Reverse Engineering Flash Memory for Fun and Benefit

Jeong Wook (Matt) Oh / 2014

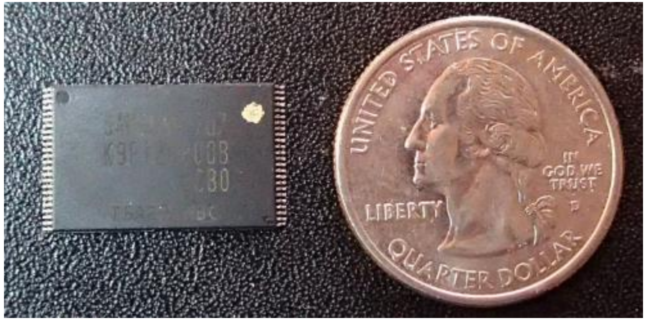
oh@hp.com

oh.jeongwook@gmail.com

NAND Flash Memory is everywhere

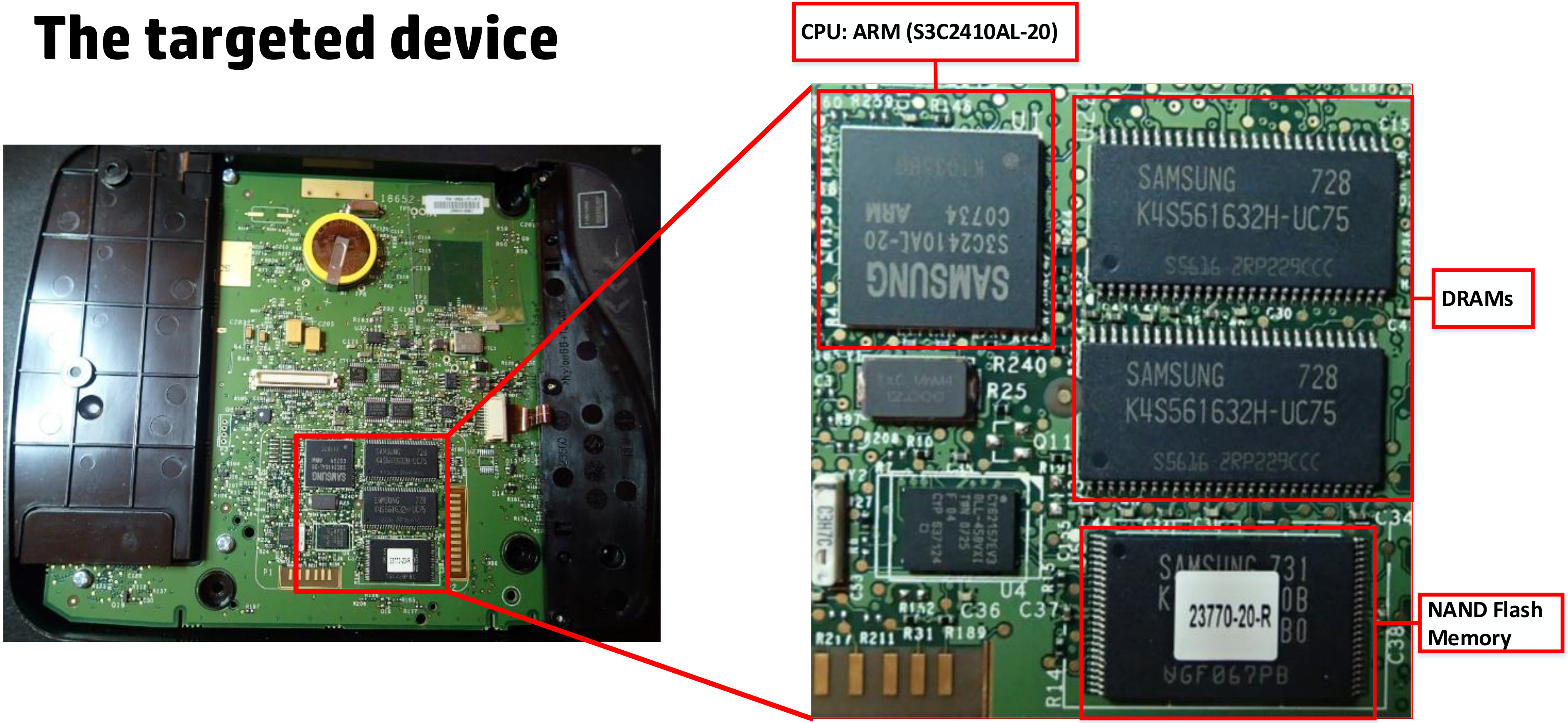
NAND Flash is used in:

- USB Sticks
- Phones
- Cameras
- Embedded devices
- Smart appliances
- IoT
- …



Even in the POS devices at the grocery store…

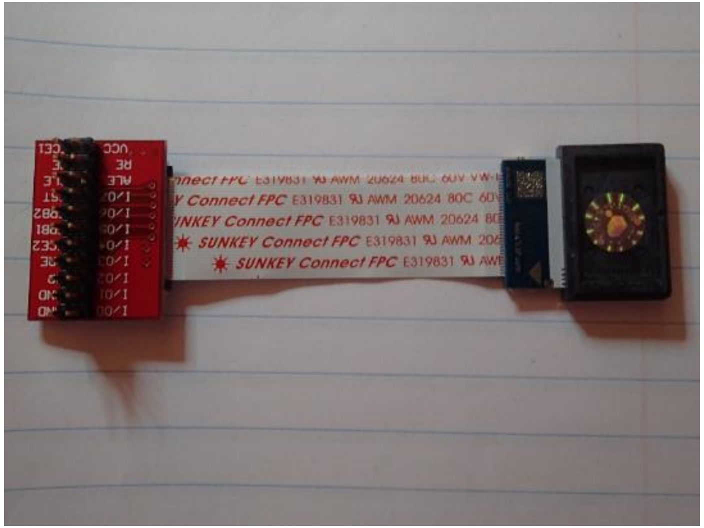
The targeted device

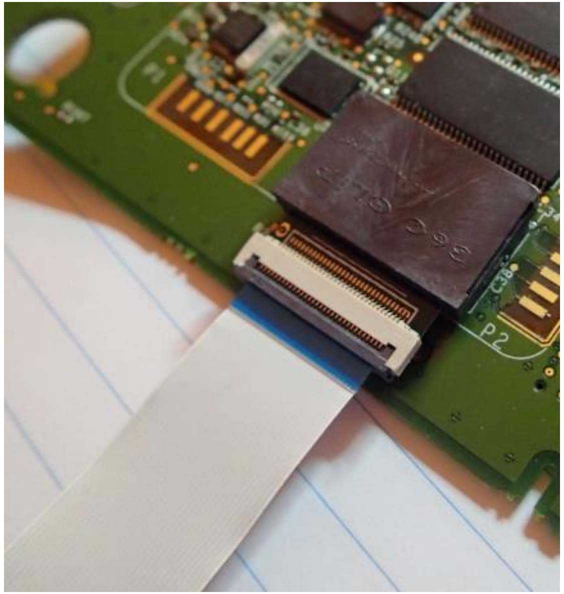


NAND Flash memory pins and names

De-soldering

Failed attempt – want to avoid de-soldering?

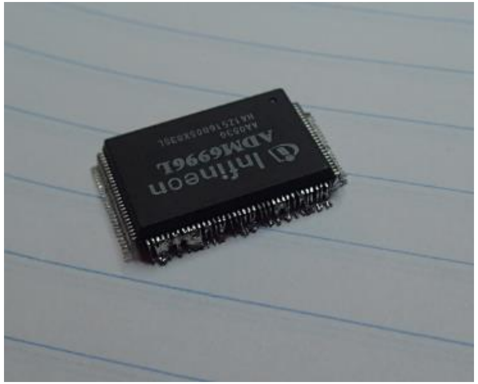




Pins essential to advanced operations are missing and other chips on the board may be woken up by the electric current supplied by the external device:

- This interferes with our custom operations

Failed attempt – want to use de-solder wire?



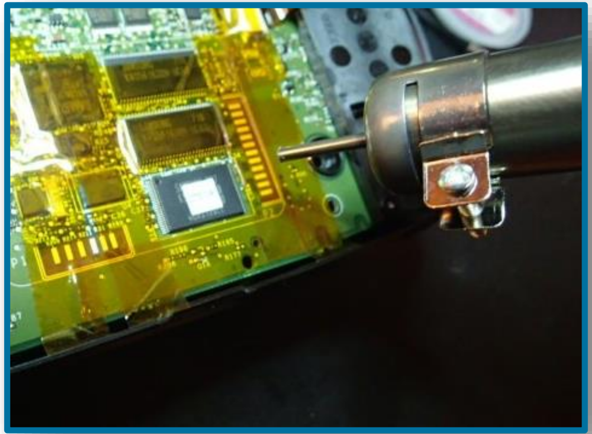
It’s tough to remove after de-soldering

Equipment & supplies

Use an SMT Rework station with a hot air blower:

- The solder alloy melts at around 180 to 190°C (360 and 370°F), but I recommend setting the temperature higher
- Use insulation tape to minimize heat damage

De-soldering



Apply heat over the pin area

- It usually takes 1 to 2 minutes to fully de-solder the chip

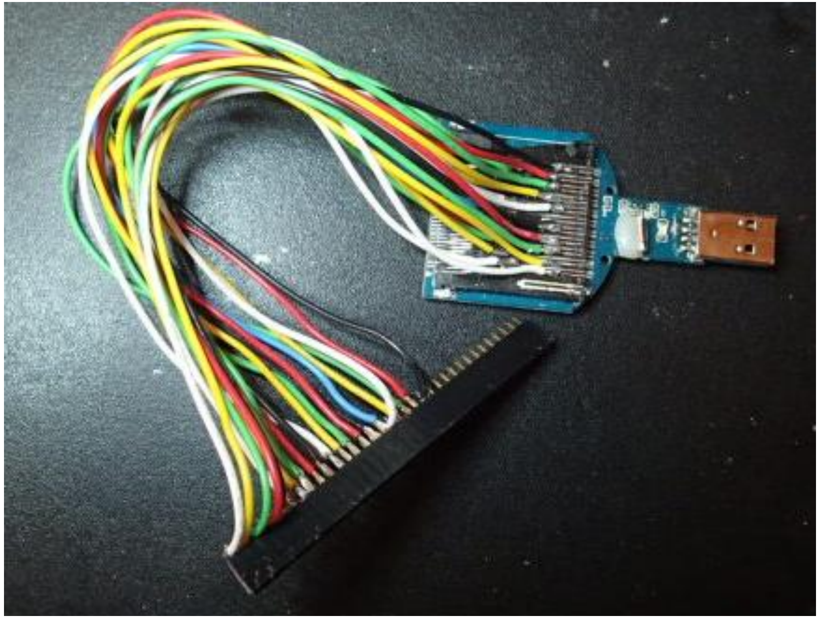
Too much heat?

Demo: The de-soldering process

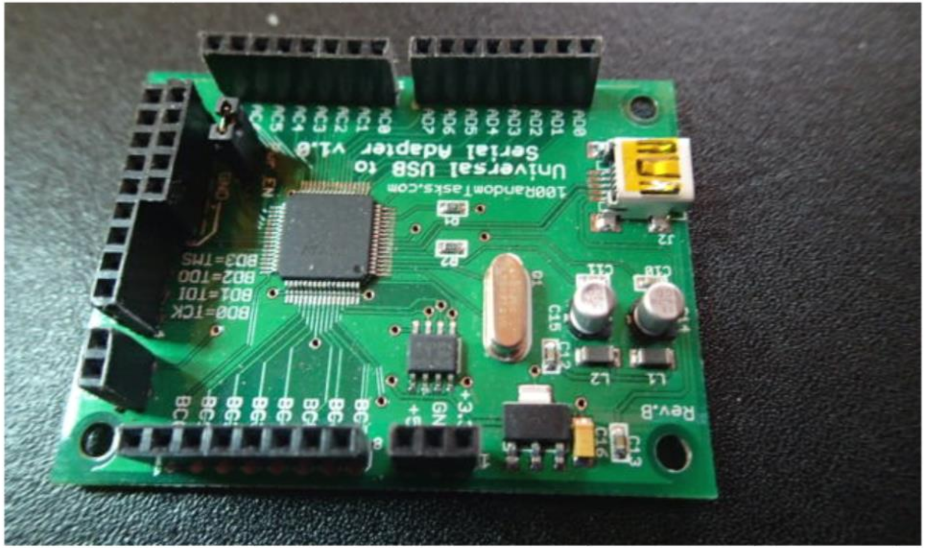
12

FTDI FT2232H & NAND Flash Memory

Failed attempt – modding an SD reader



FTDI FT2232H breakout board

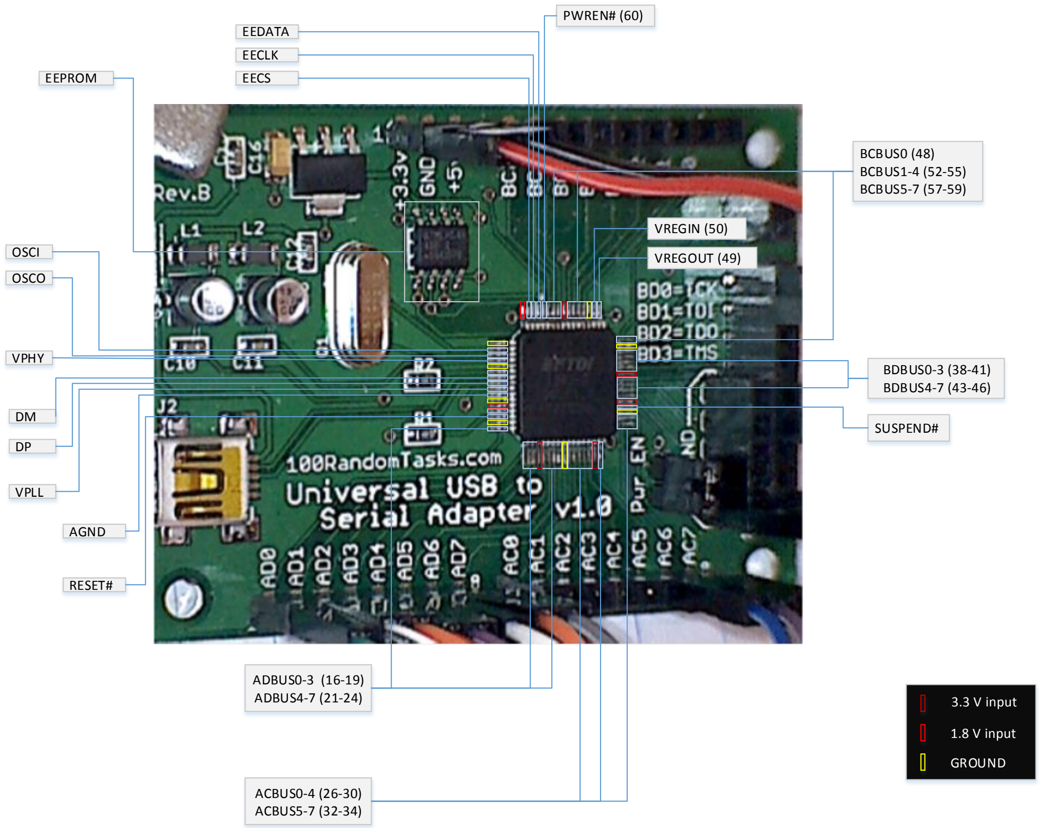


A chip for USB communication

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

Note: Put female pin headers on each port extension

FTDI FT2232H breakout board



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

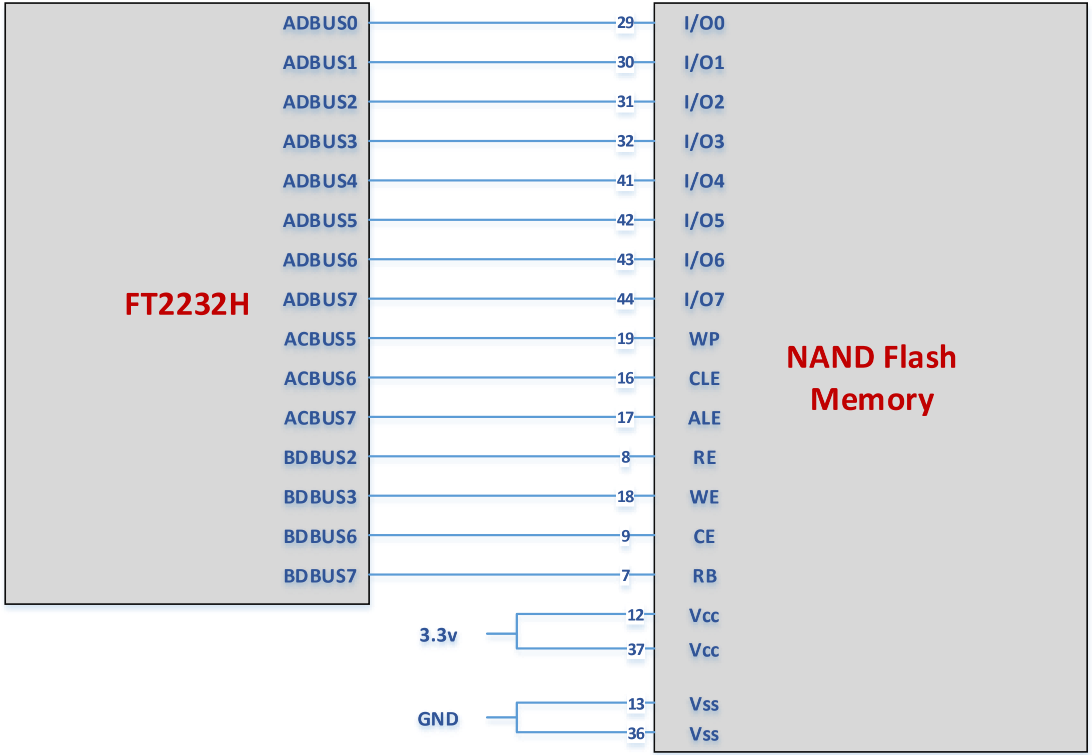
Commands	Operation	Address
0x90	Read	8bit address
0x91	Read	16bit address
0x92	Write	8bit address
0x93	Write	16bit address
0x82	Set	High byte (BDBUS6, 7)
0x83	Read	High byte (BDBUS6, 7)

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

- See FTDI's note for more detail

Connection between FT2232H and NAND Flash Memory

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



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.

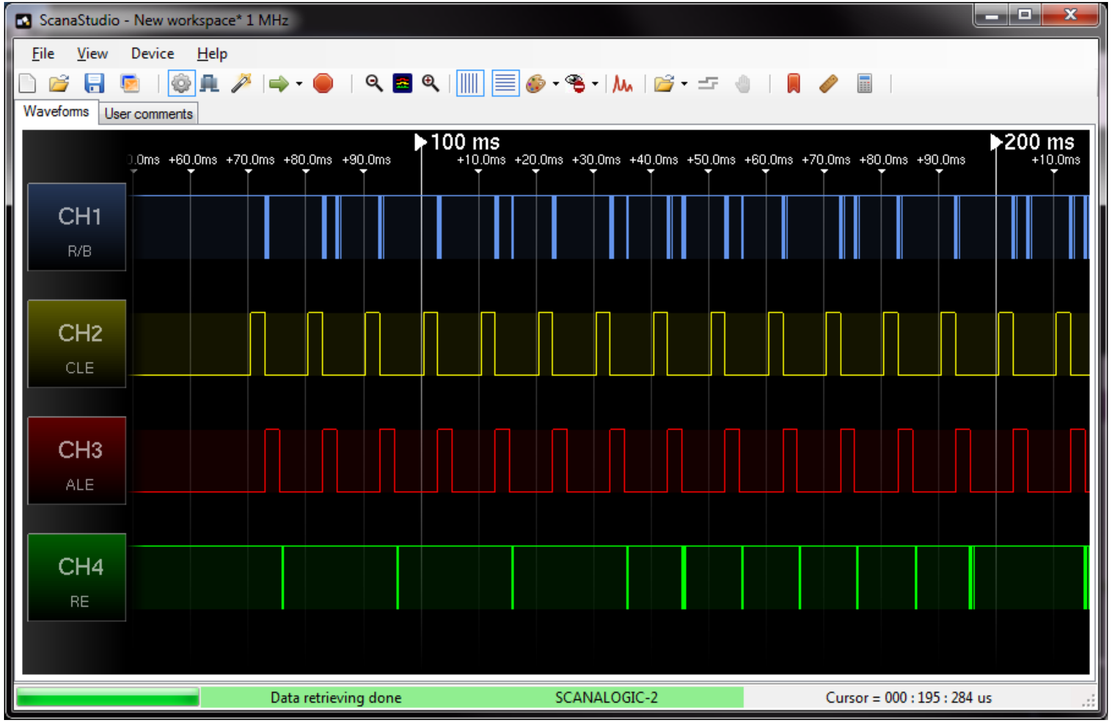
- BDBUS6 (I/O0), BDBUS7 (I/O1) is controlled by the 0x83, 0x82 command
- RD#, WR# is connected to the RE, WE pin on NAND Flash

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

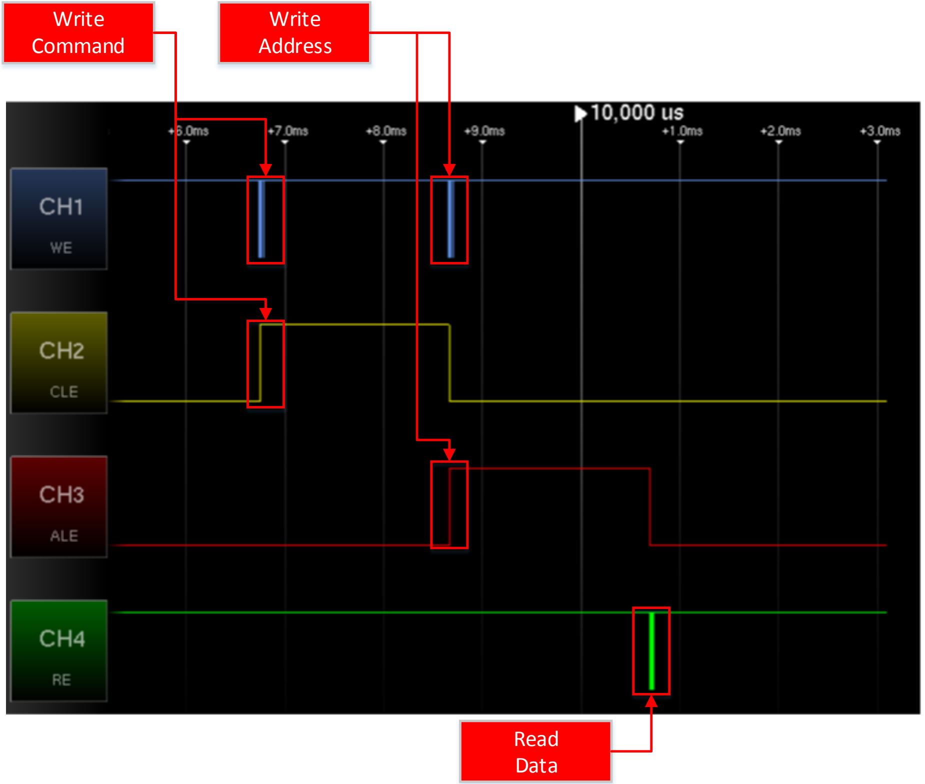
<table>
<tr><th>CLE</th><td>1</td><td colspan="3">0</td></tr>
<tr><th>ALE</th><td>0</td><td>1</td><td colspan="2">0</td></tr>
<tr><th>R/B</th><td colspan="2">1 (Ready)</td><td>R/B=0 (busy)</td><td>1 (Ready)</td></tr>
<tr><th>RE</th><td colspan="3">1</td><td>Falling for each bytes</td></tr>
<tr><th>WE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

- 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 changes state to send addresses.
3. Finally, RE is used to signal the reading of each byte.

Reading a small block page

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

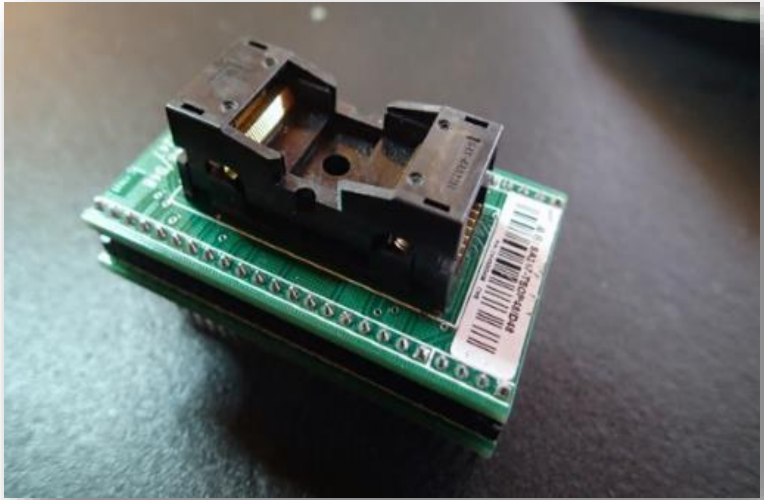
```
self.sendCmd(self.NAND_CMD_READ0)
self.waitReady()
self.sendAddr(pageno<<8,self.AddrCycles)
self.waitReady()
bytes+=self.readData(self.PageSize/2)

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)
```

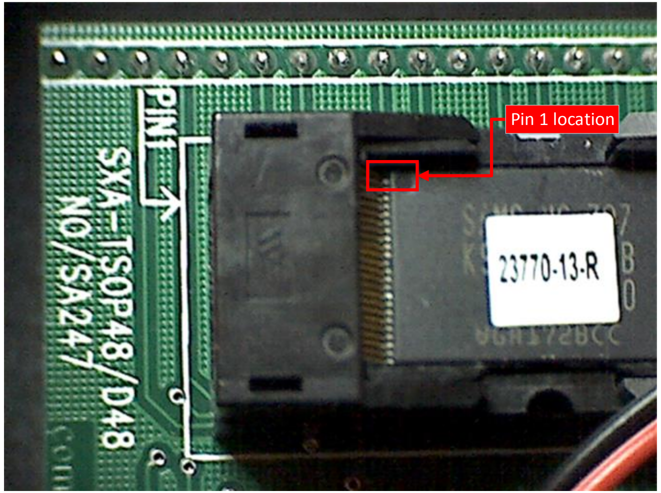
Read A area (0-255)

Read B area (256-511)

Read spare C area (512-527)

TSOP48 socket

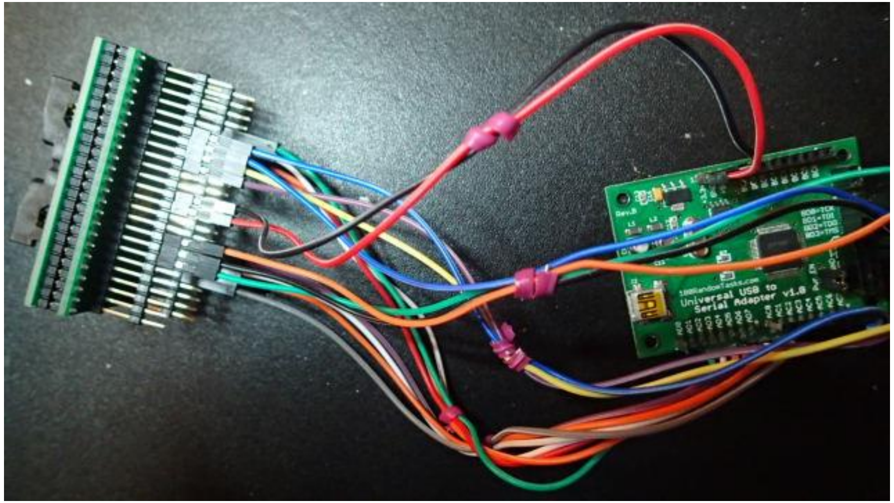


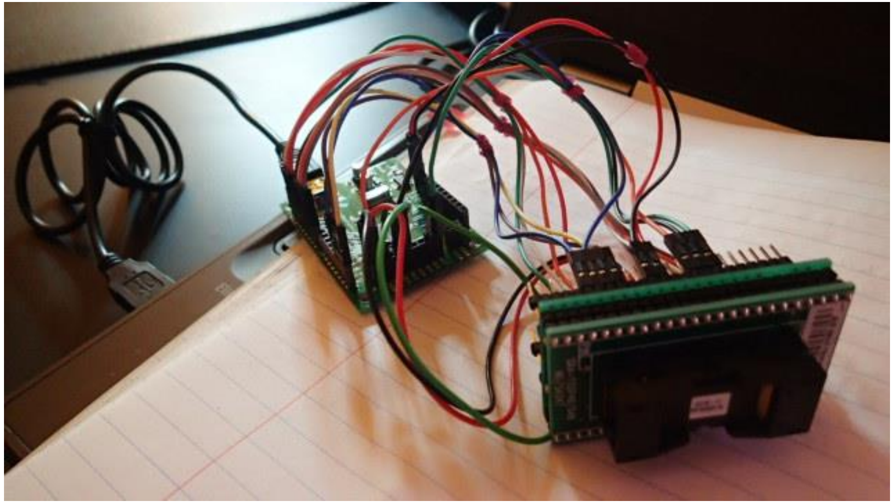


Place your NAND Flash chip inside the TSOP48 socket:

- 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

NAND Flash reader/writer





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

DumpFlash – enhanced Flash reader/writer software

DumpFlash – Python implmentation of Flash reader/writer software

- https://github.com/ohjeongwook/DumpFlash
- Python implementation of original SpirteMod code and more
- Read/Write support
- Flash image manipulation tool (ECC, Bad block check)
- Fast sequential row read mode support
- More experimental code coming
- Automatic U-Boot image extraction
- JFFS2 parsing and extraction
- Uses enhanced schematics

Install prerequisite packages like pyftdi and libusbx.

- Pyftdi: a python wrapper around the FTDI library
- Libusbx: a library that provides USB level access to user applications

DumpFlash: Show basic information

```
C:\mat\Analysis\NAND Flash\DumpFlash>c:\python27\python DumpFlash.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
```

With everything set up, you can query basic Flash information using the *–i* option

DumpFlash: Read data

```
C:\mat\Analysis\NAND Flash\DumpFlash>c:\python27\python DumpFlash.py -r flash.dm
p
Reading page: 436/131073 (58149 bytes/sec)
```

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

DumpFlash: Read data in sequential row read mode

```
C:\mat\Analysis\NAND Flash\DumpFlash>c:\python27\python DumpFlash.py -r -s flash
.dmp
Reading page: 1376/131071 (231092 bytes/sec)
```

DumpFlash supports sequential row read mode.
Specify the –s option and it increases reading performance.
Reading is 5-6 times faster than in normal page-by-page mode.

Demo: DumpFlash in action

Basic DumpFlash operations

Write operation pin states

<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>

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

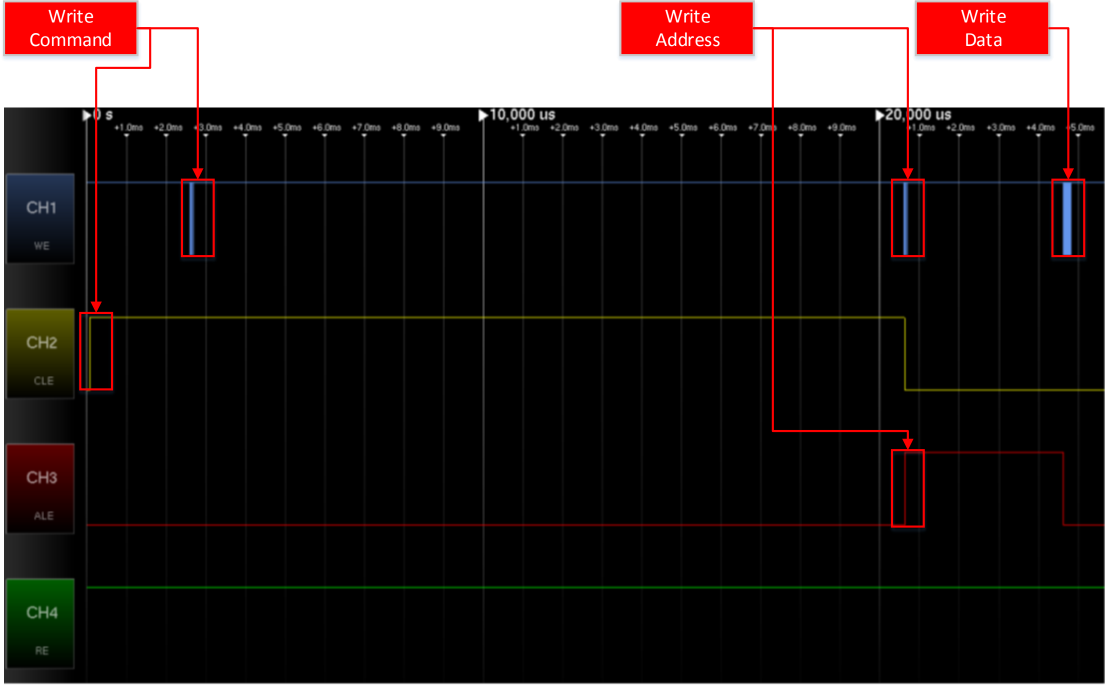
```
self.sendCmd(self.NAND_CMD_READ0)
self.sendCmd(self.NAND_CMD_SEQIN)
self.waitReady()
self.sendAddr(pageno<<8,self.AddrCycles)
self.waitReady()
self.writeData(data[0:256])
self.sendCmd(self.NAND_CMD_PAGEPROG)
err=self.Status()
if err&self.NAND_STATUS_FAIL:
    return err

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
```

Write A area (0-255)

Write B area (256-511)

Write spare C area (512-527)

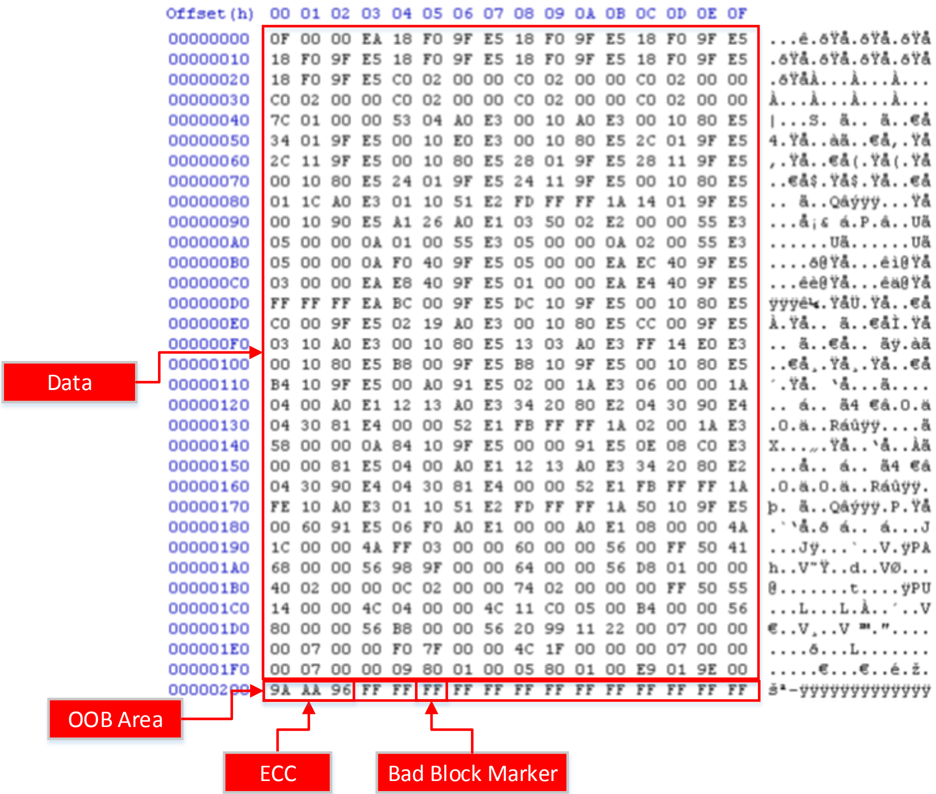
Writing data



After the 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

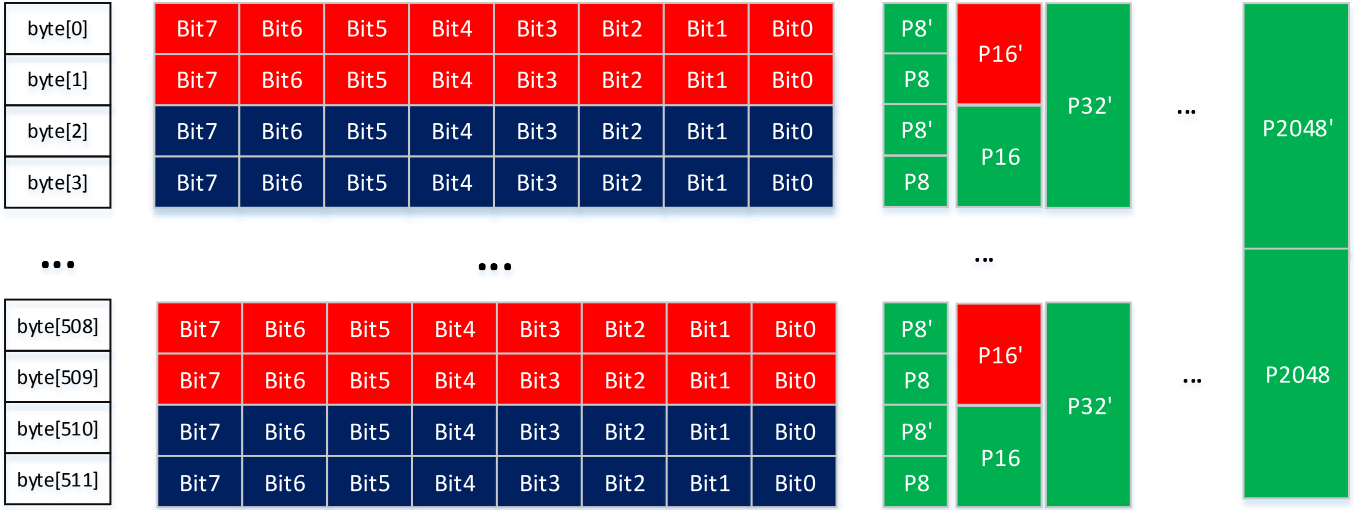
ECC calculation table

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 the 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

byte[0]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
byte[1]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
byte[2]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
byte[3]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
...				...				
byte[508]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
byte[509]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
byte[510]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
byte[511]	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
	P1	P1'	P1	P1'	P1	P1'	P1	P1'
	P2		P2'		P2		P2'	
	P4				P4'			

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

- The notion of ‘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

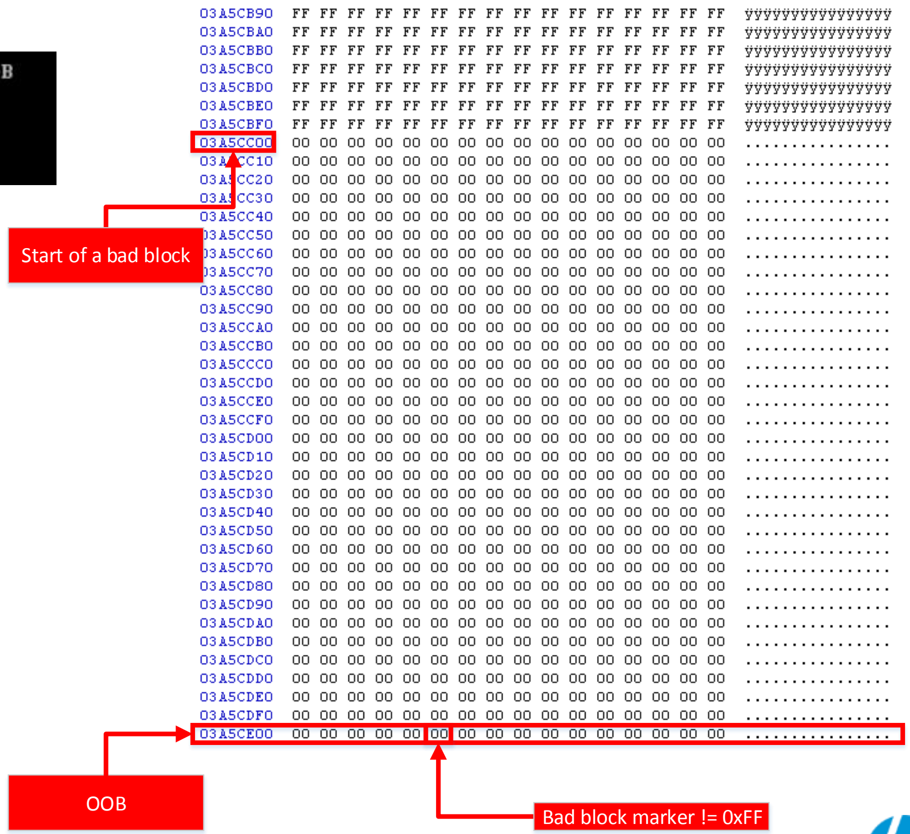
Some vendors use their own scheme for marking bad blocks:

- 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).

```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
```

How a bad block is marked

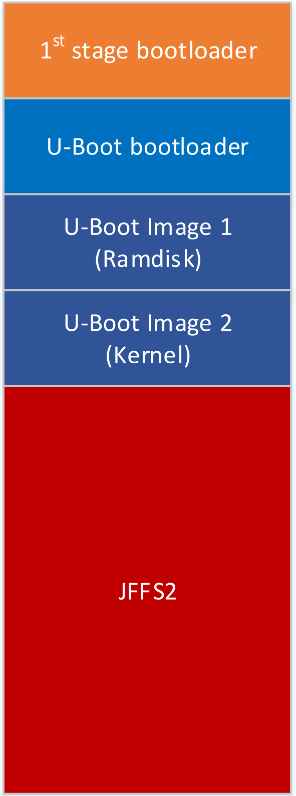
```
C:\mat\Analysis\NAND Flash\DumpFlash>c:\python27\python DumpFlash.py -B
Checking Bad Blocks 9% block: 400/4096
Bad block: 400 (at physical offset 0x672000)
Checking Bad Blocks 19% block: 780/4096
Bad block: 780 (at physical offset 0xc91800)
Checking Bad Blocks 36% block: 1504/4096
```



Demo: DumpFlash for ECC and bad block detections

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
- 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 the 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
```

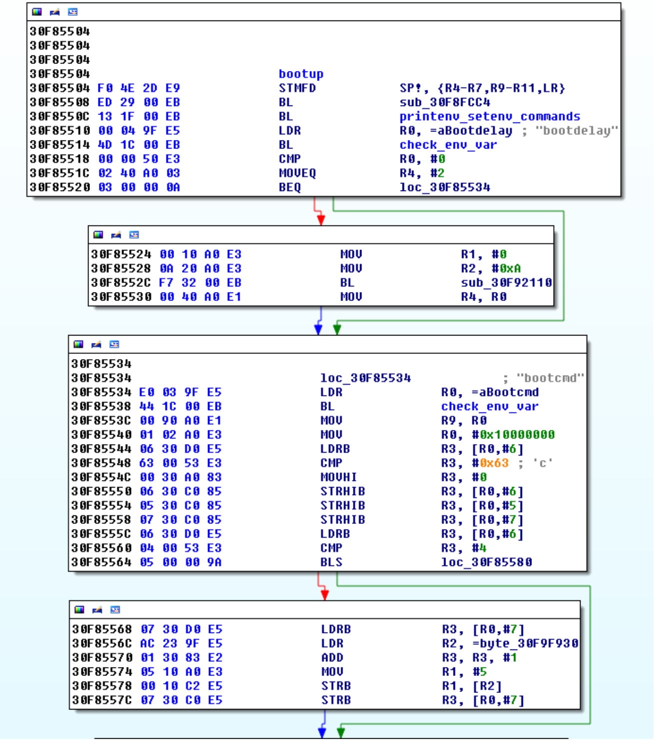
```
ROM:00000DF5 aNandBootloader DCB "Nand Bootloader(ADAM) 3.2.4",0xA
ROM:00000DF5                 DCB "    ",0
ROM:00000E16                 DCB    0
ROM:00000E17                 DCB    0
ROM:00000E18                 DCB  0xA
ROM:00000E19 aLoadingUBoot   DCB "Loading U-BOOT ",0xA
ROM:00000E19                 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

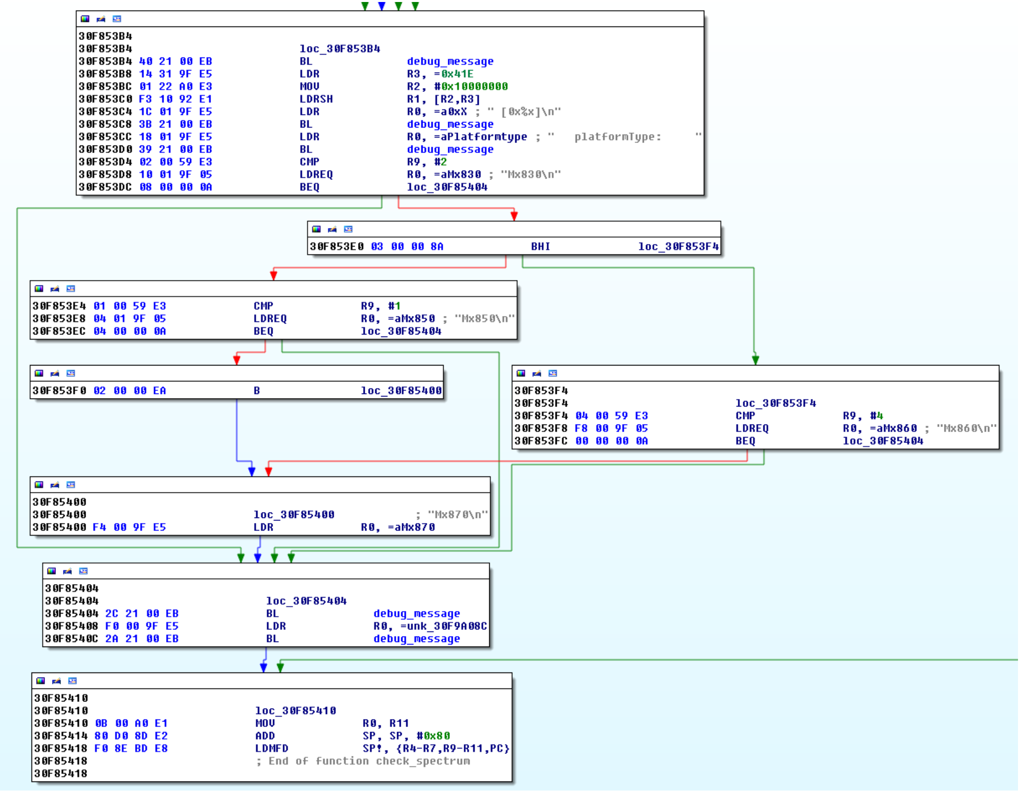
U-boot boot code



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

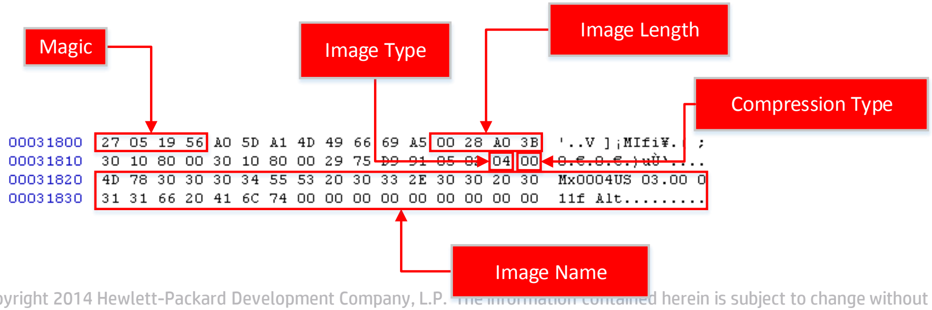
- The kernel image and actual file system are placed inside

Custom boot code



U-Boot image header structure

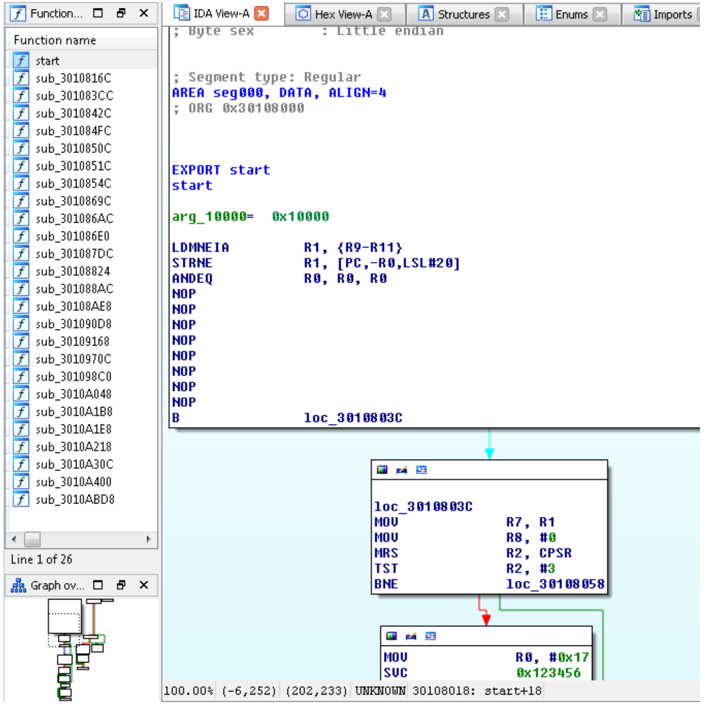
```
#define IH_MAGIC    0x27051956  /* Image Magic Number       */
#define IH_NMLEN            32  /* Image Name Length        */

/*
 * 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;
```



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 0x28A07B.

U-Boot image disassembly



IDA doesn't do well with multi-file images

Multi-file image

U-Boot header

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
```

1st image length

2nd image length

End of image length

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

```
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
```

DumpFlash – Extracting U-Boot images

```
C:\mat\Analysis\NAND Flash\DumpFlash>c:\python27\python DumpFlash.py -U
U-Boot Image found at block 0xc
Magic:  0x27051956
HCRC:   0xa05da14d
Time:   0x496669a5
Size:   0x28a03b
Load:   0x30108000
EP:     0x30108000
DCRC:   0x2975d991
OS:     0x5 (Linux)
Arch:   0x2 (ARM)
Type:   0x4 (Multi-File Image)
Comp:   0x0 (None)
Name:   Mx0004US 03.00 011f Alt

Found multi image of length 0xe9118
Found multi image of length 0x1a0f17
Extracting to U-Boot-00.dmp-00
Extracting to U-Boot-00.dmp-01
U-Boot Image found at block 0xcc
Magic:  0x27051956
HCRC:   0xbbaceac7
Time:   0x496669a6
Size:   0xe9118
Load:   0x30108000
EP:     0x30108000
DCRC:   0x2855374a
OS:     0x5 (Linux)
Arch:   0x2 (ARM)
Type:   0x2 (OS Kernel Image)
Comp:   0x0 (None)
Name:   Mx0004US 03.00 011f

Extracting to U-Boot-01.dmp-00
```

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 the *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 the 2nd U-Boot image

```
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
```

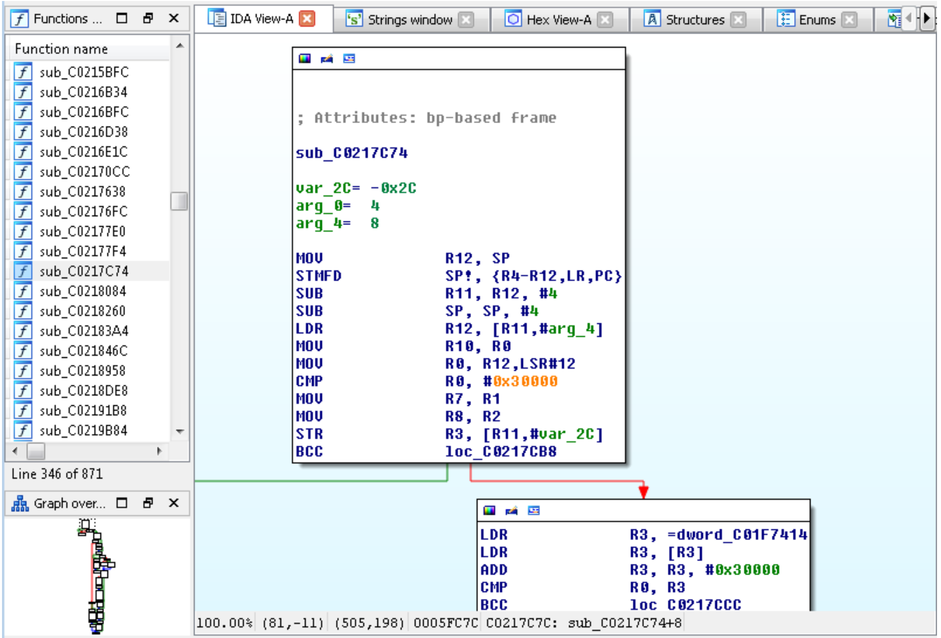
Start of gzipped kernel image

```
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 0  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.

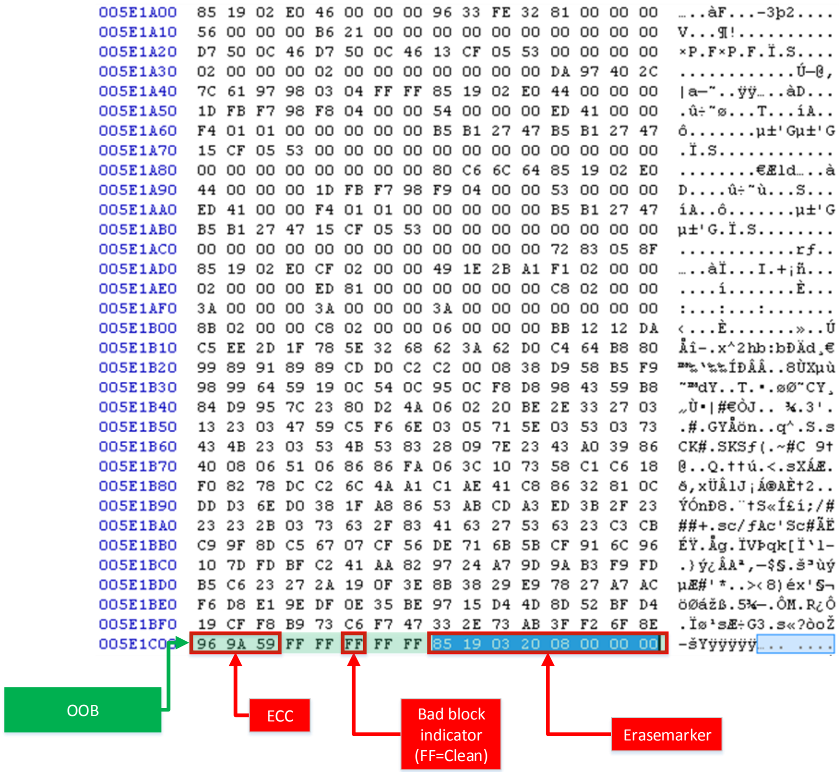
- Unfortunately the code shown by IDA is the bootstrapping code that decompresses following the gzipped kernel image
- To identify the start of the kernel image, search for the gzip image magic value (0x8b1f)

Kernel image disassembly



Demo: Extracting and analyzing U-Boot code

JFFS2 erase marker location from a page and spare column bytes



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
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
   dd if=/dev/mtdblock0 of=mtdblock0.dmp bs=512
2. Program memory at JFFS2 location
   - *python DumpFlash.py -w -b 135 0xffffffff -O mtdblock0.dmp*

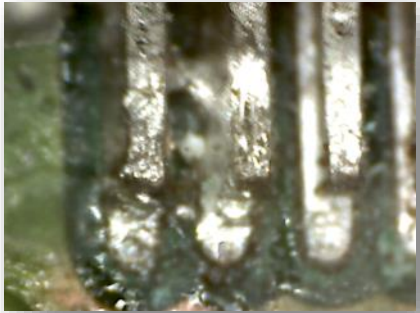
Demo: JFFS2 manipulation

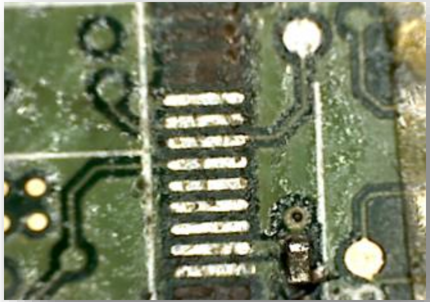
SMT Re-soldering

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

- The re-soldering process is not very different from standard SMT soldering
- SMT was originally developed for the automatic soldering of 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 *impossible*
- There are many different methods, but I placed the chip on the pin location and heated the pins using the soldering iron

Bridge & damaged pins





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

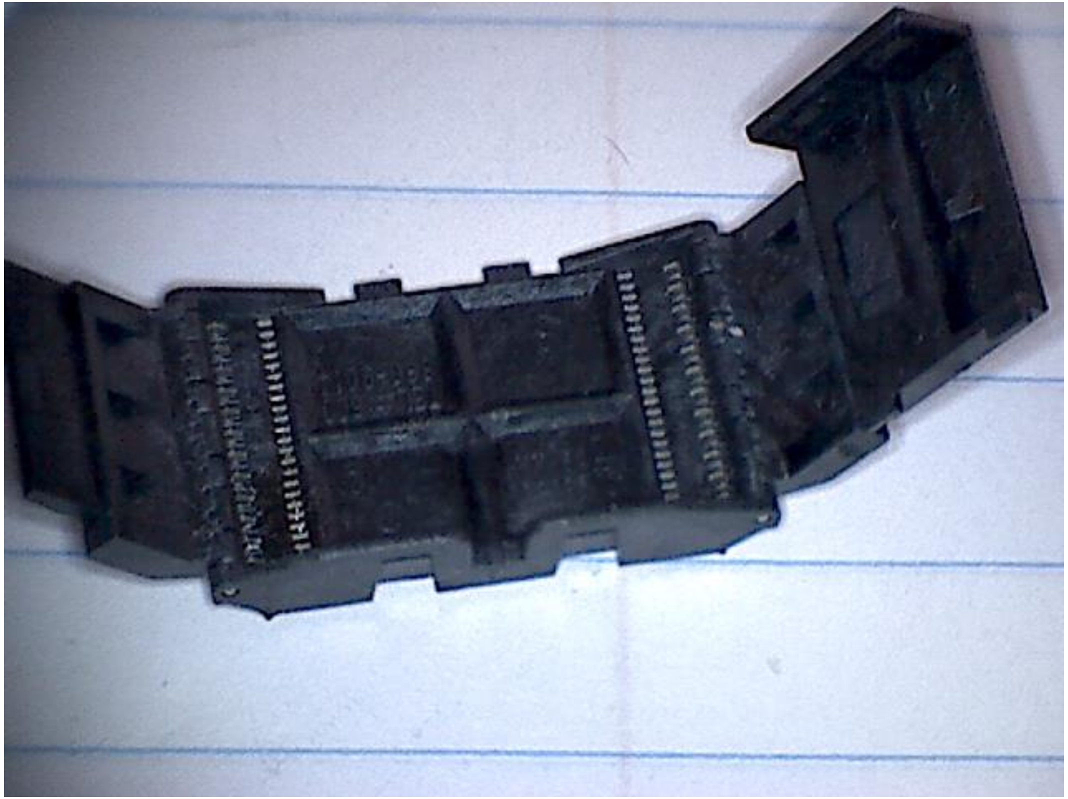
- The pitch for NAND flash TSOP48 model is 0.5 mm (which is extremely small). The solder can go over multiple pins and create shorts

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

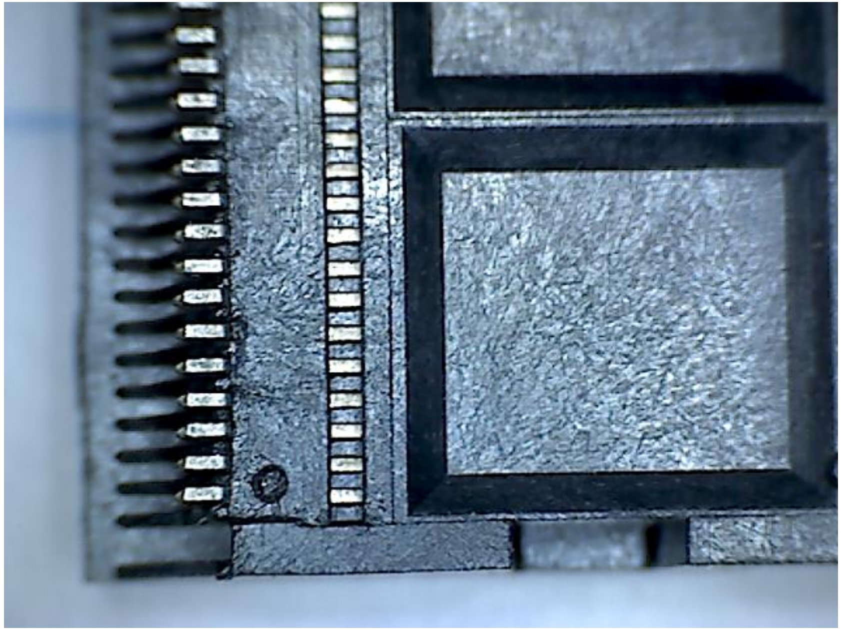
- Excessive heat is applied during de-soldering and it can damage the PCB board
- Be extra careful when you re-solder the chips!
- Luckily, with Flash memory, many pins are not used. If the damaged patterns are not used, then the chips operate normally
- Check with the chip datasheet to see if damaged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

Demo: Re-soldering

Making a development board – TSOP48 socket

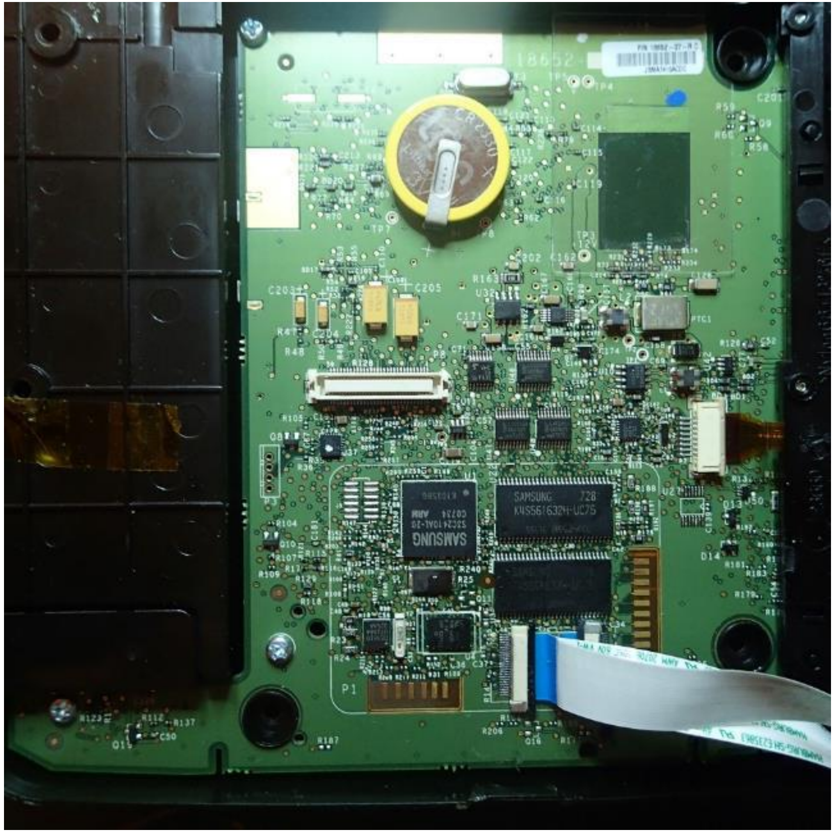


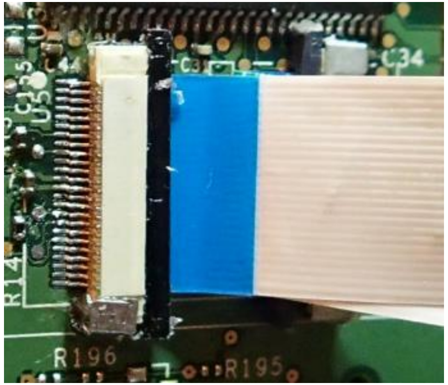
Making a development board – TSOP48 socket

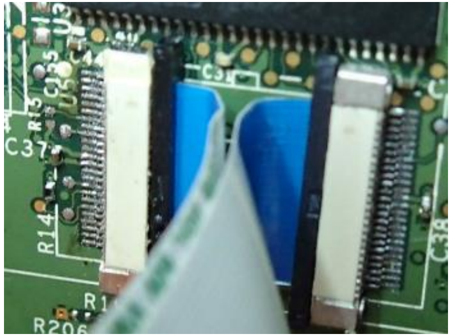


Tough to solder manually on the board

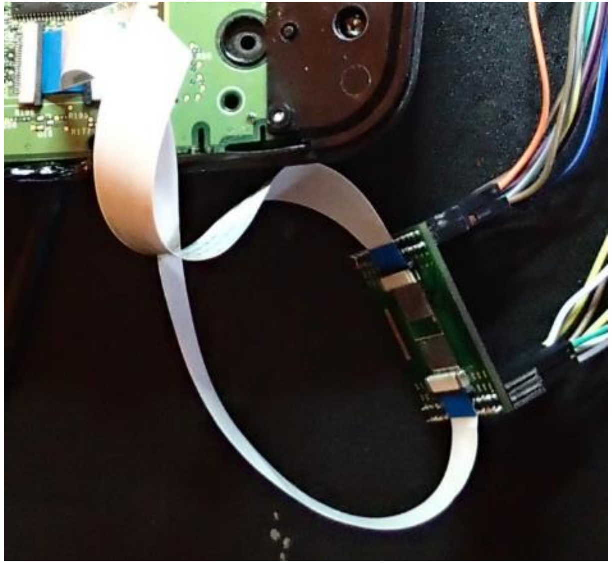
Making a development board – the hack option





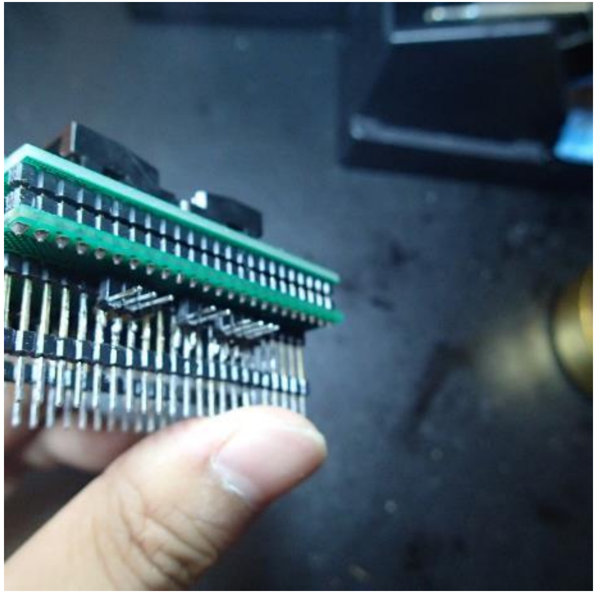


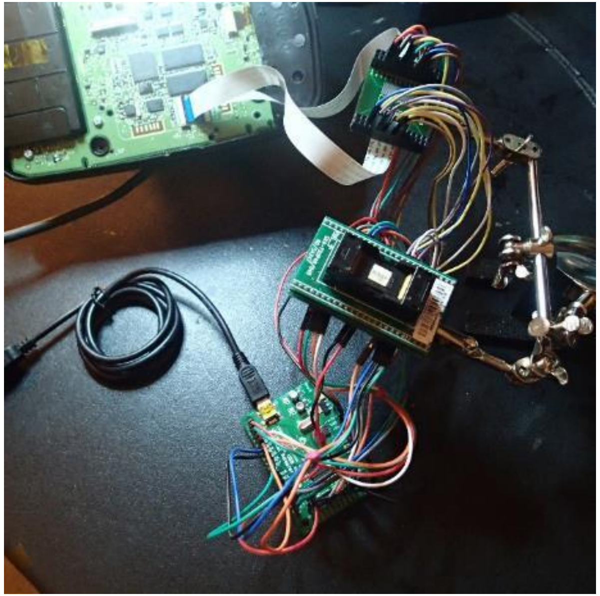
Flat flex cable to breakout board

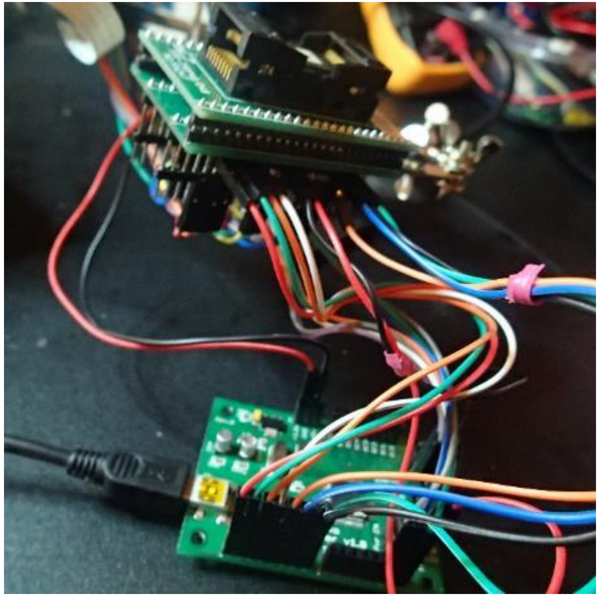


Making a development board – the hack option

Making a development board - bidirectional





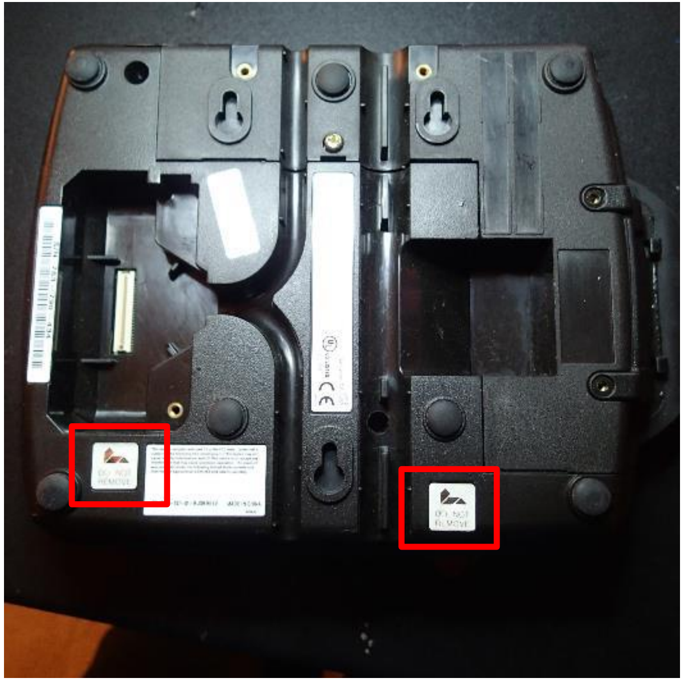


Tamper detection

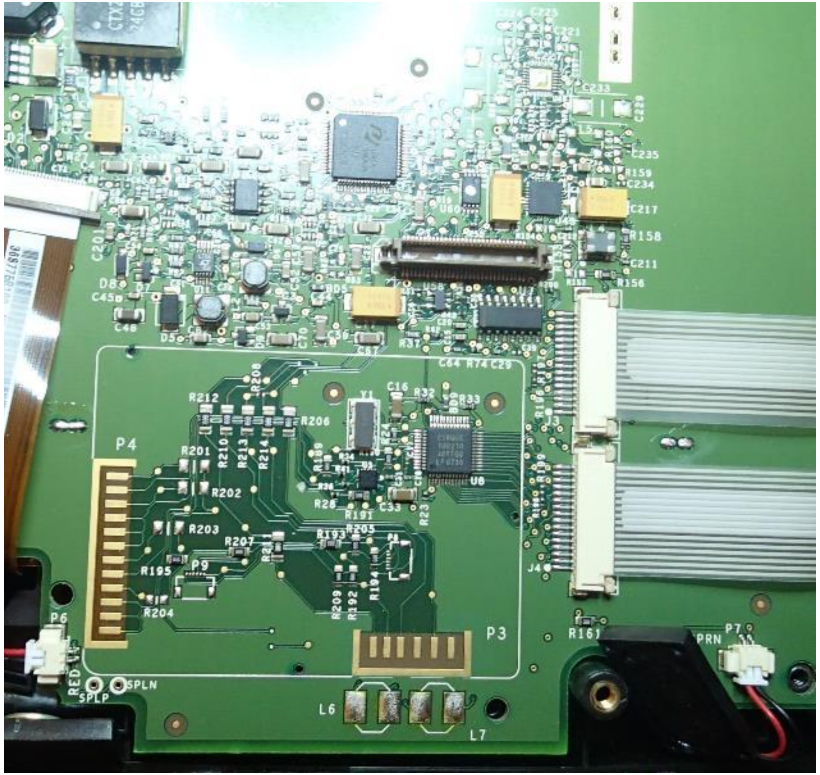
Advertisement from a modded POS device seller

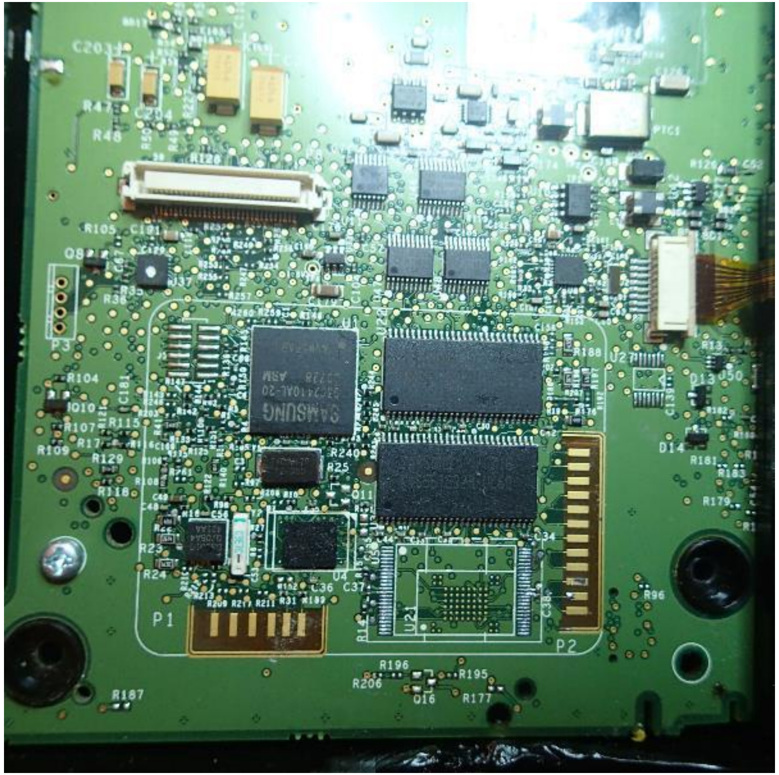
Source: dailymotion

Tamper Detection



Tamper detection

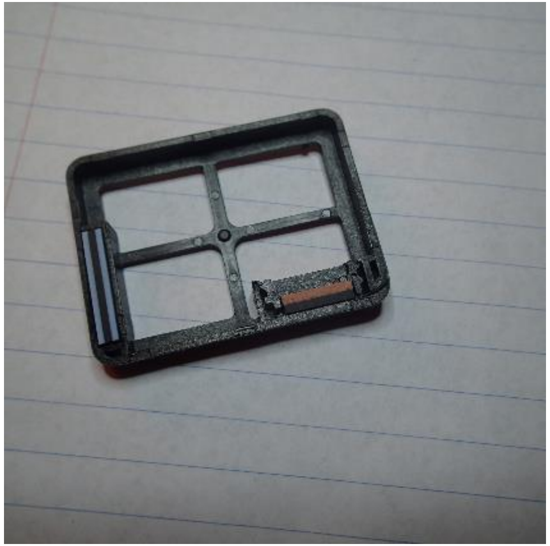


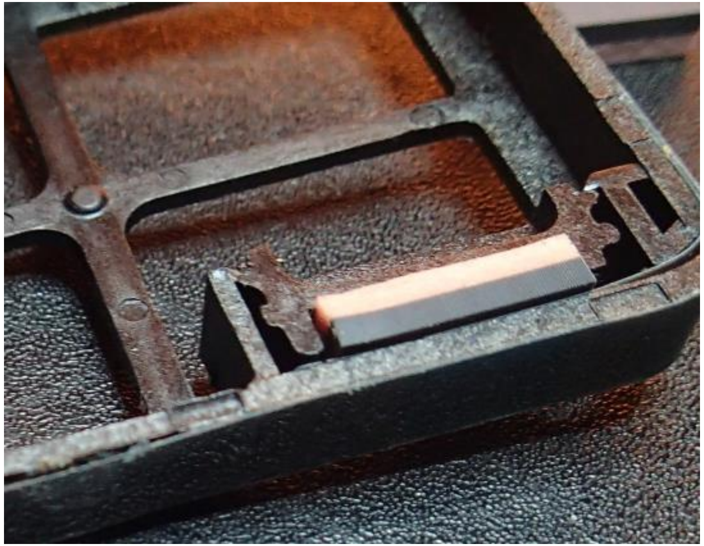


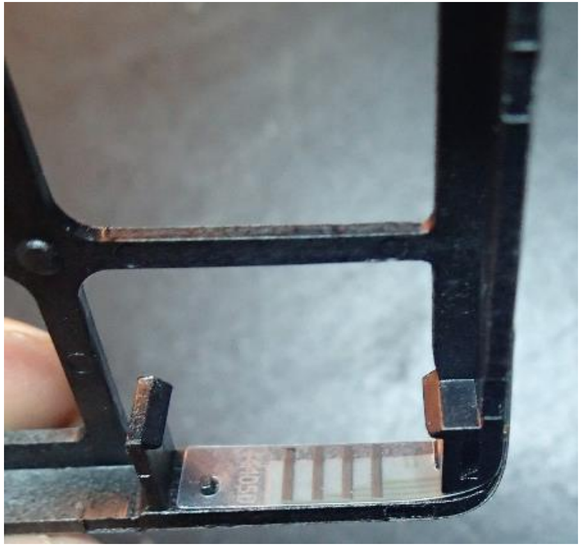
Tamper detection

84

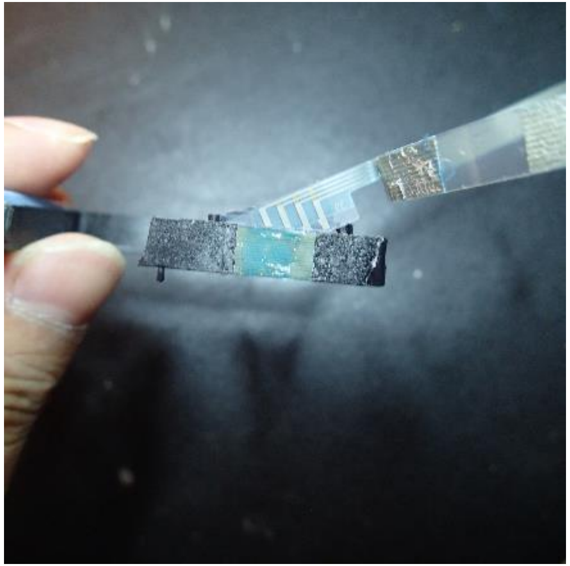
Tamper detection



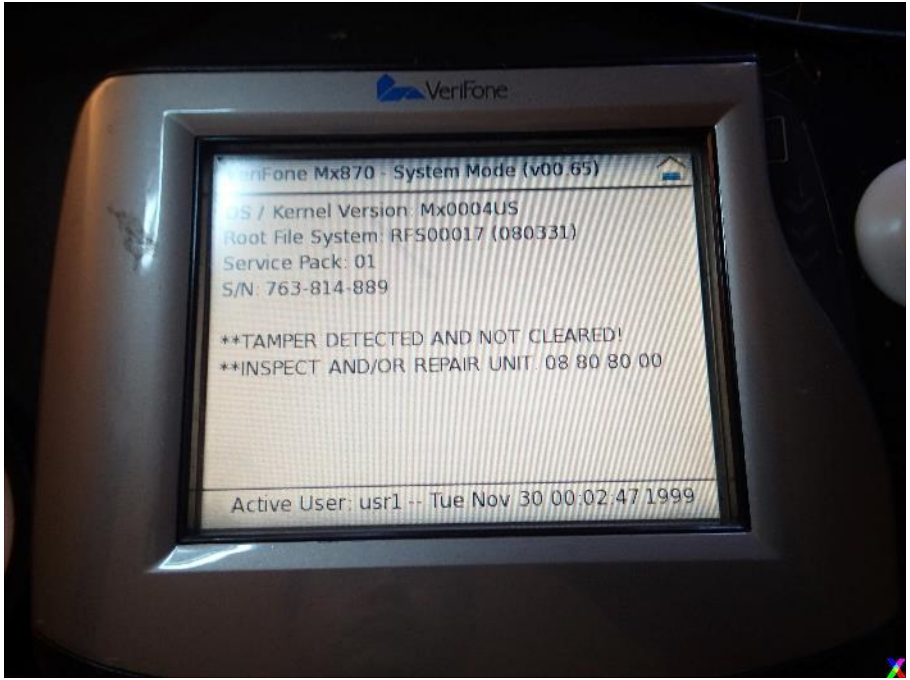




Tamper detection

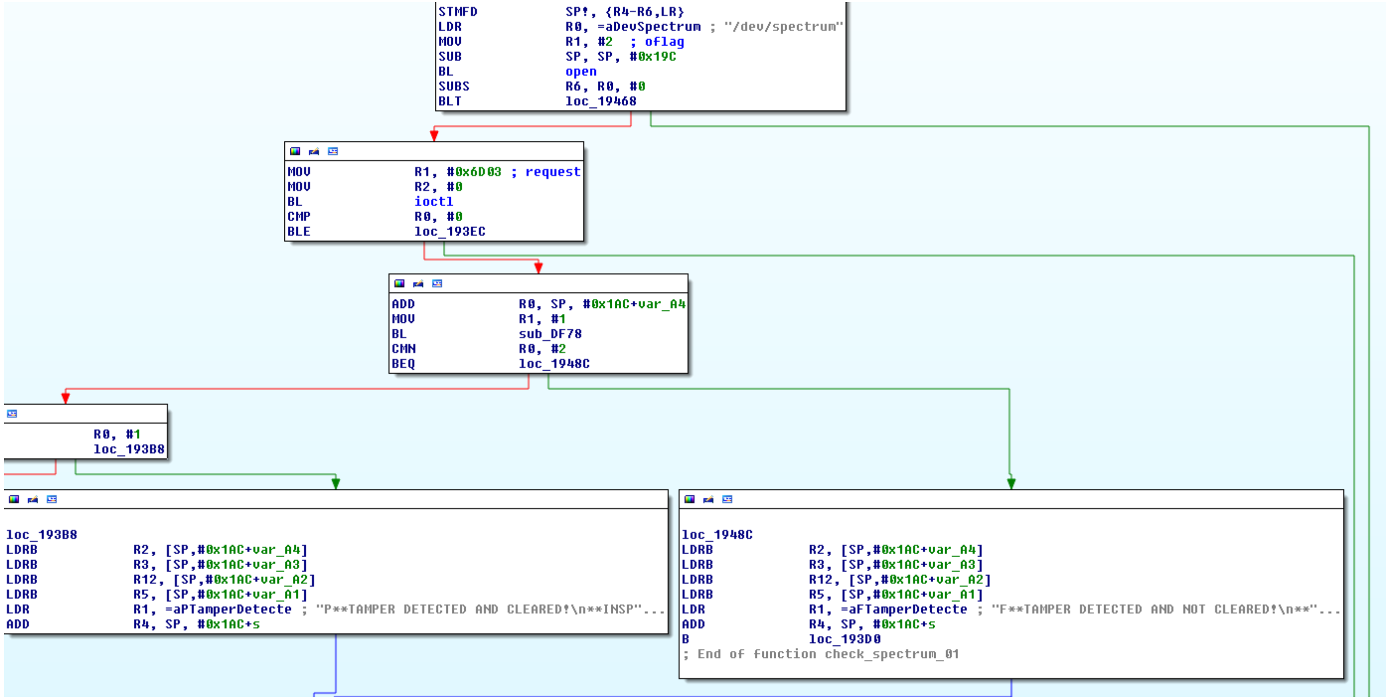


Tamper detection



87

Tamper detection



Does security by obscurity work?

The SDK is not publicly available

- To access the SDK, you need to have training (which is expensive)
- There is demand on the underground market for tampered devices

Does security by obscurity work here?

- Even without access to the SDK, it is possible to tamper with the device by directly accessing NAND flash memory
- The internal system is plain embedded Linux system, nothing special
- Do we really have a good way to detect tampered devices?

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
- USB stick low data investigations – do they clean up all remaining data when you format or erase the files?

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

Conclusion

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

Tamper detection can be sometimes security by obscurity:

- Easily defeated by modifying software components

Credits

Original design of NAND reader/writer

SpriteMod

NANDTool

SpriteMod and Bjoern Kerlers