Error Correction Code in NAND Flash Memories

This Application Note describes how to implement an Error Correction Code (ECC), in ST NAND Flash memories, which can detect 2-bit errors and correct 1-bit errors per 256 Bytes.

This Application Note should be downloaded with the *c1823.zip* file.

INTRODUCTION

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

Studies on Error Correction Codes started in the late 1940's with the works of Shannon and Hamming, and since then thousands of papers have been published on the subject.

Usually 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 Error Correction Codes (see Figure 1.):

- **Block Codes**, which are referred to as **(n, k)** codes. A block of **k** data bits is encoded to become a block of **n** bits called a **code word**.
- **Convolution Codes**, where the code words produced depend on both the data message and a given number of previously encoded messages. The encoder changes state with every message processed. The length of the code word is usually constant.

NAND Flash memories typically use Block Codes.

TABLE OF CONTENTS

BLOCK CODES

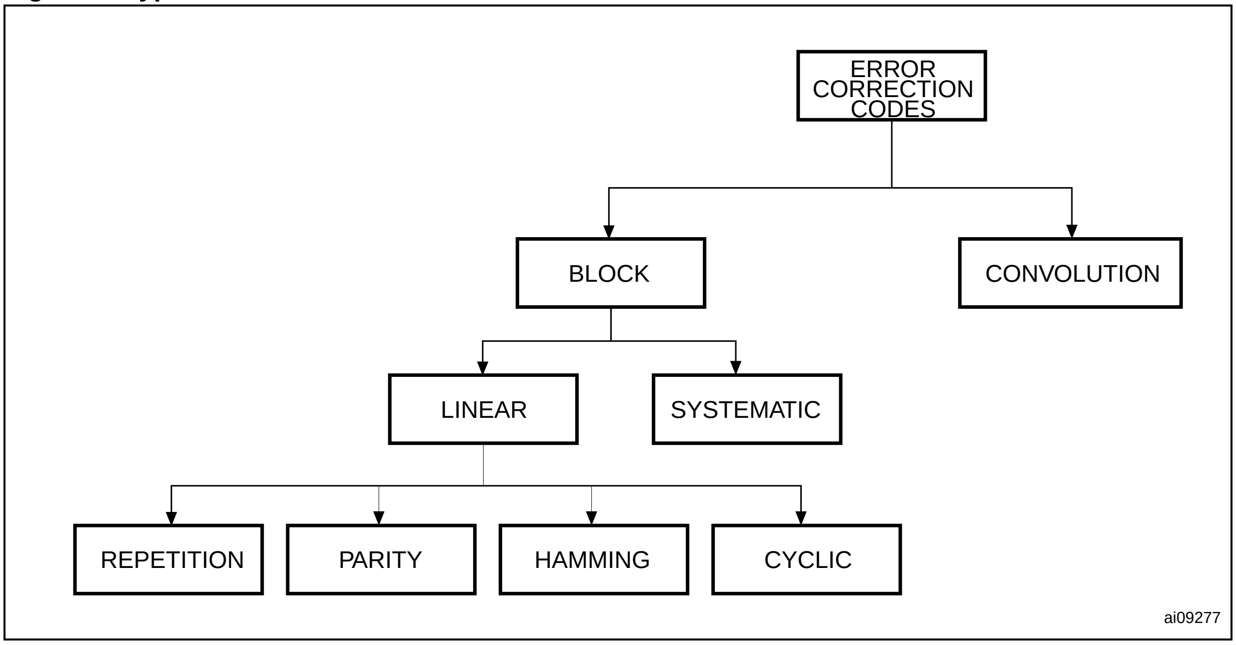
The Block Code family can be divided up into (see Figure 1.):

- **Linear Codes**, where every linear combination of valid code words (such as a binary sum) produces another valid code word. Examples of linear codes are:
  - **Cyclic Codes**, where every cyclic shift by a valid code word also yields a valid code word
  - **Hamming Codes**, which are able to detect three errors and correct one.
- **Systematic Codes**, where 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.).

In all cases the code words are longer than the data words on which they are based.

Block codes are referred to as **(n, k)** codes. A block of **k** data bits is encoded into a block of **n** bits called a **code word**. The code takes **k** data bits and computes **(n –k)** parity bits from the code generator matrix. 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.

Figure 1. Types of Error Correction Codes



Systematic Codes

In Systematic Codes, the number of original data bits is **k**, the number of additional check bits is **q**, and 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 (see Figure 2.).

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

Example: The code space of 4-bit code sequences consists of 16 code words. Only ten of them are used, these are the valid words. Code words not used or unassigned are invalid code words, they should never be stored. So if one of them is retrieved from the memory the encoder will assume that an error has occurred.

Figure 3., Example Code Word Space, shows the list of all the code words corresponding to a particular code size, 4 bits in this case.

Figure 2. Systematic Codes

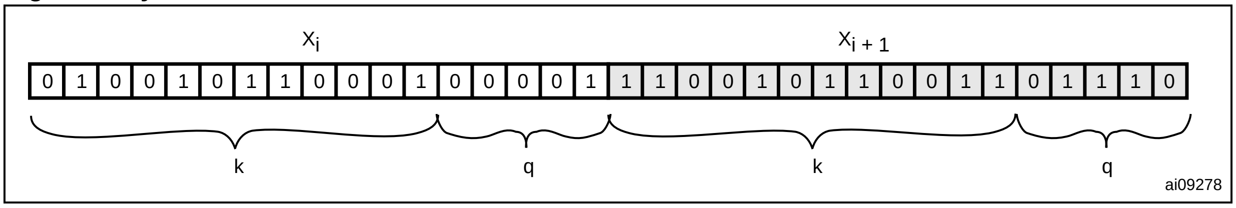
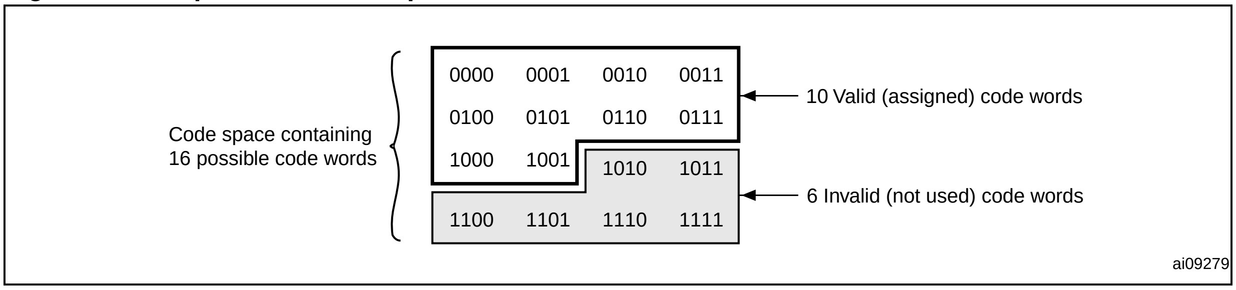


Figure 3. Example Code Word Space



Hamming Codes

Hamming Codes are the most widely used Linear Block codes.

A Hamming code is usually defined as **($2n-1$, $2n-n-1$)**, where n is the number of overhead bits, **$2n-1$** the block size and **$2n-n-1$** the number of data bits in the block. All Hamming Codes are able to detect three errors and correct one. Common Hamming Code sizes are (7, 4), (15,11) and (31, 26). They all have the same Hamming distance.

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

Hamming Weight. The Hamming Weight of a code scheme is the maximum number of 1's among valid code words. In the example illustrated in Figure 3., the Hamming Weight is 3.

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 Hammering distance measures how different binary objects are. For example the Hamming distance between sequences 0011001 and 1010100 is 4.

Error Detection Capability. For a code where **d_{min}** is the Hamming distance between code words, the maximum number of error bits that can be detected is:

$$t = d_{min} - 1$$

This means that for a code where **$d_{min} = 3$**, one-bit and two-bit errors can be detected.

Error Correction Capability. For a code where **d_{min}** is the Hamming distance between code words, the maximum number of error bits that can be corrected is:

$$t = [(d_{min} - 1)/2]$$

This means that for a code where **$d_{min} = 3$**, one bit errors can be corrected.

ECC FOR MEMORIES

The following list shows common Error Correction capabilities for memory devices:

- SEC (single error correction) Hamming codes.
- SEC-DED (single error correction double error detection) Hsiao codes.
- SEC-DED-SBD (single error correction double error detection single byte error detection) Reddy codes.
- SBC-DBD (single byte error correction double byte error detection) finite field based codes.
- DEC-TED (double error correction triple error detection) Bose-Chaudhuri-Hocquenghem codes.

ECC Generation

According to the Hamming ECC principle, a 22 bit ECC is generated in order 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, etc.

In the case of the 528 Bytes/264 Word Page NAND Flash memories, the calculation has to be done per 256 Bytes, which means a 22 bit ECC per 2048 bits (approximately 3 Bytes per 256 Bytes). Of the 22 ECC bits, 16 bits are for line parity and 6 bits are for column parity.

Table 1 shows how the 22 ECC bits are arranged in three Bytes.

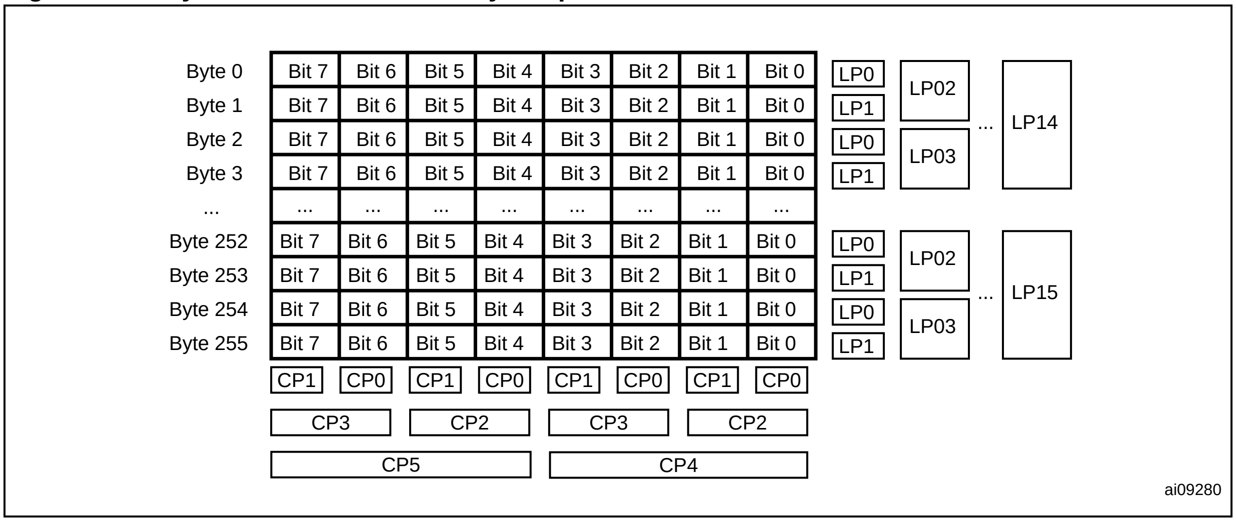
For every data Byte in each page, 16 bits for line parity and 6 bits for column parity are generated according to the scheme shown in Figure 4.

Table 1. Assignment of Data Bits with ECC Code

ECC	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Ecc0(1)	LP07	LP06	LP05	LP04	LP03	LP02	LP01	LP00
Ecc1(2)	LP15	LP14	LP13	LP12	LP11	LP10	LP09	LP08
Ecc2(3)	CP5	CP4	CP3	CP2	CP1	CP0	1	1

Note: 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 two “1”s for Bit 0 and Bit 1.

Figure 4. Parity Generation for a 256 Byte Input



Pseudo Code for ECC Generation

The following code implements the Parity Generation shown in Figure 4.

```
For i = 1 to 256
begin
   if (i & 0x01)
      LP1 = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP1;
else
  LP0  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP1;
if (i & 0x02)
  LP3  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP3;
else
  LP2  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP2;
if (i & 0x04)
  LP5  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP5;
else
  LP4  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP4;
if (i & 0x08)
  LP7  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP7;
else
  LP6  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP6;
if (i & 0x10)
  LP9  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP9;
else
  LP8  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP8;
if (i & 0x20)
  LP11  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP11;
else
LP10  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP10;
       if (i & 0x40)
  LP13  = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ LP13;
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;

CP0 = bit6 ⊕ bit4 ⊕ bit2 ⊕ bit0 ⊕ CP0;
CP1 = bit7 ⊕ bit5 ⊕ bit3 ⊕ bit1 ⊕ CP1;
CP2 = bit5 ⊕ bit4 ⊕ bit1 ⊕ bit0 ⊕ CP2;
CP3 = bit7 ⊕ bit6 ⊕ bit3 ⊕ bit2 ⊕ CP3
CP4 = bit3 ⊕ bit2 ⊕ bit1 ⊕ bit0 ⊕ CP4
CP5 = bit7 ⊕ bit6 ⊕ bit5 ⊕ bit4 ⊕ CP5
end
```

Where ⊕ means bitwise XOR operation.

ECC Detection and Correction

Error Correction Codes can detect four different type of results:

- **No Error** - the result of XOR is ‘0’.
- **Correctable Error -** the result of XOR is a code with 11 bits at ‘1’.
- **ECC Error -** the result of XOR has only 1 bit at ‘1’, ECC error means that the error is in the ECC area.
- **Non-correctable Error** - the result of XOR provides random data

When the main area has a 1 bit error, each parity pair (e.g. LP0 & LP1) is ‘1’ and ‘0’ or ‘0’ and ‘1’.

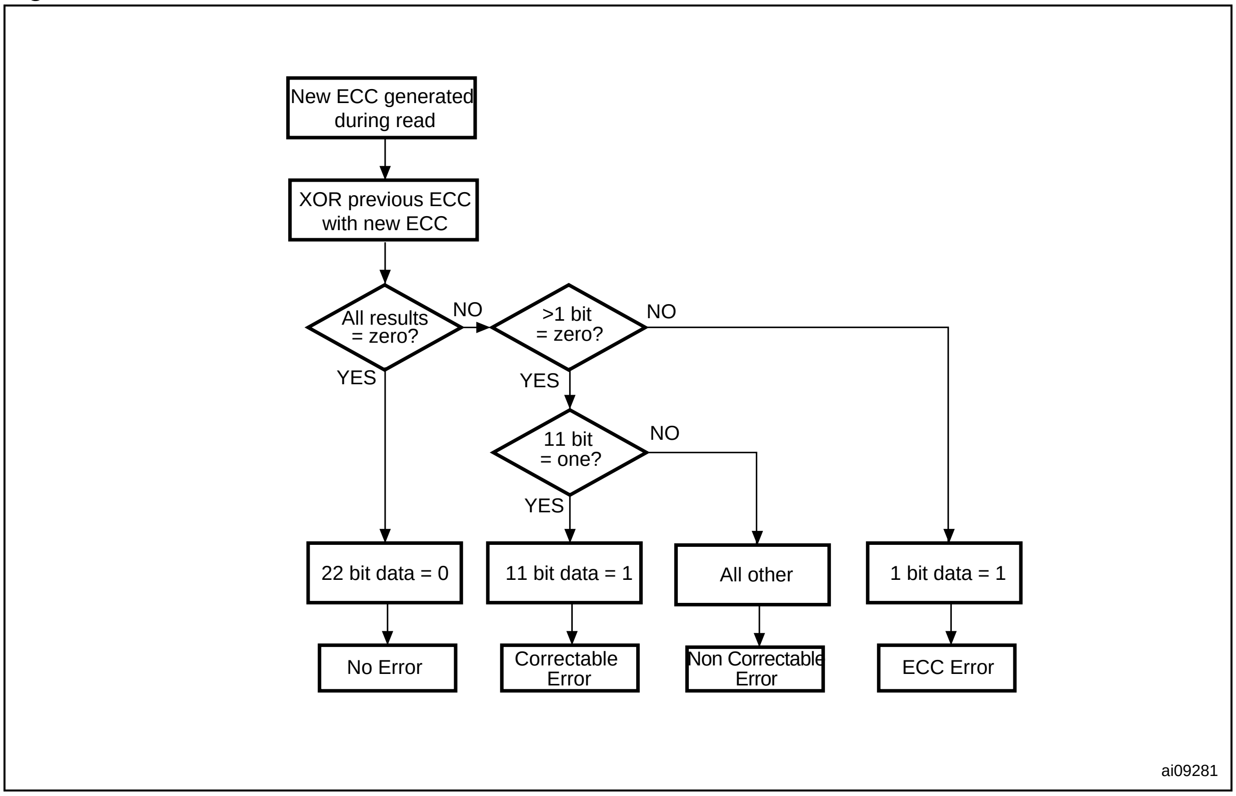
The fail bit address offset can be obtained by retrieving the following bits from the result of XOR:

Byte Address = (LP15,LP13,LP11,LP9,LP7,LP5,LP3,LP1)

Bit Address = (CP5, CP3,CP1)

When the NAND Flash has more than 2 bit errors, the data cannot be corrected.

Figure 5. ECC Detection Flowchart



Pseudo Code for ECC Detection and Correction

```
% 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];
ecc_xor[1] = new_ecc[1] ⊕ stored_ecc[1];
ecc_xor[2] = new_ecc[2] ⊕ stored_ecc[2];

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
                        (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
```

Where ⊕ means bitwise XOR.

ECC SOFTWARE INTERFACE

ST supplies a software implementation of the algorithm (*c1823.zip* file, see REFERENCES). The software interface implementation consists of two public functions:

```
void ecc_gen (const uchar *data, uchar *ecc_code)
```

that calculates the 3 Byte ECC code for 256 Bytes (half a page)

and

```
int ecc_check (uchar *data, uchar *stored_ecc, uchar *new_ecc)
```

that detects and corrects a 1 bit error for 256 Bytes (half a page)

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

Figure 6 shows the schematics of Hardware Code Generation and Correction and Table 2 gives the details of the functions. 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

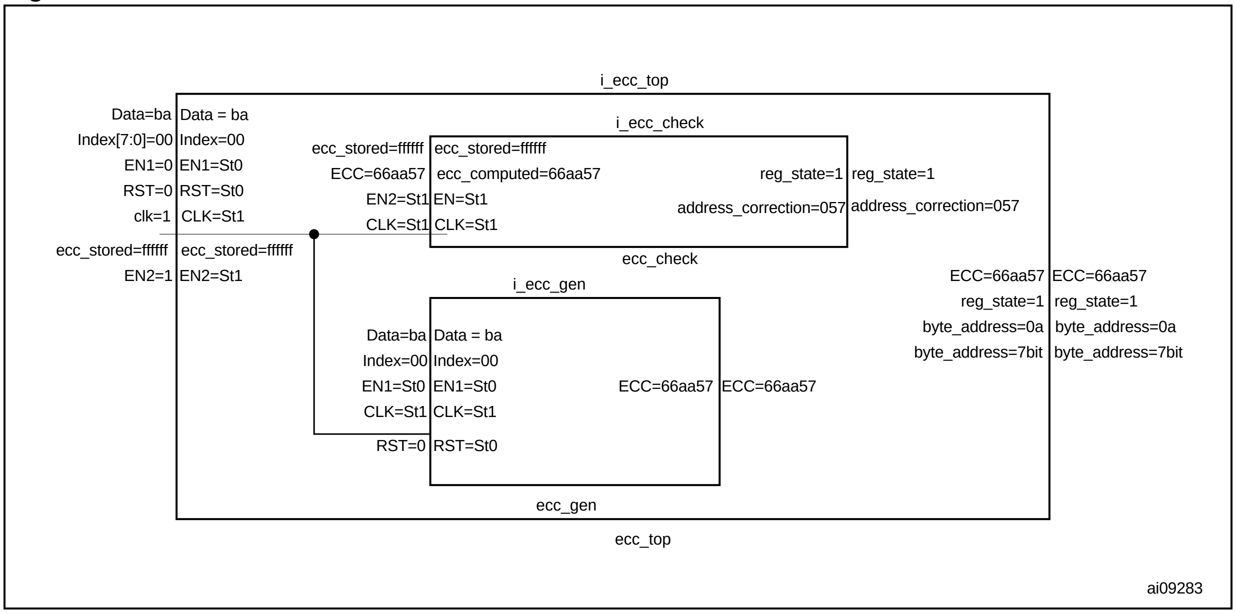


Table 2. Module Ecc_top Hardware Interface

Interface	Type	Description	
Data	input [7:0]	a single Byte	
Index	input [7:0]	the current data position in the page (0-255)	
ECC	output [23:0]	three Bytes of ECC	
EN1	input	enable for the ecc_gen module	
RST	input	Reset	
ecc_stored	input [23:0]	the ECC value stored in the NAND Flash memory	
reg_state	output [2:0]	a register that shows the state of the ecc_check module	define No_Error 000
			define Correctable_Error 001
			define Ecc_Error 010
			define Non_Correctable_Error 100
byte_address	output [7:0]	Byte address of the defective data bit	
bit_address	output [2:0]	bit address of the defective data bit	
EN2	input	enable for the ecc_check module	
CLK	input	Clock	

CONCLUSION

It is recommended to implement an Error Correction Code in devices used for data storage. Hamming based Block Codes are the most commonly used ECCs for NAND Flash memories. By using a Hamming ECC in ST NAND Flash memories, two bit errors can be detected and one bit errors corrected. This minimizes possible errors and helps to extend the lifetime of the memory.

REFERENCES

■ NAND128-A, NAND256-A, NAND512-A, NAND01G-A, 528 Byte/ 264 Word Page datasheet

■ c1823.zip file containing ECC software to download from www.st.com

REVISION HISTORY

Table 3. Document Revision History

Date	Version	Revision Details
11-May-2004	1.0	First issue.

If you have any questions or suggestions concerning the matters raised in this document, please send them to the following electronic mail addresses:

ask.memory@st.com (*for general enquiries*)

Please remember to include your name, company, location, telephone number and fax number.