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

<table>
<thead>
<tr>
<th>Commands</th>
<th>Operation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x90</td>
<td>Read</td>
<td>8bit address</td>
</tr>
<tr>
<td>0x91</td>
<td>Read</td>
<td>16bit address</td>
</tr>
<tr>
<td>0x92</td>
<td>Write</td>
<td>8bit address</td>
</tr>
<tr>
<td>0x93</td>
<td>Write</td>
<td>16bit address</td>
</tr>
<tr>
<td>0x82</td>
<td>Set</td>
<td>High byte (BDBUS6, 7)</td>
</tr>
<tr>
<td>0x83</td>
<td>Read</td>
<td>High byte (BDBUS6, 7)</td>
</tr>
</tbody>
</table>

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

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

R/B RE CE Vcc Vss CLE ALE WE WP

48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25

I/O7 I/O6 I/O5 I/O4 I/O3 I/O2 I/O1 I/O0

Vcc Vss
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
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

<table>
<thead>
<tr>
<th>FT2232H</th>
<th>Use</th>
<th>NAND Flash</th>
<th>Pin number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADBUS0</td>
<td>Bit0</td>
<td>I/00</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>ADBUS1</td>
<td>Bit1</td>
<td>I/01</td>
<td>30</td>
<td>DATA INPUT/OUTPUT</td>
</tr>
<tr>
<td>ADBUS2</td>
<td>Bit2</td>
<td>I/02</td>
<td>31</td>
<td>Input command, address and data</td>
</tr>
<tr>
<td>ADBUS3</td>
<td>Bit3</td>
<td>I/03</td>
<td>32</td>
<td>Output data during read operations</td>
</tr>
<tr>
<td>ADBUS4</td>
<td>Bit4</td>
<td>I/04</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>ADBUS5</td>
<td>Bit5</td>
<td>I/05</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>ADBUS6</td>
<td>Bit6</td>
<td>I/06</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>ADBUS7</td>
<td>Bit7</td>
<td>I/07</td>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>

**Low byte**
- 0x90,0x91,0x92,0x93 commands can be used to set values
# Data Control Lines

<table>
<thead>
<tr>
<th>FT2232H</th>
<th>Use</th>
<th>NAND Flash</th>
<th>Pin number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACBUS5</td>
<td>Bit13</td>
<td>WP</td>
<td>19</td>
<td>WRITE PROTECT&lt;br&gt;Write operations fail when this is not high</td>
</tr>
<tr>
<td>ACBUS6</td>
<td>Bit14</td>
<td>CLE</td>
<td>16</td>
<td>COMMAND LATCH ENABLE&lt;br&gt;When this is high, commands are latched into the command register through the I/O ports</td>
</tr>
<tr>
<td>ACBUS7</td>
<td>Bit15</td>
<td>ALE</td>
<td>17</td>
<td>ADDRESS LATCH ENABLE&lt;br&gt;When this is high, addresses are latched into the address registers</td>
</tr>
</tbody>
</table>

**High byte**

- 0x91, 0x93 can be used to set values
I/O and Strobe Lines

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

- 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

<table>
<thead>
<tr>
<th>Use</th>
<th>NAND Flash</th>
<th>Pin number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3v3</td>
<td>POWER</td>
<td>3v3</td>
<td>12</td>
</tr>
<tr>
<td>GND</td>
<td>GROUND</td>
<td>GND</td>
<td>13</td>
</tr>
<tr>
<td>3v3</td>
<td>POWER</td>
<td>3v3</td>
<td>36</td>
</tr>
<tr>
<td>GND</td>
<td>GROUND</td>
<td>GND</td>
<td>37</td>
</tr>
</tbody>
</table>

- 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)

<table>
<thead>
<tr>
<th>Function</th>
<th>1&lt;sup&gt;st&lt;/sup&gt; cycle</th>
<th>2&lt;sup&gt;nd&lt;/sup&gt; cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read 1</td>
<td>00h/01h</td>
<td>-</td>
</tr>
<tr>
<td>Read 2</td>
<td>50h</td>
<td>-</td>
</tr>
<tr>
<td>Read ID</td>
<td>90h</td>
<td>-</td>
</tr>
<tr>
<td>Page Program</td>
<td>80h</td>
<td>10h</td>
</tr>
<tr>
<td>Block Erase</td>
<td>60h</td>
<td>D0h</td>
</tr>
<tr>
<td>Read Status</td>
<td>70h</td>
<td></td>
</tr>
</tbody>
</table>

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

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.

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

<table>
<thead>
<tr>
<th>CLE</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALE</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R/B</td>
<td>1 (Ready)</td>
<td>R/B=0 (busy)</td>
<td>1 (Ready)</td>
</tr>
<tr>
<td>RE</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WE</td>
<td>Rising for each bytes</td>
<td>1 Rising 1</td>
<td></td>
</tr>
<tr>
<td>I/O0~7</td>
<td>80h Address Input A0–A7 A9–A25</td>
<td>Page + OOB data 10h</td>
<td>70h I/O0=Status</td>
</tr>
</tbody>
</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.writeFile(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.sendCmd(self.NAND_CMD_PAGEPROG)
err=self.Status()
if err&self.NAND_STATUS_FAIL:
    return err

self.sendCmd(self.NAND_CMD_READ008)
self.sendCmd(self.NAND_CMD_SEQIN)
self.waitReady()
self.sendAddr(pageno<<8,self.AddrCycles)
self.waitReady()
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)
After command and address are sent, WE fluctuates repeatedly to send bytes.
Working with a bare metal image
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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
</tr>
<tr>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
</tr>
<tr>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
</tbody>
</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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit7</td>
<td>Bit6</td>
<td>Bit5</td>
<td>Bit4</td>
</tr>
<tr>
<td>Bit7</td>
<td>Bit6</td>
<td>Bit5</td>
<td>Bit4</td>
</tr>
<tr>
<td>Bit7</td>
<td>Bit6</td>
<td>Bit5</td>
<td>Bit4</td>
</tr>
<tr>
<td>Bit7</td>
<td>Bit6</td>
<td>Bit5</td>
<td>Bit4</td>
</tr>
</tbody>
</table>

P8' checksum is calculated by XOR-ing all the bits in red.
Example - P16' calculation

- Uses bits from byte[0], byte[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

```c
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_
```

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

\[
\begin{align*}
\text{p1} &= \text{bit7} \oplus \text{bit5} \oplus \text{bit3} \oplus \text{bit1} \oplus \text{p1} \\
\text{p1}_- &= \text{bit6} \oplus \text{bit4} \oplus \text{bit2} \oplus \text{bit0} \oplus \text{p1}_- \\
\text{p2} &= \text{bit7} \oplus \text{bit6} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{p2} \\
\text{p2}_- &= \text{bit5} \oplus \text{bit4} \oplus \text{bit1} \oplus \text{bit0} \oplus \text{p2}_- \\
\text{p4} &= \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{p4} \\
\text{p4}_- &= \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0} \oplus \text{p4}_-
\end{align*}
\]
ECC calculation code

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).

```python
def IsBadBlock(self, block):
    for page in range(0,2,1):
        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

Start of a bad block
Bad block marker ≠ 0xFF
OOB

OOB

Bad block marker ≠ 0xFF
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

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
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
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

IDA doesn’t do well with multi-file images
Multi-file image

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

U-Boot header

Multi-file image

1st image length
2nd image length
End of image length

# mkimage -I dump-00031800-UBOOT.dmp
Image Name: M:00048US 03.00 Clif Alt
Created: Thu Jan 6 13:01:25 2009
Image Type: ARM Linux Multi-File Image (uncompressed)
Data Size: 2656459 Bytes = 2600.66 kB = 2.54 MB
Load Address: 3C018000
Entry Point: 3C018000
Contents:
  Image 0: 95464B Bytes = 932.27 kB = 0.91 MB
  Image 1: 1707799 Bytes = 1687.77 kB = 1.63 MB

© Copyright 2014 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice.
Mounting RAMdisk image

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

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

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
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 erasemakers inside the spare column of each page.
Mounting JFFS2 file system using a MTD

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

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

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
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 Implementation of Flash reader/writer software

- [https://github.com/ohjeongwook/DumpFlash/blob/master/FlashTool.py](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](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)


DumpJFFS2.py: JFFS2 parsing tool

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