Introduction

This technical note describes how to implement error correction code (ECC) in small page and large page Micron® single-level cell NAND Flash memory that can detect 2-bit errors and correct 1-bit errors per 256 or 512 bytes.

Refer to the SLC NAND Flash memory data sheets for further information, including the full list of NAND Flash memory devices covered by this technical note. Examples of software ECC are available from your local Micron distributor (see “References” on page 12).

Software ECC

When digital data is stored in nonvolatile memory, it is crucial to have a mechanism that can detect and correct a certain number of errors. Error correction code (ECC) encodes data in such a way that a decoder can identify and correct errors in the data.

Typically, data strings are encoded by adding a number of redundant bits to them. When the original data is reconstructed, a decoder examines the encoded message to check for any errors.

There are two basic types of ECC (see Figure 1 on page 2):

- **Block codes**: These codes are referred to as “n” and “k” codes. A block of k data bits is encoded to become a block of n bits called a code word. In block codes, code words do not have any dependency on previously encoded messages. NAND Flash memory devices typically use block codes.

- **Convolution codes**: These codes produce code words that depend on both the data message and a given number of previously encoded messages. The encoder changes state with every message processed. Typically, the length of the code word constant.

Block Codes

As previously described, block codes are referred to as n and k codes. A block of k data bits is encoded to become a block of n bits called a code word. A block code takes k data bits and computes (n - k) parity bits from the code generator matrix.

The block code family can be divided in linear and non-linear codes, as shown in Figure 1. Either type can be systematic. Most block codes are systematic in that the data bits remain unchanged, with the parity bits attached either to the front or to the back of the data sequence.
Systematic Codes

In systematic (linear or non-linear) block codes, each code word includes the exact data bits from the original message of length k, either followed or preceded by a separate group of check bits of length q (see Figure 2). The ratio \( k / (k + q) \) is called the code rate. Improving the quality of a code often means increasing its redundancy and, thus, reducing the code rate.

The set of all possible code words is called the code space.

Linear Codes

In linear block codes, every linear combination of valid code words (such as a binary sum) produces another valid code word. In all linear codes, the code words are longer than the data words on which they are based.

Micron NAND Flash memory devices use cyclic and Hamming linear codes.

Cyclic Codes

Cyclic codes are a type of linear code where every cyclic shift by a valid code word also yields a valid code word.
Hamming Codes

Hamming codes are the most widely used linear block codes. Typically, a Hamming code is defined as \((2n - 1, 2n - n - 1)\), where:
- \(n\) is equal to the number of overhead bits.
- \(2n - 1\) is equal to the block size.
- \(2n - n - 1\) is equal to the number of data bits in the block.

All Hamming codes can detect three errors and one correct one. Common Hamming code sizes are \((7, 4)\), \((15, 11)\), and \((31, 26)\). All have the same Hamming distance.

The Hamming distance and the Hamming weight are useful in encoding. When the Hamming distance is known, the capability of a code to detect and correct errors can be determined.

Hamming Distance

In continuous variables, distances are measured using euclidean concepts, such as lengths, angles, and vectors.

In binary encoding, the distance between two binary words is called the Hamming distance. It is the number of discrepancies between two binary sequences of the same size. The Hamming distance measures how different the binary objects are. For example, the Hamming distance between sequences 0011001 and 1010100 is 4.

The Hamming code minimum distance \(d_{\text{min}}\) is the minimum distance between all code word pairs.

Hamming Weight

The Hamming weight of a code scheme is the maximum number of 1s among valid code words.

Error Detection Capability

For a code where \(d_{\text{min}}\) is the Hamming distance between code words, the maximum number of error bits that can be detected is \(t = d_{\text{min}} - 1\). This means that 1-bit and 2-bit errors can be detected for a code where \(d_{\text{min}} = 3\).

Error Correction Capability

For a code where \(d_{\text{min}}\) is the Hamming distance between code words, the maximum number of error bits that can be corrected is \(t = (d_{\text{min}} - 1) \& 2\). This means that 1-bit errors can be corrected for a code where \(d_{\text{min}} = 3\).
ECC for Memory Devices

Common error correction capabilities for memory devices are:
- Single error correction (SEC) Hamming codes
- Single error correction/double error detection (SEC-DED) Hsiao codes
- Single error correction/double error detection/single byte error detection (SEC-DED-SBD) Reddy codes
- Single byte error correction/double byte error detection (SBC-DBD) finite field-based codes
- Double error correction/triple error detection (DEC-TED) Bose-Chaudhuri-Hocquenghem codes

ECC Generation

According to the Hamming ECC principle, a 22-bit ECC is generated to perform a 1-bit correction per 256 bytes. The Hamming ECC can be applied to data sizes of 1 byte, 8 bytes, 16 bytes, and so on.

For 528-byte/264-word page NAND devices, a Hamming ECC principle can be used that generates a 24-bit ECC per 512 bytes to perform a 2-bit detection and a 1-bit correction.

For 2112-byte/1056-word page NAND devices, the calculation can be done per 512 bytes, which means a 24-bit ECC per 4096 bits (exactly 3 bytes per 512 bytes). 2112 byte pages are divided into 512 byte (+ 16 byte spare) chunks (see Figure 3).

Figure 3: Large Page Divided Into Chunks

<table>
<thead>
<tr>
<th>2048 Bytes</th>
<th>64 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Area</td>
<td>Spare Area</td>
</tr>
<tr>
<td>1st Page Main Area</td>
<td>2nd Page Main Area</td>
</tr>
<tr>
<td>1st Page Spare</td>
<td>2nd Page Spare</td>
</tr>
</tbody>
</table>

The 24 ECC bits are arranged in three bytes (see Table 1).

Table 1: Assignment of Data Bits With ECC Code

<table>
<thead>
<tr>
<th>ECC</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecc0^1</td>
<td>LP07</td>
<td>LP06</td>
<td>LP05</td>
<td>LP04</td>
<td>LP03</td>
<td>LP02</td>
<td>LP01</td>
<td>LP00</td>
</tr>
<tr>
<td>Ecc1^2</td>
<td>LP15</td>
<td>LP14</td>
<td>LP13</td>
<td>LP12</td>
<td>LP11</td>
<td>LP10</td>
<td>LP09</td>
<td>LP08</td>
</tr>
<tr>
<td>Ecc2^3</td>
<td>CP5</td>
<td>CP4</td>
<td>CP3</td>
<td>CP2</td>
<td>CP1</td>
<td>CP0</td>
<td>LP17</td>
<td>LP16</td>
</tr>
</tbody>
</table>

Notes:
1. The first byte (Ecc0) contains line parity bits LP0–LP07.
2. The second byte (Ecc1) contains line parity bits LP08–LP15.
3. The third byte (Ecc2) contains column parity bits CP0–CP5, plus LP16 and LP17 generated only for a 512-byte input.
For every data byte in each page, 16 or 18 bits for line parity and 6 bits for column parity are generated according to the scheme shown in Figure 4.

**Figure 4: Parity Generation**

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 1</td>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
<tr>
<td>Byte 2</td>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
<tr>
<td>Byte 3</td>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Byte 252</td>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
<tr>
<td>Byte 253</td>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
<tr>
<td>Byte 254</td>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
<tr>
<td>Byte 255(^1)</td>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
</tbody>
</table>

**Notes:**

1. For 512-byte inputs, the highest byte number is 511.
2. For 512-byte inputs, additional XOR operations must be executed. Also, the last line parity bits calculated are LP16 and LP17 instead of LP14 and LP15, respectively. The column parity bits are unchanged.
ECC Generation Pseudo Code

The following pseudo code implements the parity generation shown in Figure 4 on page 5.

For \( i = 1 \) to \( 256^1 \)

begin

if \((i \& 0x01)\)

\[ \text{LP1} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0} \oplus \text{LP1}; \]

else

\[ \text{LP0} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0} \oplus \text{LP1}; \]

if \((i \& 0x02)\)

\[ \text{LP3} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

else

\[ \text{LP2} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0} \oplus \text{LP2}; \]

if \((i \& 0x04)\)

\[ \text{LP5} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

else

\[ \text{LP4} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

if \((i \& 0x08)\)

\[ \text{LP7} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

else

\[ \text{LP6} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

if \((i \& 0x10)\)

\[ \text{LP9} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

else

\[ \text{LP8} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

if \((i \& 0x20)\)

\[ \text{LP11} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

else

\[ \text{LP10} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]

if \((i \& 0x40)\)

\[ \text{LP13} = \text{bit7} \oplus \text{bit6} \oplus \text{bit5} \oplus \text{bit4} \oplus \text{bit3} \oplus \text{bit2} \oplus \text{bit1} \oplus \text{bit0}; \]
else
    LP12 = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP12; if (i & 0x80)
    LP15 = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP15;
else
    LP14 = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP14;
if(i & A0)
    LP17 = bit7 (Xor) bit6 (Xor) bit5 (Xor) bit4 (Xor) bit3 (Xor) bit2 (Xor) bit1 (Xor) bit0 (Xor) LP17
else
    LP16 = bit7 (Xor) bit6 (Xor) bit5 (Xor) bit4 (Xor) bit3 (Xor) bit2 (Xor) bit1 (Xor) bit0 (Xor) LP16
end

Notes: 1. For 512-byte inputs, the “i” range is from 1–512 bytes.
2. "⊕" indicates a bitwise XOR operation.
3. The last control “if(i & A0)” is executed only in the case of 512-byte inputs.

ECC Detection and Correction

ECC can detect the following:
- **No error**: The result of XOR is 0.
- **Correctable error**: The result of XOR is a code with 11 bits at 1. For 512-byte inputs, the generated ECC has 12 bits at 1.
- **ECC error**: The result of XOR has only 1 bit at 1. This means that the error is in the ECC area.
- **Non-correctable error**: The result of XOR provides all other results.

When the main area has a 1-bit error, each parity pair (for example, LP0 and LP1) is 1 and 0 or 0 and 1.

The fail bit address offset can be obtained by retrieving the following bits from the result of XOR:
- Byte address = (LP17, LP15, LP13, LP11, LP9, LP7, LP5, LP3, LP1). For 512-byte inputs, the resulting byte address is LP17.
- Bit address = (CP5, CP3, CP1)

When the NAND devices has more than two bit errors, the data cannot be corrected.
1. For 512-byte inputs, the generated ECC is has 12 bits at 1.
2. For 512-byte inputs, the generated ECC is 24 bits long.

### ECC Detection and Correction Pseudo Code

```plaintext
% Detect and correct a 1 bit error for 256 byte block
int ecc_check (data, stored_ecc, new_ecc)
begin
% Basic Error Detection phase
  ecc_xor[0] = new_ecc[0] ⊕ stored_ecc[0];
  if ((ecc_xor[0] or ecc_xor[1] or ecc_xor[2]) == 0)
    begin
      return 0; % No errors
    end
  else
    begin
% Counts the bit number
      bit_count = BitCount(ecc_xor);
      if (bit_count == 11)
        begin
% Set the bit address
          bit_address = (ecc_xor[2] >> 3) and 0x01 or
        end
    end
end
```
(ecc_xor[2] >> 4) and 0x02 or
(ecc_xor[2] >> 5) and 0x04;

byte_address = (ecc_xor[0] >> 1) and 0x01 or
(ecc_xor[0] >> 2) and 0x02 or
(ecc_xor[0] >> 3) and 0x04 or
(ecc_xor[0] >> 4) and 0x08 or
(ecc_xor[1] << 3) and 0x10 or
(ecc_xor[1] << 2) and 0x20 or
(ecc_xor[1] << 1) and 0x40 or
(ecc_xor[1] and 0x80);

% Correct bit error in the data
data[byte_address]=data[byte_address] ⊕ (0x01<<bit_address);
return 1;
end
else if (bit_count == 1)
begin

% ECC Code Error Correction
stored_ecc[0] = new_ecc[0];
stored_ecc[1] = new_ecc[1];
stored_ecc[2] = new_ecc[2];
return 2;
end
else
begin

% Uncorrectable Error
return -1;
end
end
end

Note: "⊕" indicates bitwise XOR.
ECC Software Interface

Micron supplies a software implementation of the algorithm in the c1823.zip file (see “References” on page 12). The software interface implementation consists of two public functions:

```c
void make_ecc(ubyte *ecc_code, const ubyte *data);
```
that calculates the 3-byte ECC, and

```c
eccdiff_t ecc_check(ubyte *data, ubyte *stored_ecc, ubyte *new_ecc);
```
that detects and corrects a 1-bit error.

Where:

- `Stored_ecc` is the ECC stored in the NAND Flash memory.
- `New_ecc` is the ECC code generated on the data page read operation.
ECC Hardware Code Generation and Correction - Verilog Model

The hardware code generation and correction schematics are shown in Figure 6. The function details are provided in Table 2 on page 11. The interface for the hardware model is: module ecc_top(Data, Index, ECC, EN1, RST, ecc_stored, reg_state, byte_address, bit_address, EN2, CLK).

Figure 6: Hardware Code Generation and Correction

Table 2: Module Ecc_top Hardware Interface

<table>
<thead>
<tr>
<th>Interface</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>input [7:0]</td>
<td>A single byte.</td>
</tr>
<tr>
<td>Index</td>
<td>input [7:0]</td>
<td>The current data position in the page (0-255).</td>
</tr>
<tr>
<td>ECC</td>
<td>output [23:0]</td>
<td>Three bytes of ECC.</td>
</tr>
<tr>
<td>EN1</td>
<td>input</td>
<td>Enable for the ecc_gen module.</td>
</tr>
<tr>
<td>RST</td>
<td>input</td>
<td>Reset.</td>
</tr>
<tr>
<td>ecc_stored</td>
<td>input [23:0]</td>
<td>The ECC value stored in the NAND Flash memory.</td>
</tr>
<tr>
<td>reg_state</td>
<td>output [2:0]</td>
<td>A register that shows the state of the ecc_check module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>define No_Error 000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>define Correctable_Error 001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>define Ecc_Error 010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>define Non_Correctable_Error 100</td>
</tr>
<tr>
<td>byte_address</td>
<td>output [7:0]</td>
<td>Byte address of the defective data bit.</td>
</tr>
<tr>
<td>bit_address</td>
<td>output [2:0]</td>
<td>Bit address of the defective data bit.</td>
</tr>
<tr>
<td>EN2</td>
<td>input</td>
<td>Enable for the ecc_check module.</td>
</tr>
<tr>
<td>CLK</td>
<td>input</td>
<td>Clock.</td>
</tr>
</tbody>
</table>
Conclusion

The implementation of ECC in devices used for data storage is recommended. Hamming-based block codes are commonly used ECC for NAND Flash memory devices. By using a Hamming ECC in Micron NAND Flash memory devices, 2-bit errors can be detected and 1-bit errors corrected. This minimizes possible errors and helps to extend the lifetime of the memory device.

References

- NANDxxx-A single-level cell small page NAND Flash memory family data sheets
- NANDxxx-B single-level cell large page NAND Flash memory family data sheets
- Examples of software ECC, which are available from your local distributor