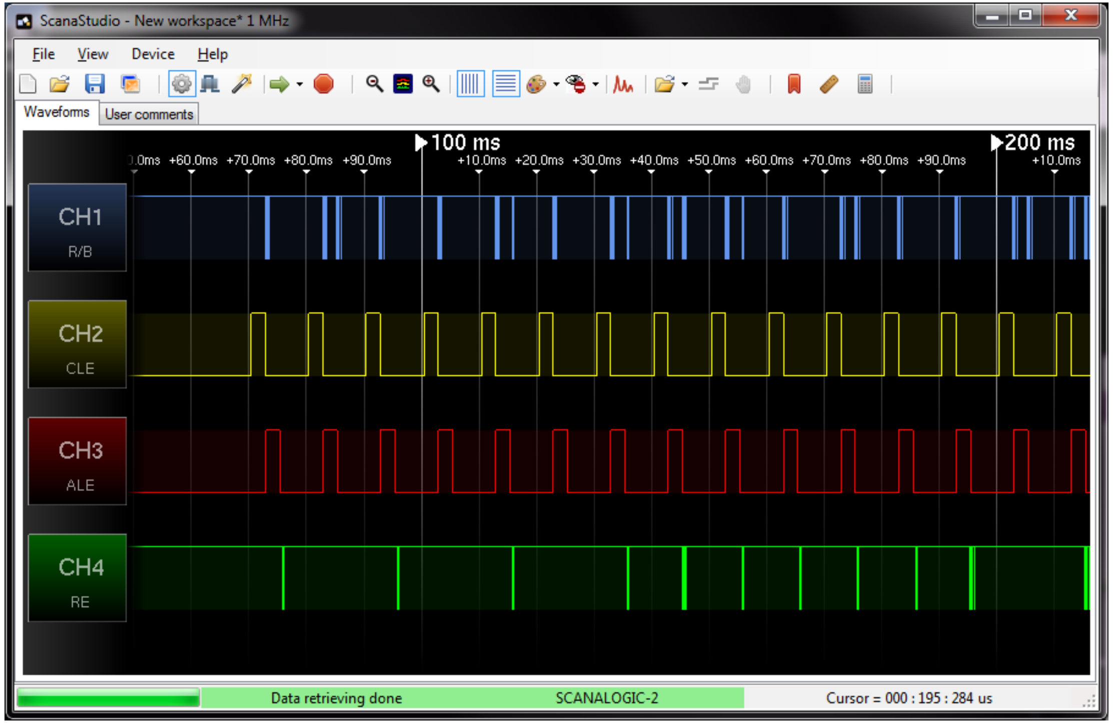
- BDBUS6 (I/O0), BDBUS7 (I/O1) is controlled by 0x83, 0x82 command
- RD#, WR# is connected to RE, WE pin on NAND Flash

Power Lines

	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

- Power lines

Read Operation Example



- CLE and ALE go high - 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

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

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	

There are more complicated commands available depending on the chipsets.

- The pins and other descriptions presented here are mostly focused on small block NAND Flash models (512 bytes of data with 16 bytes OOB)
- The model with a large block size uses a different set of commands, but the principle is the same

Read operation

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, the 50h command is used to retrieve the OOB of the page (spare C area)

Read operation

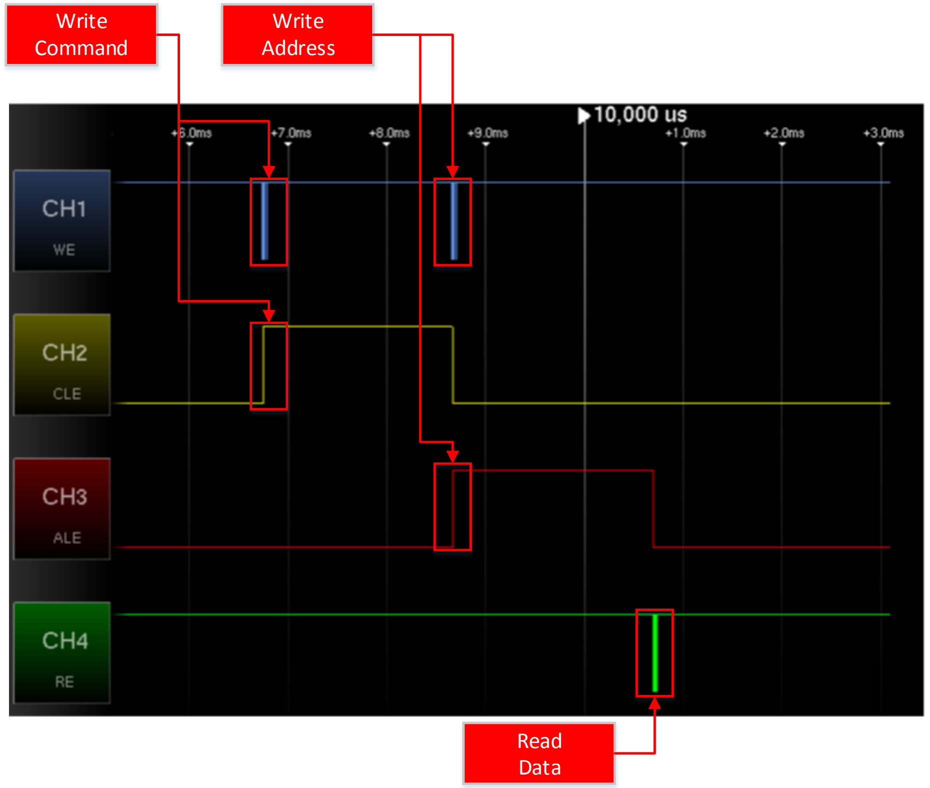
CLE	1	0		
ALE	0	1	0	
R/B	1 (Ready)		R/B=0 (busy)	1 (Ready)
RE	1			Falling for each bytes
WE	Rising for each bytes		1	
I/O0~7	00h/01h /50h	Start Address A0 – A7 A9 – A25		Data Output

- 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

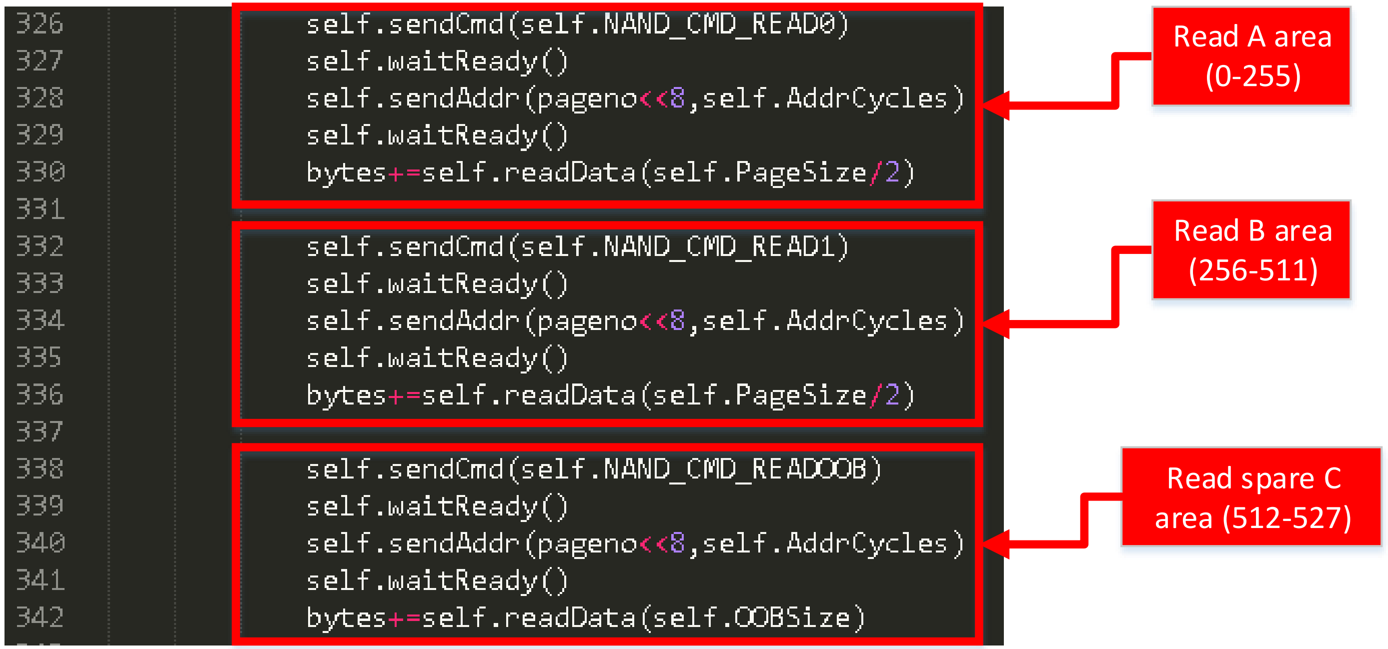
Reading data



1. First, the WE and CLE logic changes to send commands.
2. Next, the WE and ALE will change state to send addresses.
3. Finally, RE is used to signal reading of each byte.

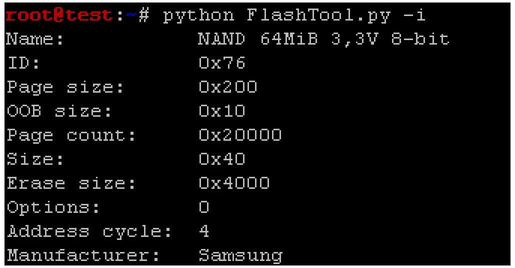
Reading a small block page

- NAND_CMD_READ0 (00h)
- NAND_CMD_READ1 (01h)
- NAND_CMD_READOOB (50h)



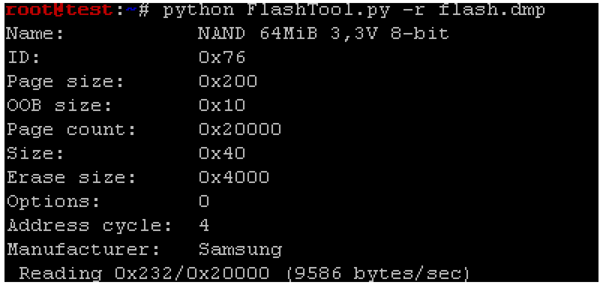
```
        self.sendCmd(self.NAND_CMD_READ0)
        self.waitReady()
        self.sendAddr(pageno<<8,self.AddrCycles)
        self.waitReady()
        bytes+=self.readData(self.PageSize/2)

        self.sendCmd(self.NAND_CMD_READ1)
        self.waitReady()
        self.sendAddr(pageno<<8,self.AddrCycles)
        self.waitReady()
        bytes+=self.readData(self.PageSize/2)

        self.sendCmd(self.NAND_CMD_READOOB)
        self.waitReady()
        self.sendAddr(pageno<<8,self.AddrCycles)
        self.waitReady()
        bytes+=self.readData(self.OOBSize)
```

Reading data



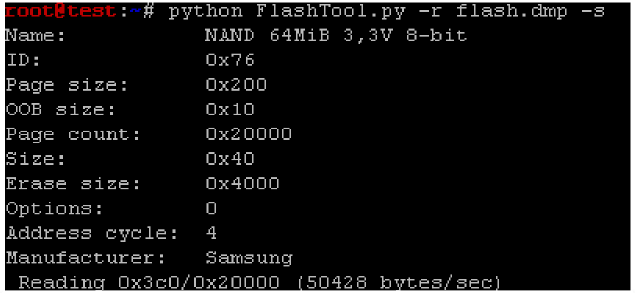
```
root@test:~# python FlashTool.py -i
Name:            NAND 64MiB 3,3V 8-bit
ID:              0x76
Page size:       0x200
OOB size:        0x10
Page count:      0x20000
Size:            0x40
Erase size:      0x4000
Options:         0
Address cycle:   4
Manufacturer:    Samsung
```

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.



```
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)
```

You can also read the raw data with the –r option. It takes some time to retrieve all the data depending on the size of the memory.



```
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)
```

FlashTool supports sequential row read mode. You can specify the –s option and it will use the mode and increase reading performance. The speed of reading is 5-6 times faster than normal page-by-page mode.

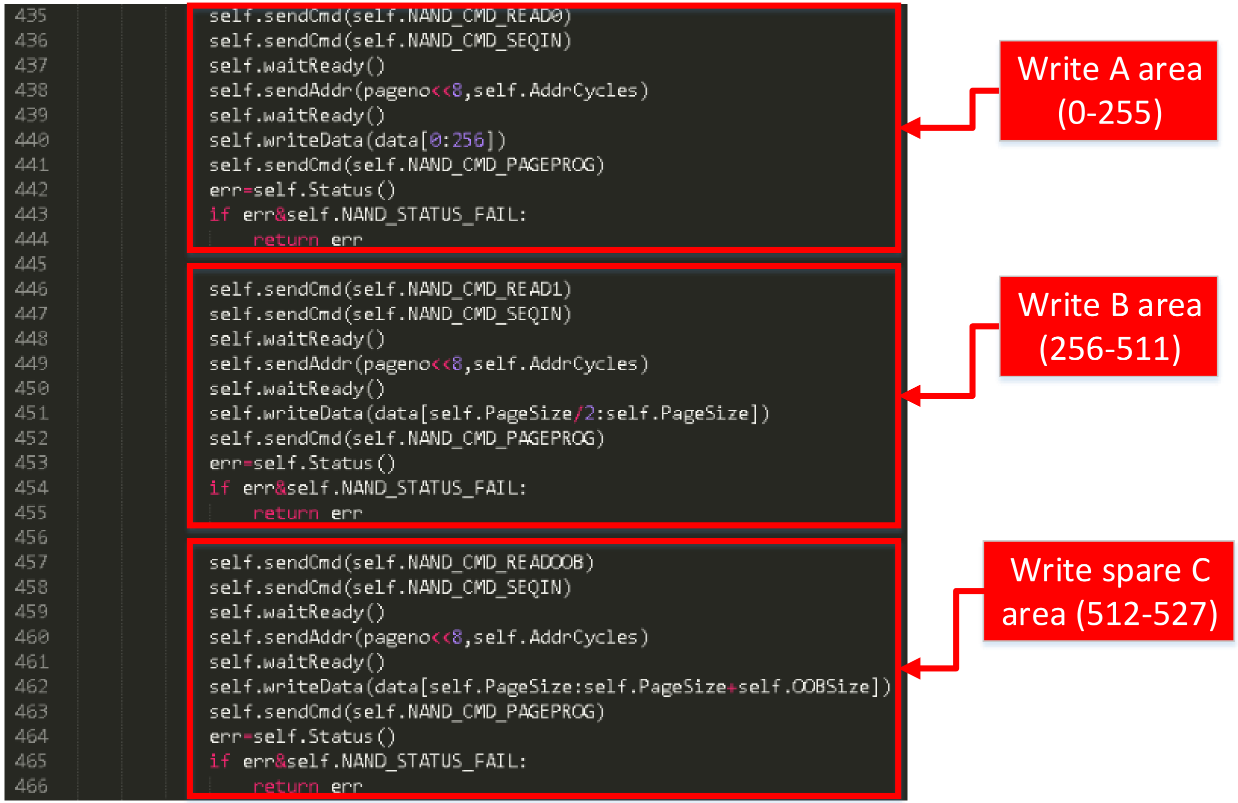
Write operation pin states

CLE	1	0		1			
ALE	0	1	0				
R/B	1 (Ready)				R/B=0 (busy)	1 (Ready)	
RE	1						Falling
WE	Rising for each bytes				1	Rising	1
I/O0~7	80h	Address Input A0 – A7 A9 – A25	Page + OOB data	10h		70h	I/O0=status

The writing operation is done through sequence-in command (80h) and program command (10h):

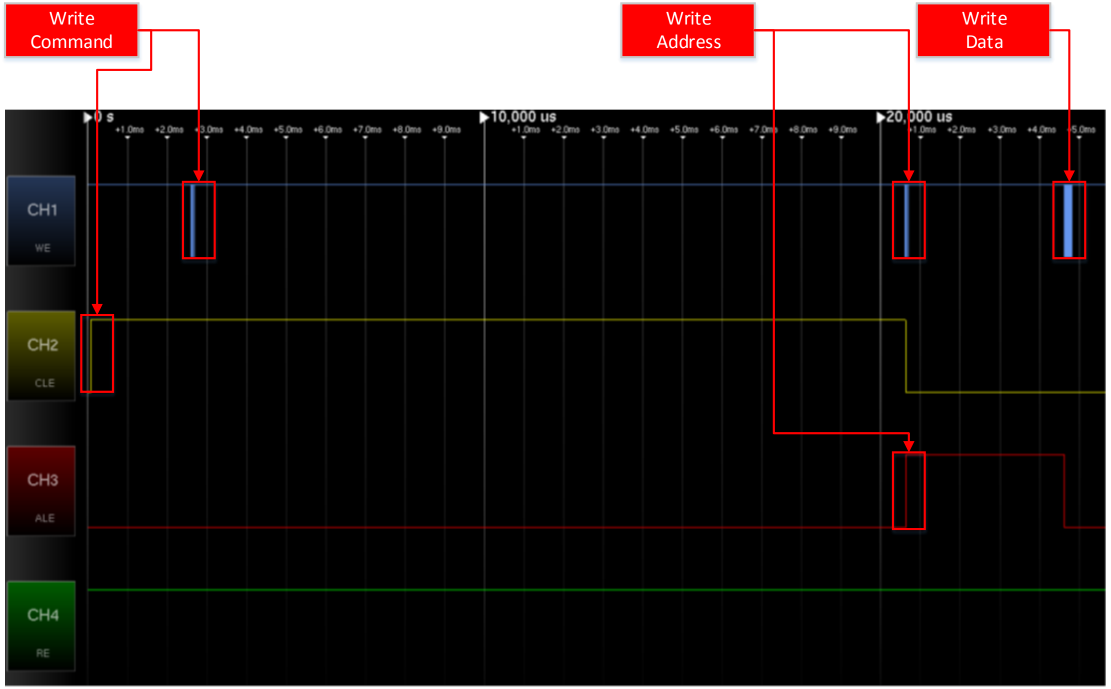
- The read status command (70h) is used to retrieve the result of the write operation
- If I/O0 is 0, the operation was successful

Writing a small block page with spare C area



```
self.sendCmd(self.NAND_CMD_READ0)
self.sendCmd(self.NAND_CMD_SEQIN)
self.waitReady()
self.sendAddr(pageno<<8,self.AddrCycles)
self.waitReady()
self.writeData(data[0:256])
self.sendCmd(self.NAND_CMD_PAGEPROG)
err=self.Status()
if err&self.NAND_STATUS_FAIL:
    return err

self.sendCmd(self.NAND_CMD_READ1)
self.sendCmd(self.NAND_CMD_SEQIN)
self.waitReady()
self.sendAddr(pageno<<8,self.AddrCycles)
self.waitReady()
self.writeData(data[self.PageSize/2:self.PageSize])
self.sendCmd(self.NAND_CMD_PAGEPROG)
err=self.Status()
if err&self.NAND_STATUS_FAIL:
    return err

self.sendCmd(self.NAND_CMD_READOOB)
self.sendCmd(self.NAND_CMD_SEQIN)
self.waitReady()
self.sendAddr(pageno<<8,self.AddrCycles)
self.waitReady()
self.writeData(data[self.PageSize:self.PageSize+self.OOBSize])
self.sendCmd(self.NAND_CMD_PAGEPROG)
err=self.Status()
if err&self.NAND_STATUS_FAIL:
    return err
```

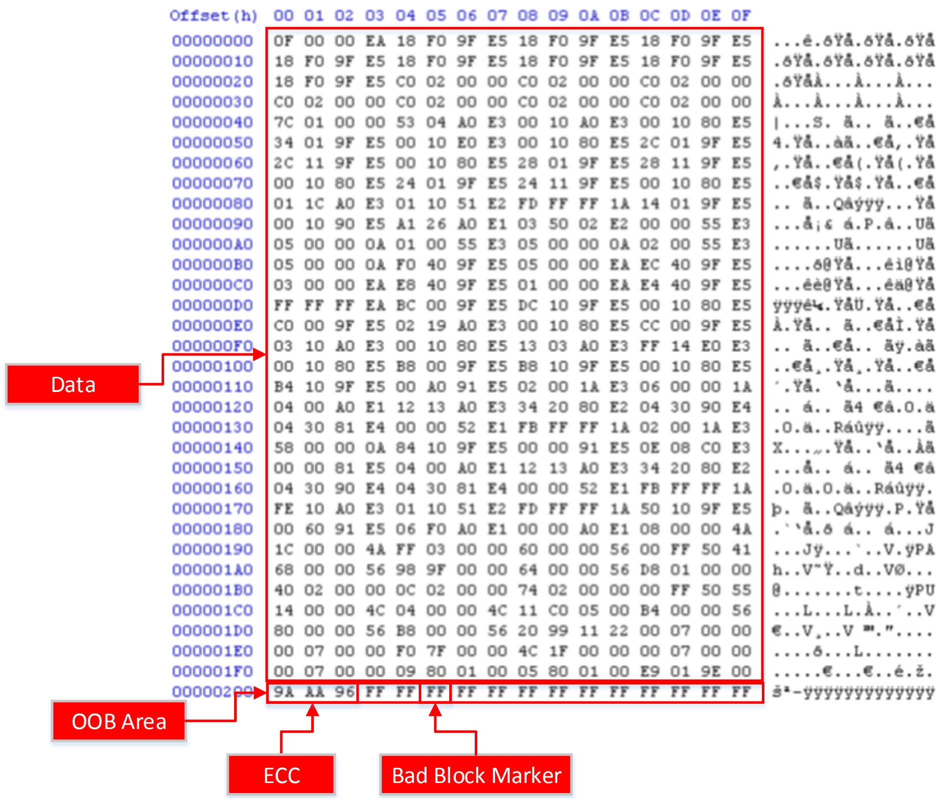
Writing Data



After command and address are sent, WE fluctuates repeatedly to send bytes.

Working with a bare metal image

Page+OOB



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  0F 00 00 EA 18 F0 9F E5 18 F0 9F E5 18 F0 9F E5  ...ê.ðŸå.ðŸå.ðŸå
00000010  18 F0 9F E5 18 F0 9F E5 18 F0 9F E5 18 F0 9F E5  .ðŸå.ðŸå.ðŸå.ðŸå
00000020  18 F0 9F E5 C0 02 00 00 C0 02 00 00 C0 02 00 00  .ðŸåÀ...À...À...
00000030  C0 02 00 00 C0 02 00 00 C0 02 00 00 C0 02 00 00  À...À...À...À...
00000040  7C 01 00 00 53 04 A0 E3 00 10 A0 E3 00 10 80 E5  |...S. ã.. ã..€å
00000050  34 01 9F E5 00 10 E0 E3 00 10 80 E5 2C 01 9F E5  4.Ÿå..àã..€å,.Ÿå
00000060  2C 11 9F E5 00 10 80 E5 28 01 9F E5 28 11 9F E5  ,.Ÿå..€å(.Ÿå(.Ÿå
00000070  00 10 80 E5 24 01 9F E5 24 11 9F E5 00 10 80 E5  ..€å$.Ÿå$.Ÿå..€å
00000080  01 1C A0 E3 01 10 51 E2 FD FF FF 1A 14 01 9F E5  .. ã..Qâýÿÿ...Ÿå
00000090  00 10 90 E5 A1 26 A0 E1 03 50 02 E2 00 00 55 E3  ...å¡& á.P.â..Uã
000000A0  05 00 00 0A 01 00 55 E3 05 00 00 0A 02 00 55 E3  ......Uã......Uã
000000B0  05 00 00 0A F0 40 9F E5 05 00 00 EA EC 40 9F E5  ....ð@Ÿå...êì@Ÿå
000000C0  03 00 00 EA E8 40 9F E5 01 00 00 EA E4 40 9F E5  ...êè@Ÿå...êä@Ÿå
000000D0  FF FF FF EA BC 00 9F E5 DC 10 9F E5 00 10 80 E5  ÿÿÿê¼.ŸåÜ.Ÿå..€å
000000E0  C0 00 9F E5 02 19 A0 E3 00 10 80 E5 CC 00 9F E5  À.Ÿå.. ã..€åÌ.Ÿå
000000F0  03 10 A0 E3 00 10 80 E5 13 03 A0 E3 FF 14 E0 E3  .. ã..€å.. ãÿ.àã
00000100  00 10 80 E5 B8 00 9F E5 B8 10 9F E5 00 10 80 E5  ..€å¸.Ÿå¸.Ÿå..€å
00000110  B4 10 9F E5 00 A0 91 E5 02 00 1A E3 06 00 00 1A  ´.Ÿå. 'å...ã....
00000120  04 00 A0 E1 12 13 A0 E3 34 20 80 E2 04 30 90 E4  .. á.. ã4 €â.0.ä
00000130  04 30 81 E4 00 00 52 E1 FB FF FF 1A 02 00 1A E3  .0.ä..Ráûÿÿ....ã
00000140  58 00 00 0A 84 10 9F E5 00 00 91 E5 0E 08 C0 E3  X...„.Ÿå..'å..Àã
00000150  00 00 81 E5 04 00 A0 E1 12 13 A0 E3 34 20 80 E2  ...å.. á.. ã4 €â
00000160  04 30 90 E4 04 30 81 E4 00 00 52 E1 FB FF FF 1A  .0.ä.0.ä..Ráûÿÿ.
00000170  FE 10 A0 E3 01 10 51 E2 FD FF FF 1A 50 10 9F E5  þ. ã..Qâýÿÿ.P.Ÿå
00000180  00 60 91 E5 06 F0 A0 E1 00 00 A0 E1 08 00 00 4A  .`'å.ð á.. á...J
00000190  1C 00 00 4A FF 03 00 00 60 00 00 56 00 FF 50 41  ...Jÿ...`..V.ÿPA
000001A0  68 00 00 56 98 9F 00 00 64 00 00 56 D8 01 00 00  h..V˜Ÿ..d..VØ...
000001B0  40 02 00 00 0C 02 00 00 74 02 00 00 00 FF 50 55  @.......t....ÿPU
000001C0  14 00 00 4C 04 00 00 4C 11 C0 05 00 B4 00 00 56  ...L...L.À..´..V
000001D0  80 00 00 56 B8 00 00 56 20 99 11 22 00 07 00 00  €..V¸..V ™.".... 
000001E0  00 07 00 00 F0 7F 00 00 4C 1F 00 00 00 07 00 00  ....ð...L.......
000001F0  00 07 00 00 09 80 01 00 05 80 01 00 E9 01 9E 00  .....€...€..é.ž.
00000200  9A AA 96 FF FF FF FF FF FF FF FF FF FF FF FF FF  šª–ÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

ECC (Error Correction Code)

Failures occur with data on memory:

- A checksum can be useful to detect these errors

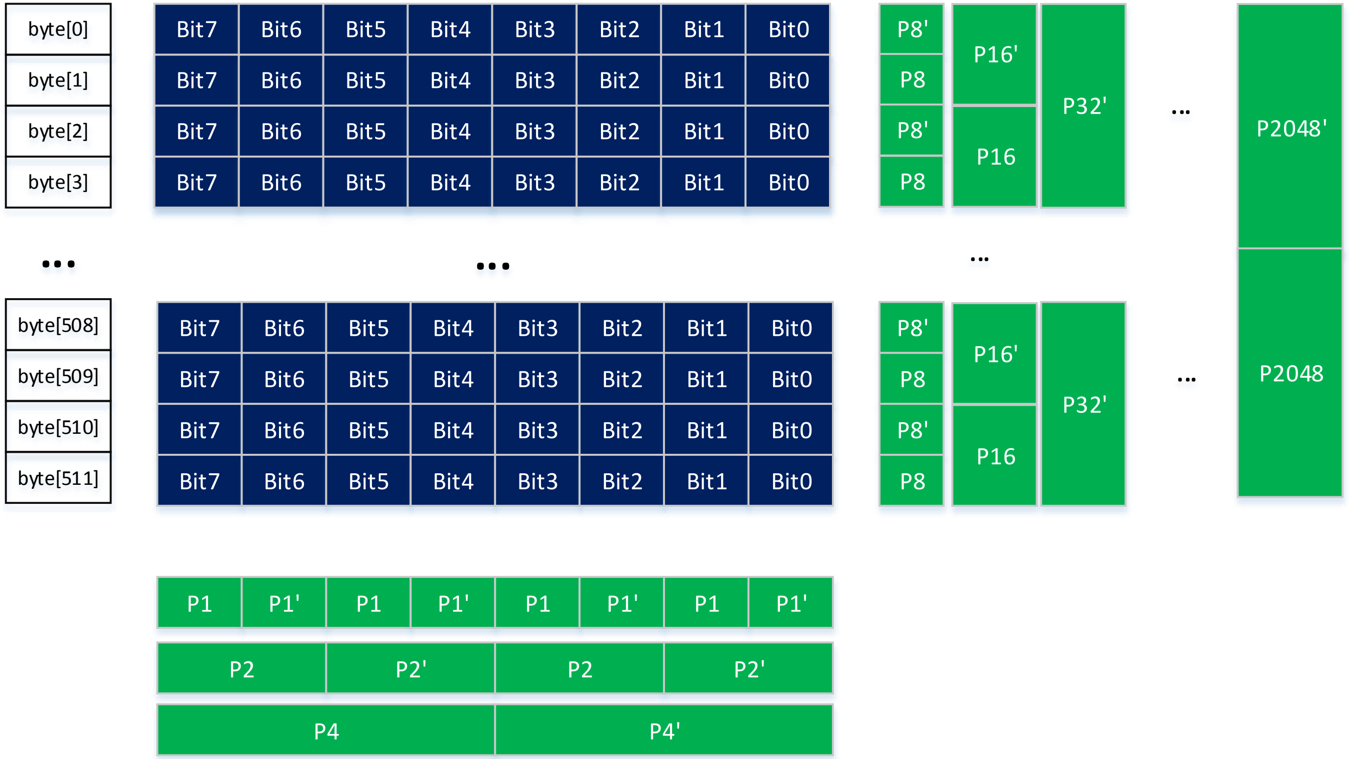
ECC (Error Correction Code) is a way to correct one bit of failure from a page:

- Besides detecting errors, ECC can correct them too if they are minor
- Uses the concept of Hamming code

Modern Flash memories use various ECC algorithms that have their roots in Hamming code:

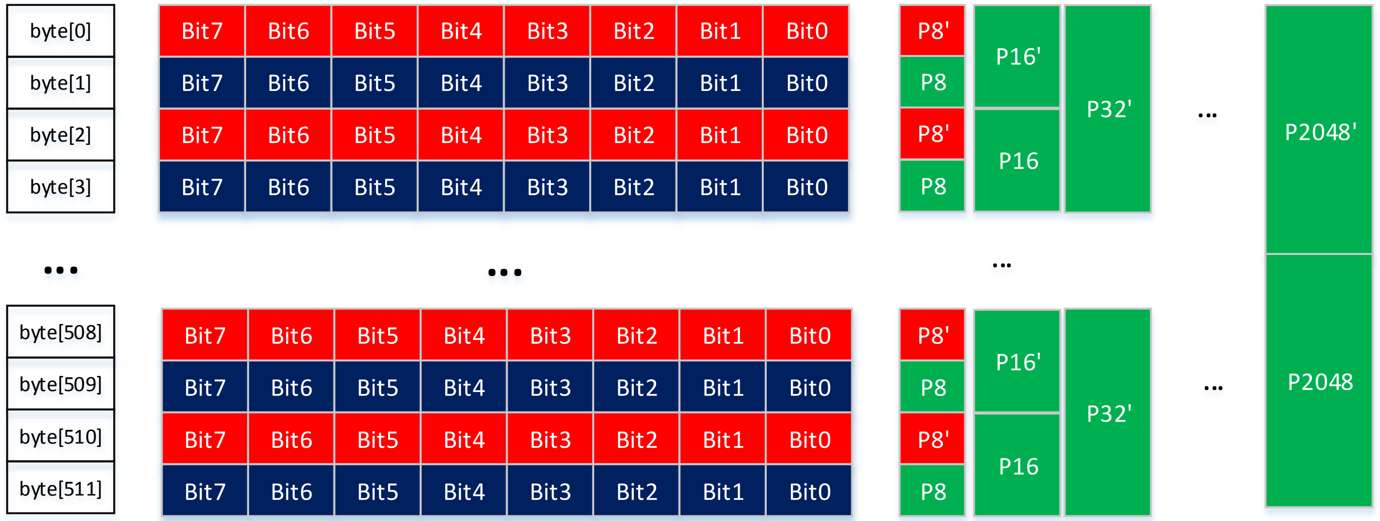
- Even similar chipsets from the same vendor may have slightly different ECC algorithms
- Differences are generally minor (tweaks of XOR or shifting orders or methods)
- You need to figure out the correct algorithm to verify the validity of each page and to generate ECC

ECC calculation table



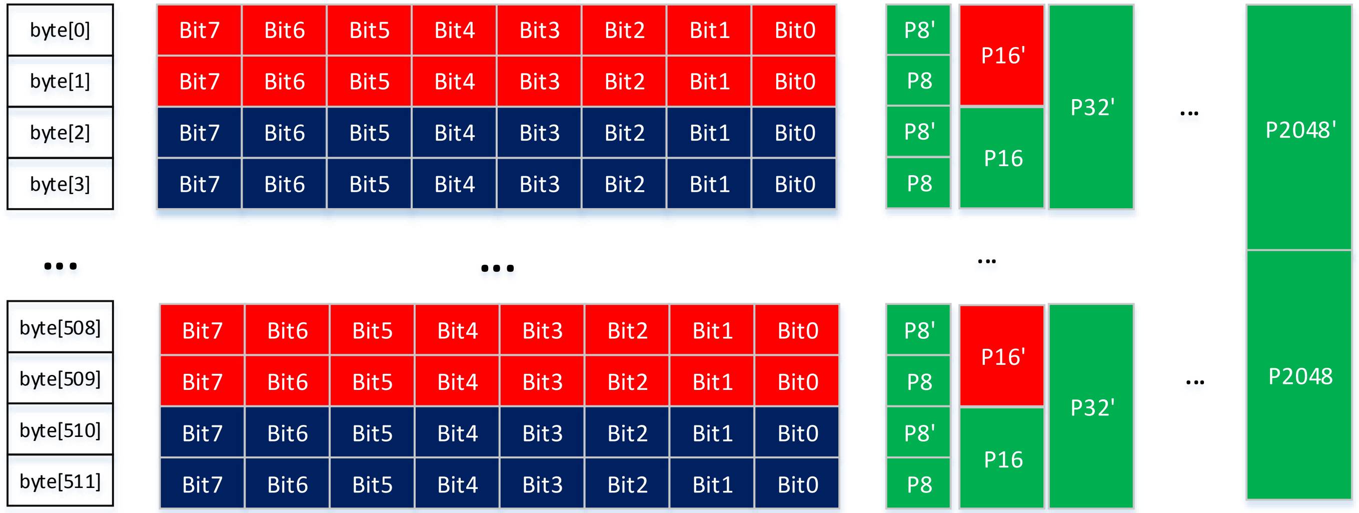
Representation of bits on a page with size of 512. Each bit is represented by a cell and each row is one byte. From this matrix, you can calculate various checksums across bits.

Example - P8' calculation



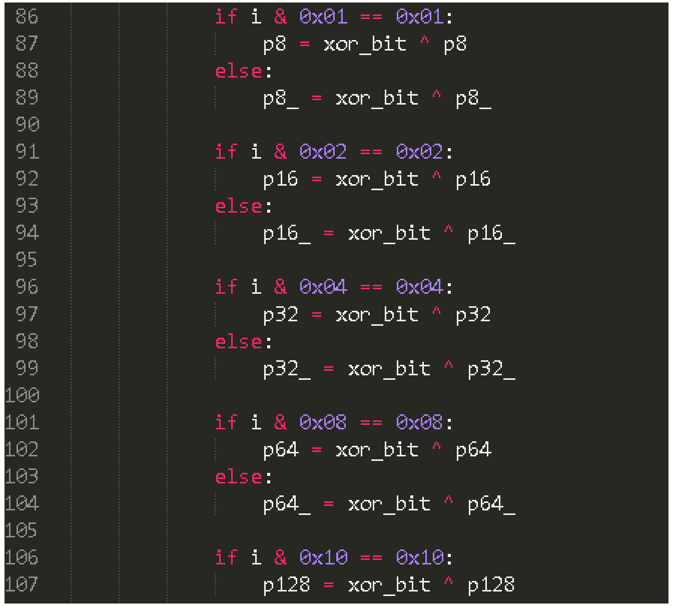
P8’ checksum is calculated by XOR-ing all the bits in red

Example - P16' calculation

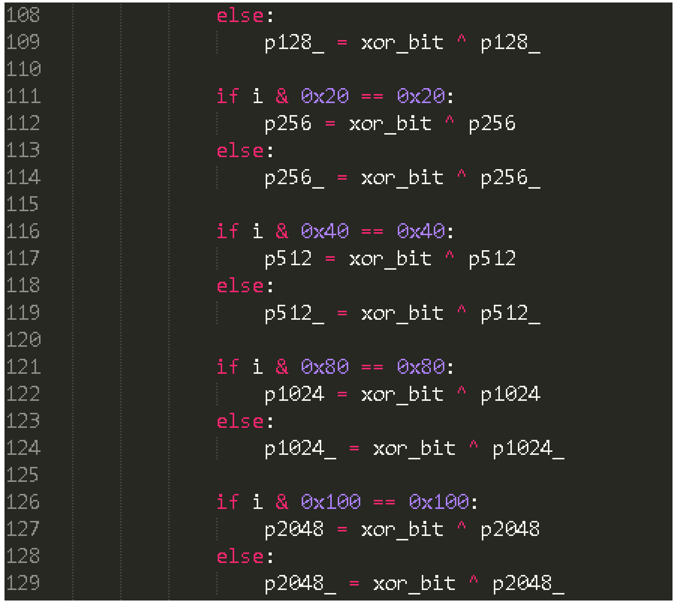


- Uses bits from byte[0], bytes[1], byte[4], byte[5] and so on until byte[508] and byte[509] for checksum calculation
- Other column checksums like P8, P16', P16, P32', P32, P2048' and P2048 are calculated in same manner

Code for calculating row checksums

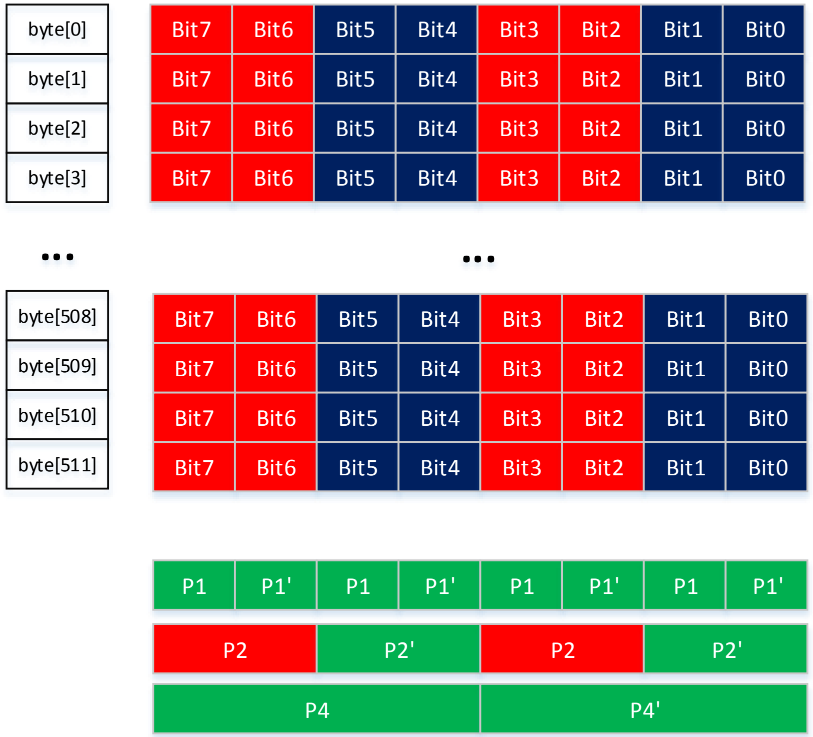


```
            if i & 0x01 == 0x01:
                p8 = xor_bit ^ p8
            else:
                p8_ = xor_bit ^ p8_

            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_
```

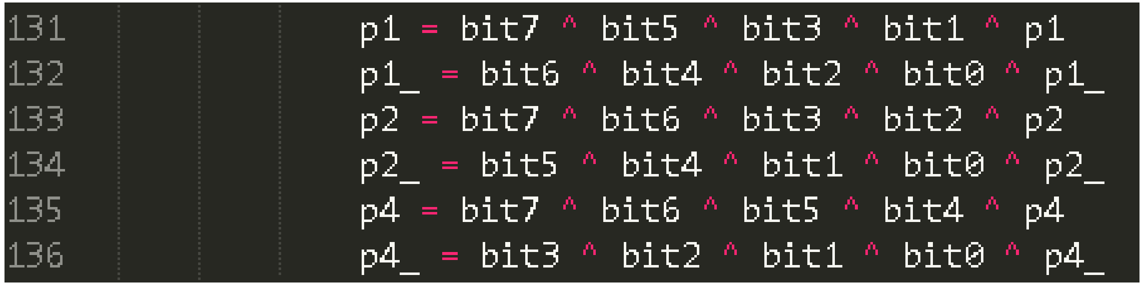
Example - P2 calculation



The column checksums are calculated over the same bit locations over all the bytes in the page.

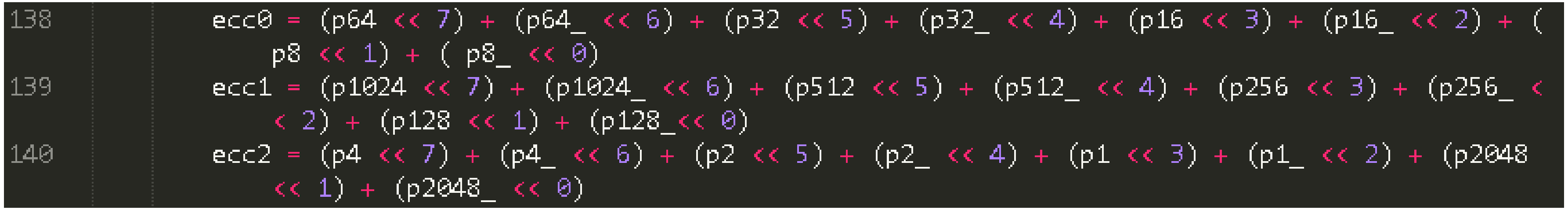
The picture shows how P2 can be calculated by taking bits 2,3,6,7 from each byte.

Row checksum calculation code



```
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_
```

ECC calculation code



```
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)
```

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

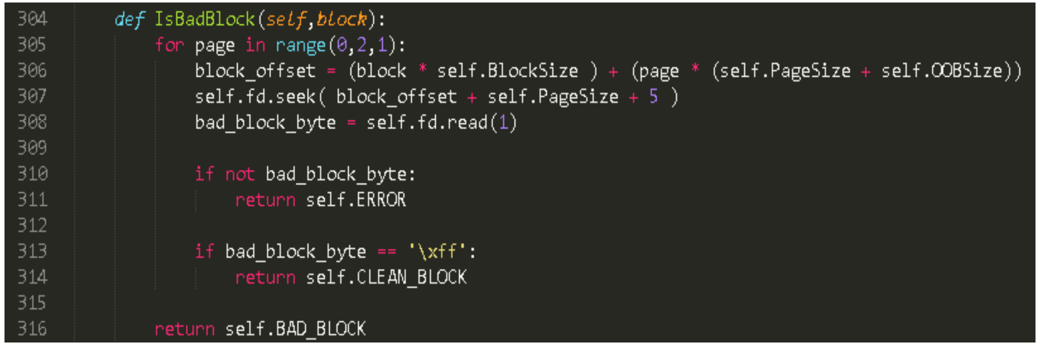
Bad blocks

- Bad blocks is a very generic concept that is also used with hard disk technology:
- With Flash memory, if errors are more than the ECC can handle, the entire block is marked as bad
- Bad blocks are isolated from other blocks and are no longer used
- According to the ONFI standard, the first or last pages are used for marking bad blocks

Example bad block check routine

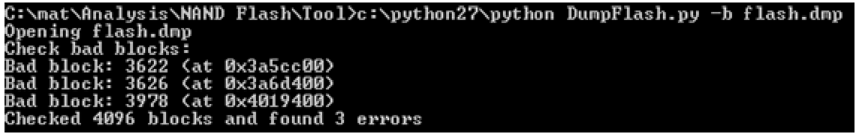
Some vendors use their own scheme for marking bad blocks:

- Ex) If the 6th 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 (Samsung and Micron).

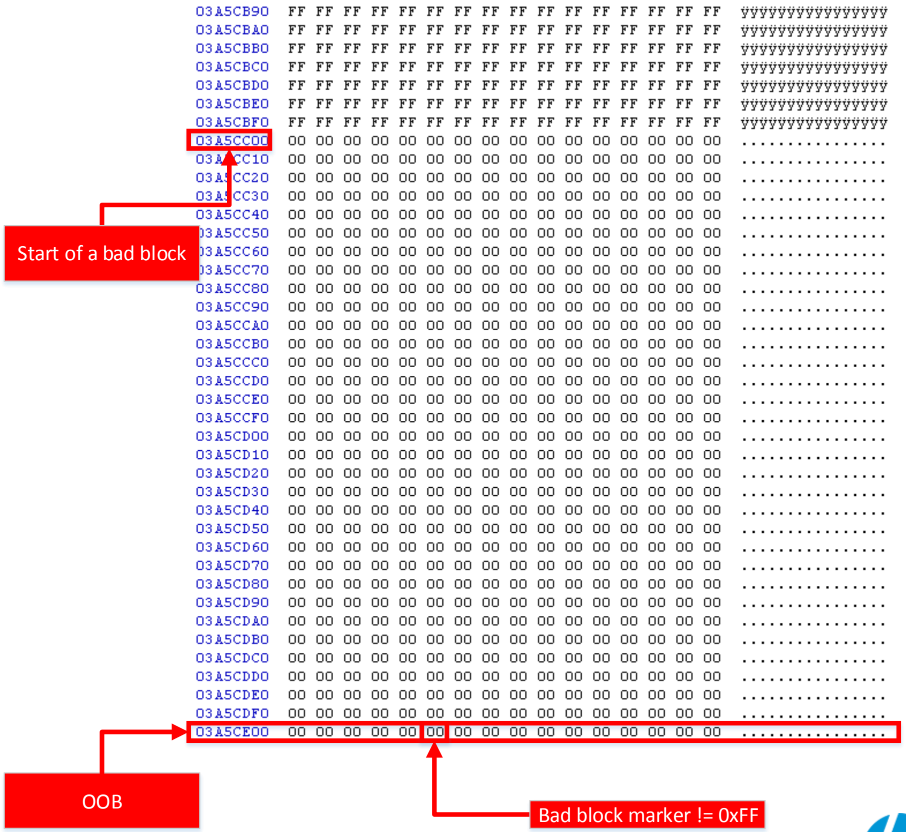


```
    def IsBadBlock(self,block):
        for page in range(0,2,1):
            block_offset = (block * self.BlockSize ) + (page * (self.PageSize + self.OOBSize))
            self.fd.seek( block_offset + self.PageSize + 5 )
            bad_block_byte = self.fd.read(1)

            if not bad_block_byte:
                return self.ERROR

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

        return self.BAD_BLOCK
```

How a bad block is marked



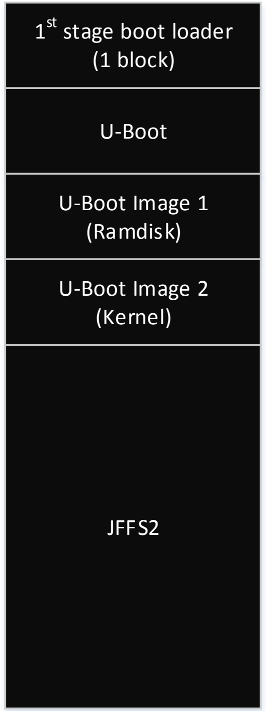
```
C:\mat\Analysis\NAND Flash\Tool>c:\python27\python DumpFlash.py -b flash.dmp
Opening flash.dmp
Check bad blocks:
Bad block: 3622 (at 0x3a5cc00)
Bad block: 3626 (at 0x3a6d400)
Bad block: 3978 (at 0x4019400)
Checked 4096 blocks and found 3 errors
```



```
03A5CB90  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
03A5CBA0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
03A5CBB0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
03A5CBC0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
03A5CBD0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
03A5CBE0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
03A5CBF0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
03A5CC00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC40  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC50  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CC90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CCA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CCB0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CCC0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CCD0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CCE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CCF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD40  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD50  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CD90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CDA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CDB0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CDC0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CDD0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CDE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CDF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03A5CE00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Reverse engineering Flash memory data

An example of Flash memory layout



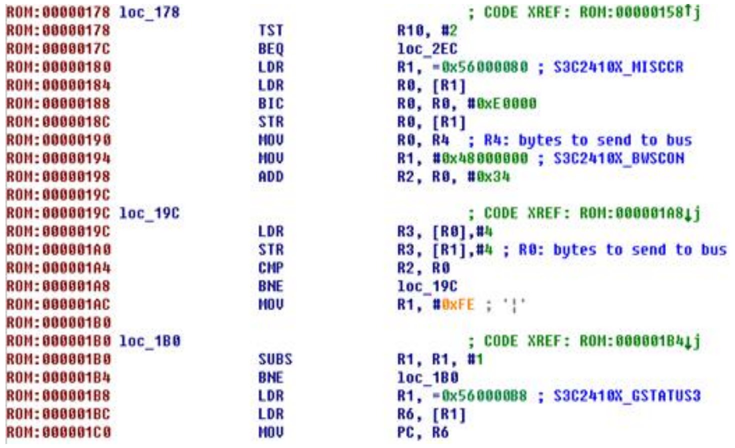
Usual structure of NAND Flash memory used for booting up embedded systems:

- The first block is always loaded to address 0x00000000
- U-Boot code and images follow
- When boot loading, code and U-Boot images are read only

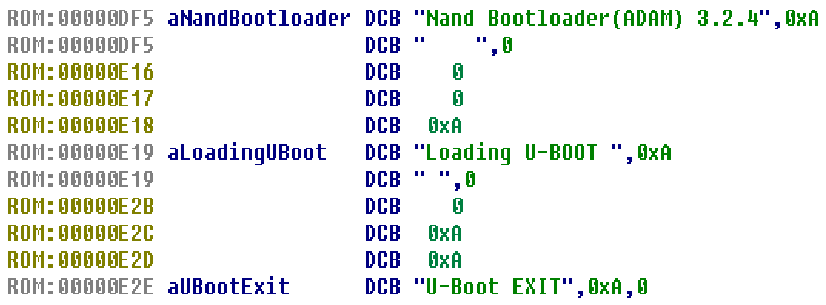
The JFFS2 file system is used for read and write:

- When a file is saved, it goes to JFFS2 file system

Low level initialization of the system



```
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
```



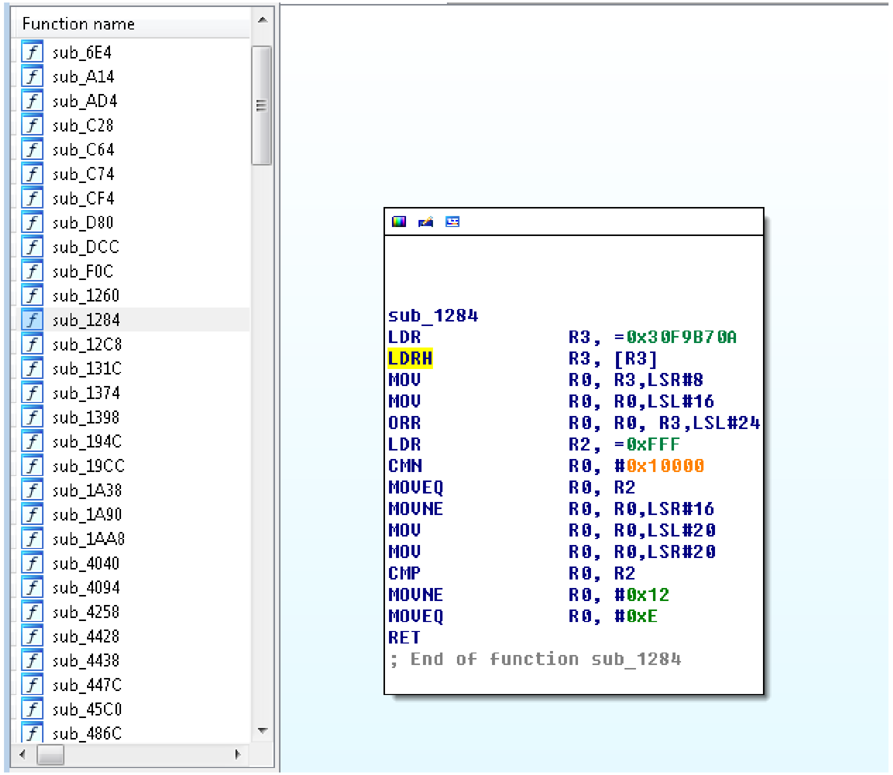
```
ROM:00000DF5 aNandBootloader DCB "Nand Bootloader(ADAM) 3.2.4",0xA
ROM:00000DF5                 DCB "    ",0
ROM:00000E16                 DCB    0
ROM:00000E17                 DCB    0
ROM:00000E18                 DCB  0xA
ROM:00000E19 aLoadingUBoot   DCB "Loading U-BOOT ",0xA
ROM:00000E19      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

This boot loader does low level initialization:

- It loads up the next level boot loader

Note: The image I worked on showed very interesting strings, like the name of the 1st boot loader and some log messages

U-boot boot code



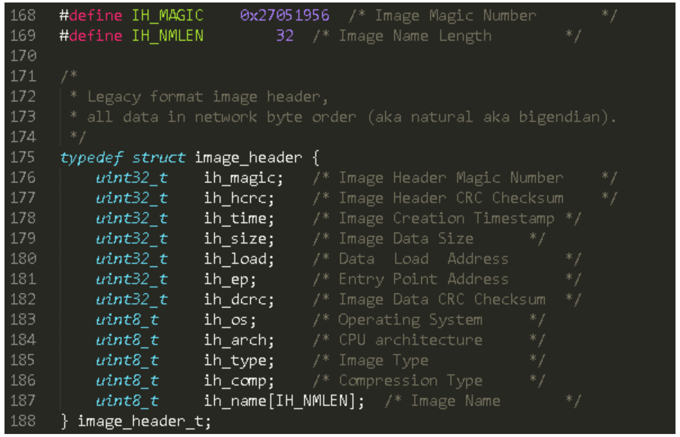
Function name
sub_6E4
sub_A14
sub_AD4
sub_C28
sub_C64
sub_C74
sub_CF4
sub_D80
sub_DCC
sub_F0C
sub_1260
sub_1284
sub_12C8
sub_131C
sub_1374
sub_1398
sub_194C
sub_19CC
sub_1A38
sub_1A90
sub_1AA8
sub_4040
sub_4094
sub_4258
sub_4428
sub_4438
sub_447C
sub_45C0
sub_486C

```
sub_1284
LDR                 R3, =0x30F9B70A
LDRH                R3, [R3]
MOV                 R0, R3,LSR#8
MOV                 R0, R0,LSL#16
ORR                 R0, R0, R3,LSL#24
LDR                 R2, =0xFFF
CMN                 R0, #0x10000
MOVEQ               R0, R2
MOVNE               R0, R0,LSR#16
MOV                 R0, R0,LSL#20
MOV                 R0, R0,LSR#20
CMP                 R0, R2
MOVNE               R0, #0x12
MOVEQ               R0, #0xE
RET
; End of function sub_1284
```

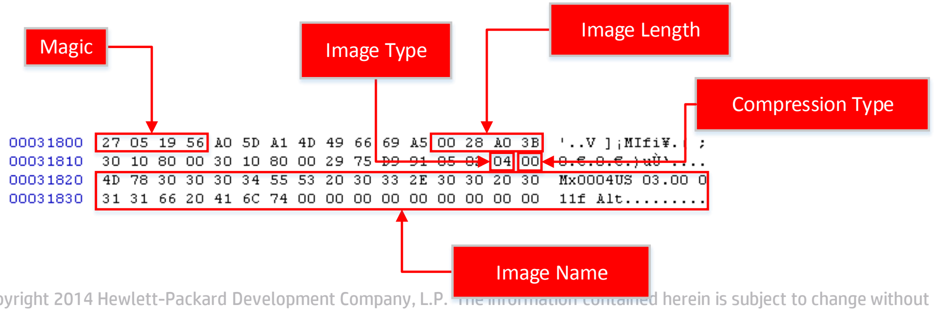
After the 1st stage boot loader, there is a next level boot loader that performs various, more complex operations:

- The kernel image and actual file system are placed inside

U-Boot image header structure



```
#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;
```



```
00031800  27 05 19 56 A0 5D A1 4D 49 66 69 A5 00 28 A0 3B  '..V ]¡MIfi¥.( ;
00031810  30 10 80 00 30 10 80 00 29 75 D9 91 05 03 04 00  0.€.0.€.)uÙ'....
00031820  4D 78 30 30 30 34 55 53 20 30 33 2E 30 30 20 30  Mx0004US 03.00 0
00031830  31 31 66 20 41 6C 74 00 00 00 00 00 00 00 00 00  11f Alt.........
```

The important value in retrieving the whole image file is the image length:

- The header size is 0x40 and the image length is 0x28A03B in this case. This makes total image size of 0x28A07B.

Calculating U-Boot image size on a bare metal image

For my example:

One page is 0x200 bytes, so a page of 0x28A07B/0x200 = 0x1450 and more of 0x28A07B%0x200 = 0x7B bytes are needed

One page on the NAND dump image is 0x210 because of the extra OOB size (0x10)

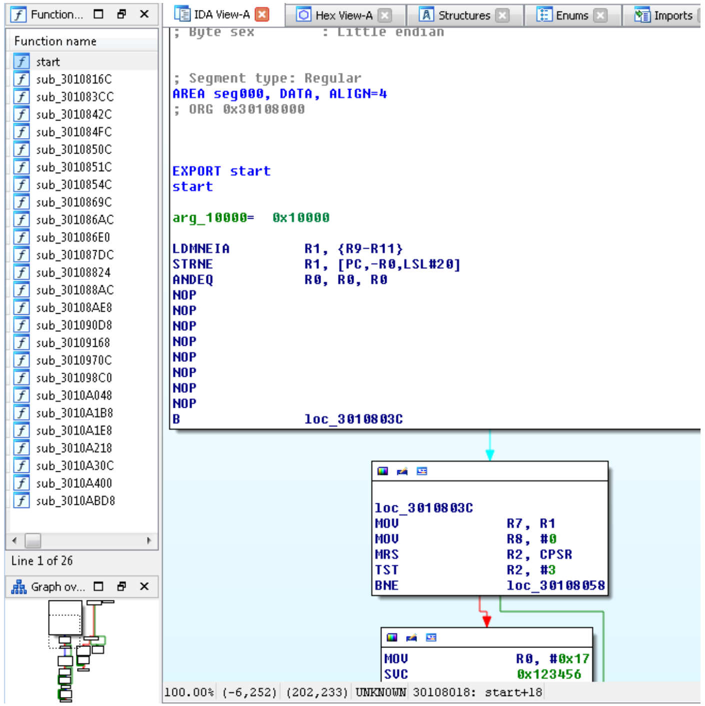
So the physical address of the image end is similar to the following:

page count * (page size + oob size) + extra data
= 0x1450 * (0x200 + 0x10) + 0x7b
= 0x29E57B

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

```
c:\python27\python DumpFlash.py -r 0x00031800 0x002CFD7B -o Dump-00031800-UBOOT.dmp flash.dmp
```

U-Boot image disassembly

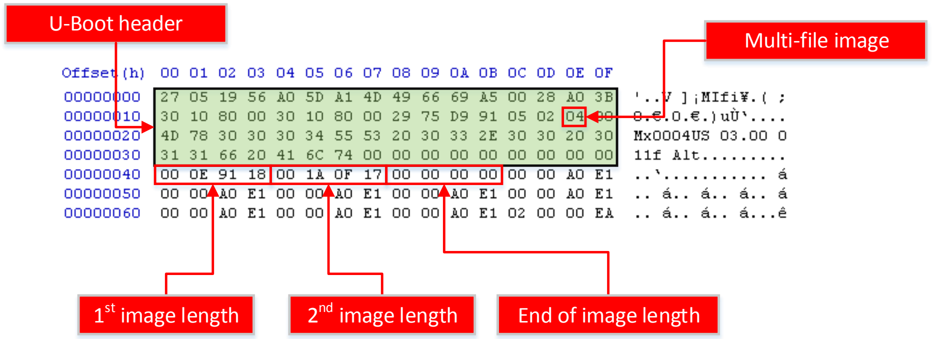


```
Function name
start
sub_3010816C
sub_301083CC
sub_3010842C
sub_301084FC
sub_3010850C
sub_3010851C
sub_3010854C
sub_3010869C
sub_301086AC
sub_301086E0
sub_301087DC
sub_30108824
sub_301088AC
sub_30108AE8
sub_301090D8
sub_30109168
sub_3010970C
sub_301098C0
sub_3010A048
sub_3010A1B8
sub_3010A1E8
sub_3010A218
sub_3010A30C
sub_3010A400
sub_3010ABD8
Line 1 of 26
```

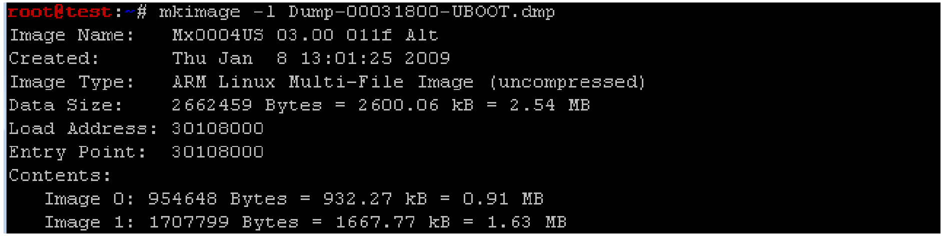
```
; Byte sex        : Little endian

; Segment type: Regular
AREA seg000, DATA, ALIGN=4
; ORG 0x30108000


EXPORT start
start

arg_10000=  0x10000

LDMNEIA         R1, {R9-R11}
STRNE           R1, [PC,-R0,LSL#20]
ANDEQ           R0, R0, R0
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
B               loc_3010803C

loc_3010803C
MOV                     R7, R1
MOV                     R8, #0
MRS                     R2, CPSR
TST                     R2, #3
BNE                     loc_30108058

MOV                     R0, #0x17
SVC                     0x123456

100.00% (-6,252) (202,233) UNKNOWN 30108018: start+18
```

IDA doesn't do well with multi-file images

Multi-file image



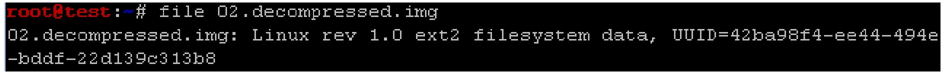
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  27 05 19 56 A0 5D A1 4D 49 66 69 A5 00 28 A0 3B  '..V ]¡MIfi¥.( ;
00000010  30 10 80 00 30 10 80 00 29 75 D9 91 05 02 04 00  0.€.0.€.)uÙ`....
00000020  4D 78 30 30 30 34 55 53 20 30 33 2E 30 30 20 30  Mx0004US 03.00 0
00000030  31 31 66 20 41 6C 74 00 00 00 00 00 00 00 00 00  11f Alt.........
00000040  00 0E 91 18 00 1A 0F 17 00 00 00 00 00 00 A0 E1  ..`.......... á
00000050  00 00 A0 E1 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1  .. á.. á.. á.. á
00000060  00 00 A0 E1 00 00 A0 E1 00 00 A0 E1 02 00 00 EA  .. á.. á.. á...ê
```



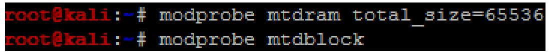
```
root@test:~# mkimage -l Dump-00031800-UBOOT.dmp
Image Name:   Mx0004US 03.00 011f Alt
Created:      Thu Jan  8 13:01:25 2009
Image Type:   ARM Linux Multi-File Image (uncompressed)
Data Size:    2662459 Bytes = 2600.06 kB = 2.54 MB
Load Address: 30108000
Entry Point:  30108000
Contents:
   Image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

This image has two images inside it with lengths of 0x000E9118 and 0x001A0F17

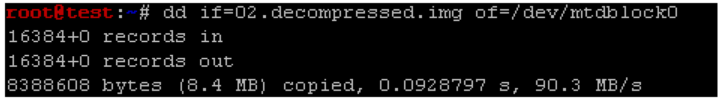
Mounting RAMdisk image



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



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

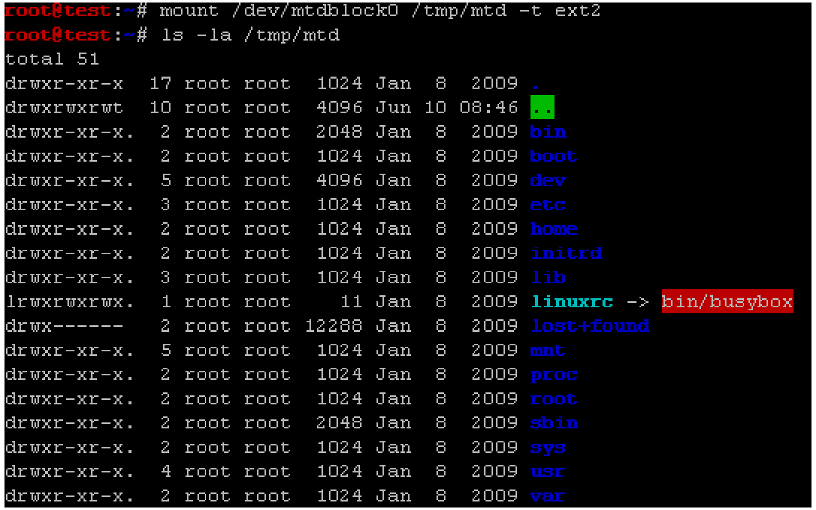


```
root@test:~# dd if=02.decompressed.img of=/dev/mtdblock0
16384+0 records in
16384+0 records out
8388608 bytes (8.4 MB) copied, 0.0928797 s, 90.3 MB/s
```

When image 0 looks like a code file, image 1 has more interesting contents:

- You can identify that it is gzip compressed
- After decompression, if you run file command on the file, it shows that the file is an ext2 file system file

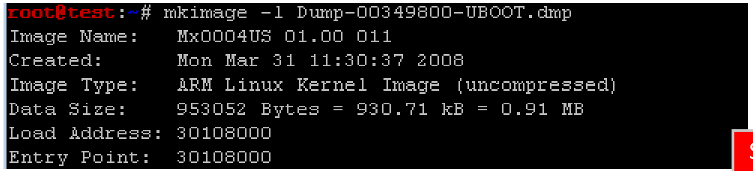
Mounting RAMdisk image



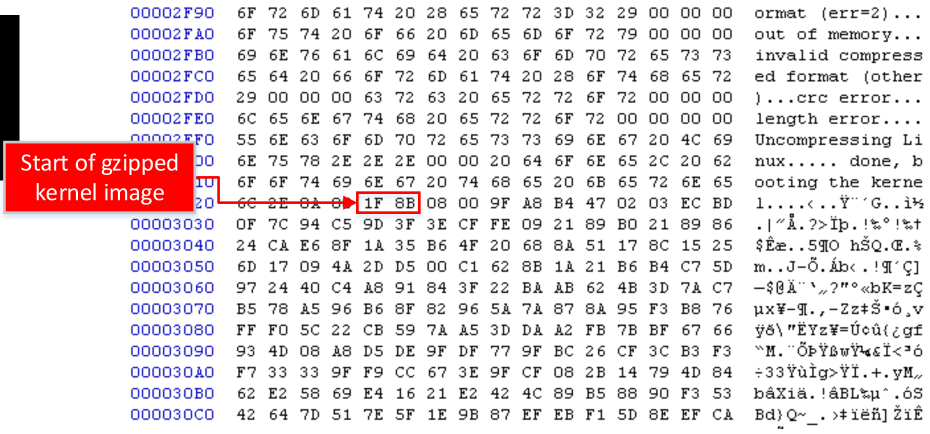
```
root@test:~# mount /dev/mtdblock0 /tmp/mtd -t ext2
root@test:~# ls -la /tmp/mtd
total 51
drwxr-xr-x  17 root root  1024 Jan  8  2009 .
drwxrwxrwt  10 root root  4096 Jun 10 08:46 ..
drwxr-xr-x.  2 root root  2048 Jan  8  2009 bin
drwxr-xr-x.  2 root root  1024 Jan  8  2009 boot
drwxr-xr-x.  5 root 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
```

After pushing the image, you can mount the MTD block device using the mount command and browse and modify the file.

mkimage information for 2nd U-Bootimage



```
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
```

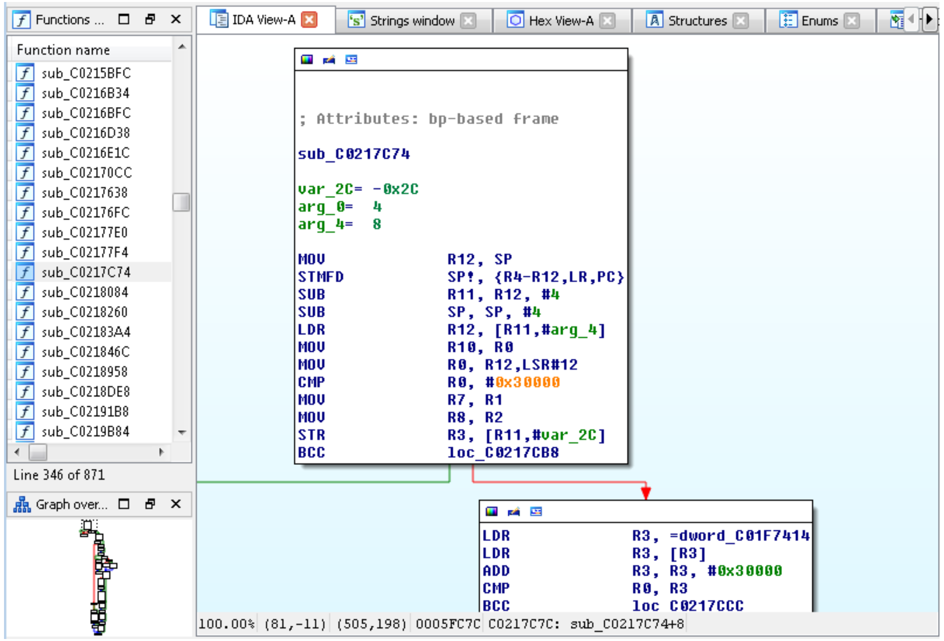


```
00002F90   6F 72 6D 61 74 20 28 65 72 72 3D 32 29 00 00 00   ormat (err=2)...
00002FA0   6F 75 74 20 6F 66 20 6D 65 6D 6F 72 79 00 00 00   out of memory...
00002FB0   69 6E 76 61 6C 69 64 20 63 6F 6D 70 72 65 73 73   invalid compress
00002FC0   65 64 20 66 6F 72 6D 61 74 20 28 6F 74 68 65 72   ed format (other
00002FD0   29 00 00 00 63 72 63 20 65 72 72 6F 72 00 00 00   )...crc error...
00002FE0   6C 65 6E 67 74 68 20 65 72 72 6F 72 00 00 00 00   length error....
00002FF0   55 6E 63 6F 6D 70 72 65 73 73 69 6E 67 20 4C 69   Uncompressing Li
      00   6E 75 78 2E 2E 2E 00 00 20 64 6F 6E 65 2C 20 62   nux..... done, b
      10   6F 6F 74 69 6E 67 20 74 68 65 20 6B 65 72 6E 65   ooting the kerne
      20   6C 2E 0A 0D 1F 8B 08 00 9F A8 B4 47 02 03 EC BD   l....‹..Ÿ¨´G..ì½
00003030   0F 7C 94 C5 9D 3F 3E CF FE 09 21 89 B0 21 89 86   .|”Å.?>Ïþ.!‰°!‰†
00003040   24 CA E6 8F 1A 35 B6 4F 20 68 8A 51 17 8C 15 25   $Êæ..5¶O hŠQ.Œ.%
00003050   6D 17 09 4A 2D D5 00 C1 62 8B 1A 21 B6 B4 C7 5D   m..J-Õ.Áb‹.!¶´Ç]
00003060   97 24 40 C4 A8 91 84 3F 22 BA AB 62 4B 3D 7A C7   —$@Ä¨‘„?"º«bK=zÇ
00003070   B5 78 A5 96 B6 8F 82 96 5A 7A 87 8A 95 F3 B8 76   µx¥–¶.‚–Zz‡Š•ó¸v
00003080   FF F0 5C 22 CB 59 7A A5 3D DA A2 FB 7B BF 67 66   ÿð\"ËYz¥=Ú¢û{¿gf
00003090   93 4D 08 A8 D5 DE 9F DF 77 9F BC 26 CF 3C B3 F3   “M.¨ÕÞŸßwŸ¼&Ï<³ó
000030A0   F7 33 33 9F F9 CC 67 3E 9F CF 08 2B 14 79 4D 84   ÷33ŸùÌg>ŸÏ.+.yM„
000030B0   62 E2 58 69 E4 16 21 E2 42 4C 89 B5 88 90 F3 53   bâXiä.!âBL‰µˆ.óS
000030C0   42 64 7D 51 7E 5F 1E 9B 87 EF EB F1 5D 8E EF CA   Bd}Q~_.›‡ïëñ]ŽïÊ
```

IDA loads this image up without any issues. There are no hidden images.

- Unfortunately the code shown by IDA is the bootstrapping code that decompresses following the gzipped kernel image
- To identify the start of the kernel image, search for the gzip image magic value (0x8b1f)

Kernel image disassembly



```
sub_C0215BFC
sub_C0216B34
sub_C0216BFC
sub_C0216D38
sub_C0216E1C
sub_C02170CC
sub_C0217638
sub_C02176FC
sub_C02177E0
sub_C02177F4
sub_C0217C74
sub_C0218084
sub_C0218260
sub_C02183A4
sub_C021846C
sub_C0218958
sub_C0218DE8
sub_C02191B8
sub_C0219B84
```

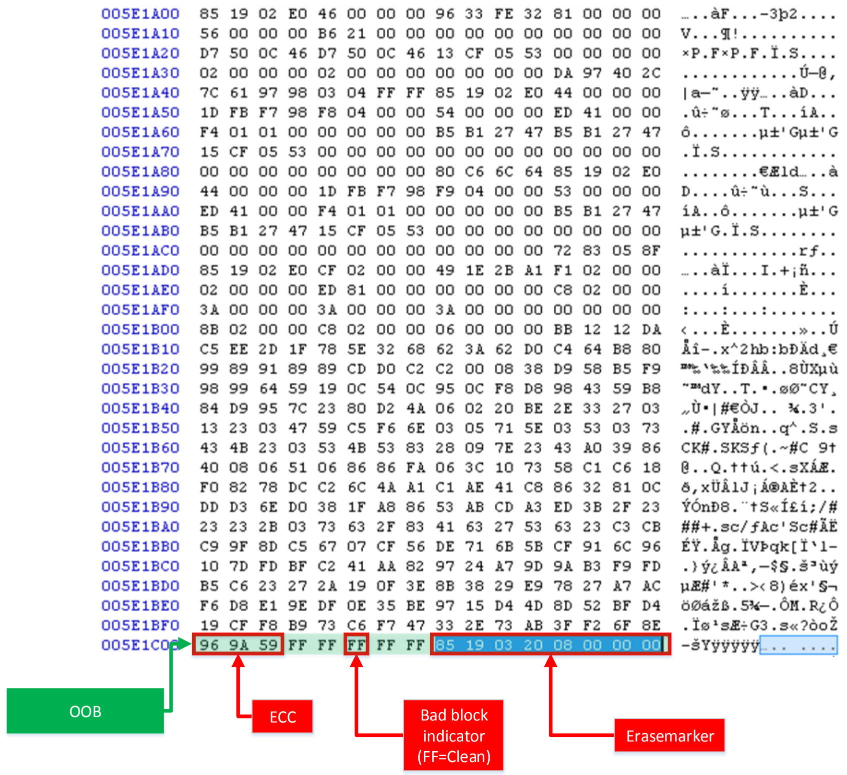
Line 346 of 871

```
; Attributes: bp-based frame

sub_C0217C74

var_2C= -0x2C
arg_0=  4
arg_4=  8

MOV             R12, SP
STMFD           SP!, {R4-R12,LR,PC}
SUB             R11, R12, #4
SUB             SP, SP, #4
LDR             R12, [R11,#arg_4]
MOV             R10, R0
MOV             R0, R12,LSR#12
CMP             R0, #0x30000
MOV             R7, R1
MOV             R8, R2
STR             R3, [R11,#var_2C]
BCC             loc_C0217CB8
```

```
LDR             R3, =dword_C01F7414
LDR             R3, [R3]
ADD             R3, R3, #0x30000
CMP             R0, R3
BCC             loc_C0217CCC
```

```
100.00% (81,-11) (505,198) 0005FC7C C0217C7C: sub_C0217C74+8
```

JFFS2 erase marker location from a page and spare column bytes

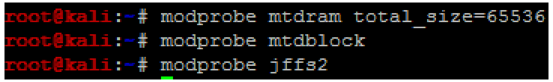


```
005E1A00  85 19 02 E0 46 00 00 00 96 33 FE 32 81 00 00 00   …..àF...–3þ2....
005E1A10  56 00 00 00 B6 21 00 00 00 00 00 00 00 00 00 00   V...¶!..........
005E1A20  D7 50 0C 46 D7 50 0C 46 13 CF 05 53 00 00 00 00   ×P.F×P.F.Ï.S....
005E1A30  02 00 00 00 02 00 00 00 00 00 00 00 DA 97 40 2C   ............Ú–@,
005E1A40  7C 61 97 98 03 04 FF FF 85 19 02 E0 44 00 00 00   |a–˜..ÿÿ…..àD...
005E1A50  1D FB F7 98 F8 04 00 00 54 00 00 00 ED 41 00 00   .û÷˜ø...T...íA..
005E1A60  F4 01 01 00 00 00 00 00 B5 B1 27 47 B5 B1 27 47   ô.......µ±'Gµ±'G
005E1A70  15 CF 05 53 00 00 00 00 00 00 00 00 00 00 00 00   .Ï.S............
005E1A80  00 00 00 00 00 00 00 00 80 C6 6C 64 85 19 02 E0   ........€Æld…..à
005E1A90  44 00 00 00 1D FB F7 98 F9 04 00 00 53 00 00 00   D....û÷˜ù...S...
005E1AA0  ED 41 00 00 F4 01 01 00 00 00 00 00 B5 B1 27 47   íA..ô.......µ±'G
005E1AB0  B5 B1 27 47 15 CF 05 53 00 00 00 00 00 00 00 00   µ±'G.Ï.S........
005E1AC0  00 00 00 00 00 00 00 00 00 00 00 00 72 83 05 8F   ............rƒ..
005E1AD0  85 19 02 E0 CF 02 00 00 49 1E 2B A1 F1 02 00 00   …..àÏ...I.+¡ñ...
005E1AE0  02 00 00 00 ED 81 00 00 00 00 00 00 C8 02 00 00   ....í.......È...
005E1AF0  3A 00 00 00 3A 00 00 00 3A 00 00 00 00 00 00 00   :...:...:.......
005E1B00  8B 02 00 00 C8 02 00 00 06 00 00 00 BB 12 12 DA   ‹...È.......»..Ú
005E1B10  C5 EE 2D 1F 78 5E 32 68 62 3A 62 D0 C4 64 B8 80   Åî-.x^2hb:bÐÄd¸€
005E1B20  99 89 91 89 89 CD D0 C2 C2 00 08 38 D9 58 B5 F9   ™‰'‰‰ÍÐÂÂ..8ÙXµù
005E1B30  98 99 64 59 19 0C 54 0C 95 0C F8 D8 98 43 59 B8   ˜™dY..T.•.øØ˜CY¸
005E1B40  84 D9 95 7C 23 80 D2 4A 06 02 20 BE 2E 33 27 03   „Ù•|#€ÒJ.. ¾.3'.
005E1B50  13 23 03 47 59 C5 F6 6E 03 05 71 5E 03 53 03 73   .#.GYÅön..q^.S.s
005E1B60  43 4B 23 03 53 4B 53 83 28 09 7E 23 43 A0 39 86   CK#.SKSƒ(.~#C 9†
005E1B70  40 08 06 51 06 86 86 FA 06 3C 10 73 58 C1 C6 18   @..Q.††ú.<.sXÁÆ.
005E1B80  F0 82 78 DC C2 6C 4A A1 C1 AE 41 C8 86 32 81 0C   ð‚xÜÂlJ¡Á®AÈ†2..
005E1B90  DD D3 6E D0 38 1F A8 86 53 AB CD A3 ED 3B 2F 23   ÝÓnÐ8.¨†S«Í£í;/#
005E1BA0  23 23 2B 03 73 63 2F 83 41 63 27 53 63 23 C3 CB   ##+.sc/ƒAc'Sc#ÃË
005E1BB0  C9 9F 8D C5 67 07 CF 56 DE 71 6B 5B CF 91 6C 96   ÉŸ.Åg.ÏVÞqk[Ï'l–
005E1BC0  10 7D FD BF C2 41 AA 82 97 24 A7 9D 9A B3 F9 FD   .}ý¿ÂAª‚—$§.š³ùý
005E1BD0  B5 C6 23 27 2A 19 0F 3E 8B 38 29 E9 78 27 A7 AC   µÆ#'*..>‹8)éx'§¬
005E1BE0  F6 D8 E1 9E DF 0E 35 BE 97 15 D4 4D 8D 52 BF D4   öØážß.5¾—.ÔM.R¿Ô
005E1BF0  19 CF F8 B9 73 C6 F7 47 33 2E 73 AB 3F F2 6F 8E   .Ïø¹sÆ÷G3.s«?òoŽ
005E1C00  96 9A 59 FF FF FF FF FF 85 19 03 20 08 00 00 00   –šYÿÿÿÿÿ….. ....
```

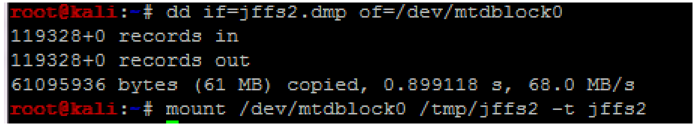
Identifying the JFFS2 file system from the raw NAND Flash image is relatively easy

Usually JFFS2 puts specialized *erasemarkers* inside the spare column of each page

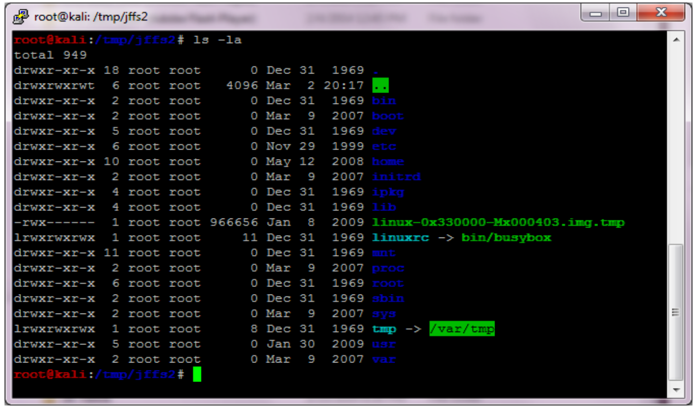
Mounting JFFS2 file system using a MTD



```
root@kali:~# modprobe mtdram total_size=65536
root@kali:~# modprobe mtdblock
root@kali:~# modprobe jffs2
```



```
root@kali:~# dd if=jffs2.dmp of=/dev/mtdblock0
119328+0 records in
119328+0 records out
61095936 bytes (61 MB) copied, 0.899118 s, 68.0 MB/s
root@kali:~# mount /dev/mtdblock0 /tmp/jffs2 -t jffs2
```



```
root@kali:/tmp/jffs2# ls -la
total 949
drwxr-xr-x 18 root root      0 Dec 31  1969 .
drwxrwxrwt  6 root root   4096 Mar  2 20:17 ..
drwxr-xr-x  2 root root      0 Dec 31  1969 bin
drwxr-xr-x  2 root root      0 Mar  9  2007 boot
drwxr-xr-x  5 root root      0 Dec 31  1969 dev
drwxr-xr-x  6 root root      0 Nov 29  1999 etc
drwxr-xr-x 10 root root      0 May 12  2008 home
drwxr-xr-x  2 root root      0 Mar  9  2007 initrd
drwxr-xr-x  4 root root      0 Dec 31  1969 ipkg
drwxr-xr-x  4 root root      0 Dec 31  1969 lib
-rwx------  1 root root 966656 Jan  8  2009 linux-0x330000-Mx000403.img.tmp
lrwxrwxrwx  1 root root     11 Dec 31  1969 linuxrc -> bin/busybox
drwxr-xr-x 11 root root      0 Dec 31  1969 mnt
drwxr-xr-x  2 root root      0 Mar  9  2007 proc
drwxr-xr-x  6 root root      0 Dec 31  1969 root
drwxr-xr-x  2 root root      0 Dec 31  1969 sbin
drwxr-xr-x  2 root root      0 Mar  9  2007 sys
lrwxrwxrwx  1 root root      8 Dec 31  1969 tmp -> /var/tmp
drwxr-xr-x  5 root root      0 Jan 30  2009 usr
drwxr-xr-x  2 root root      0 Mar  9  2007 var
root@kali:/tmp/jffs2#
```

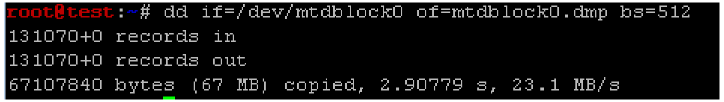
First, you need to create a MTD device:

- Load related Linux kernel modules like mtdram, mtdblock and JFFS2. This creates a MTD device on the system.

After successful mounting, you can navigate and modify the file system on the fly

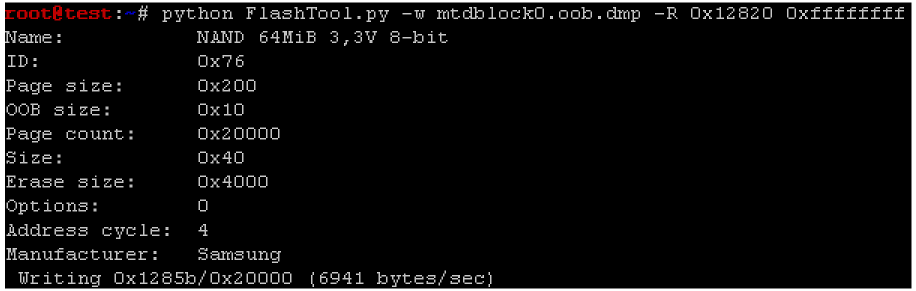
Writing JFFS2 data

1. Dump mtdblock data to a file



```
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
```

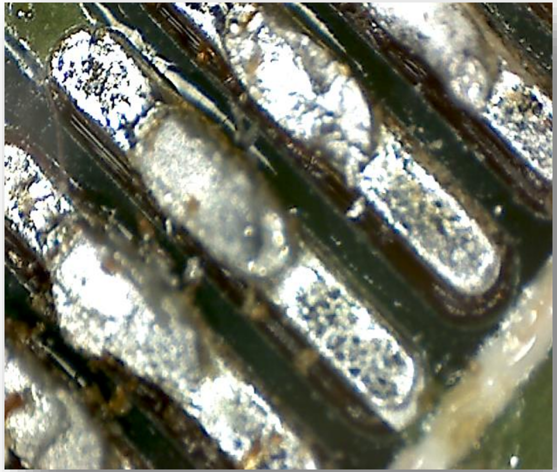
2. Add OOB area
   - *python DumpFlash.py -R -o mtdblock0.oob.dmp mtdblock0.dmp*



```
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)
```

3. Program memory at JFFS2 location
   - *python FlashTool.py -w mtdblock.mod.oob.dmp -R 0x12820 0xffffffff*

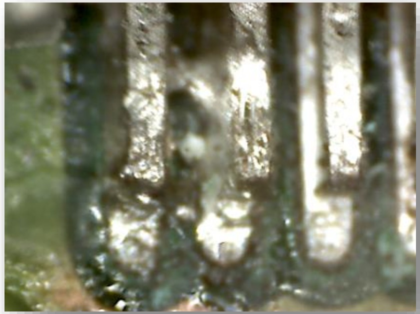
SMT Re-soldering



After modifying the raw data and writing it back to the Flash memory, re-solder the chip:

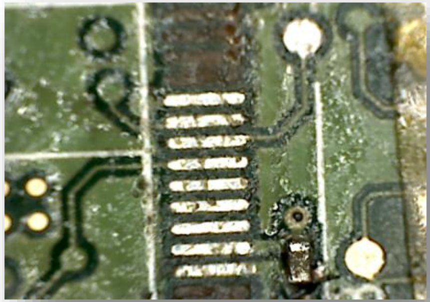
- The re-soldering process is not much different from usual SMT soldering
- SMT was originally developed for automatic soldering of the PCB components
- The chips are usually small and the pitch of the pins is also relatively small
- Soldering these chips to the PCB manually is challenging, but not *terribly* difficult
- There are many different methods, but I placed the chip on the pin location and heated the pins using the soldering iron
- The solder residue left from previous de-soldering process melts again and the chip can be soldered again using this solder
- Various other detailed technologies can be found on the Internet

Bridge & damaged pins



There are many pitfalls with SMT soldering and one of the big issues is bridging:

- The pitch for NAND flash TSOP48 model is 0.5 mm (which is extremely small). The solder can go over multiple pins and create shorts



One of the big problems with re-soldering is possible damage to the board:

- Excessive heat is applied during desoldering and it can damage the PCB board
- Be extra careful when you re-solder the chips!
- Luckily, with Flash memory, many pins are not used at all. If the damaged patterns are not used, then the chips operate normally
- Check with the chip datasheet to see if damaged patterns are used by the chip

My 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): added writing support

- https://github.com/ohjeongwook/NANDReader_FTDI
  - Write support, which is missing from original NANDTool project from Sprites Mode

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 is increasingly relevant as vendors obfuscate or remove JTAG interfaces to protect their intellectual property
- By directly interacting with the low level Flash memory interface, you can access data that sometimes can't be retrieved otherwise

The de-soldering method is referred to as destructive, but it is still possible to re-solder the chip to the system using SMT soldering methods:

- There is more chance of damaging the circuit board, but the chance of success is still high enough

There are many factors 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 the boot loaders, the kernel or the JFFS2 root image

Credits

Original design of NAND reader/writer

  Sprites Mod

NANDTool

  Sprites Mod and Bjoern Kerlers