NAND Error Correction Codes Introduction

Introduction

With the continued scaling of NAND flash technology and advancements in multi-level cell technology, flash memory-based storage has gained widespread use in systems ranging from mobile platforms to enterprise servers. However, the robustness of NAND flash cells has become an increasing concern, especially with nanometer-regime process geometries. Error correcting codes (ECC) are used with NAND flash memory to detect and correct bit errors that may occur with the memory. NAND flash memory bit error rates increase with the number of program/erase cycles and the scaling of technology. Stronger error correcting code (ECC) can be used to support higher raw bit error rates and enhance the lifespan of NAND flash memory.

The goal of this technical note is to explain the ECC algorithms employed on the Macronix single-level cell (SLC) NAND flash memory technology. ECC can be implemented using the NAND controller hardware or by using firmware/software, depending on the architecture of the main system. In Macronix NAND series products, ECC requirements vary with technology. In general, as the process technology geometry continues to shrink, the ECC requirement increases. In this technical note, we will introduce the ECC algorithms such as Hamming code, Reed-Solomon code and BCH code.

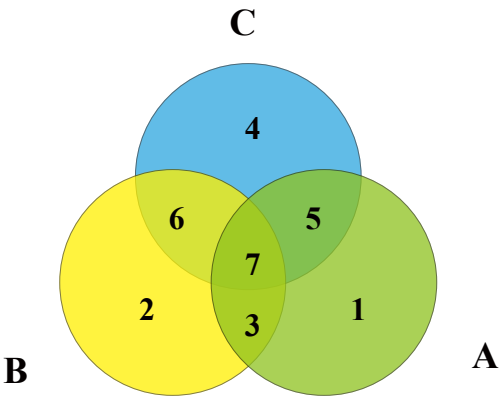
Error Correct Code: The Fundamental Operation

ECC is formed from serial mathematic polynomial terms combined together to encode and decode specific covered data in order to correct limited errors occurring within the NAND cell array. The error correction code algorithms are named after their inventors. For example, Hamming code was developed by Richard Hamming at Bell Telephone Laboratories in 1950. Reed-Solomon code was invented by Irving S. Reed and Gustave Solomon in 1960. BCH code was developed by Alexis Hocquenghem in 1959, and independently in 1960 by Raj Bose and D. K. Ray-Chaudhuri. The acronym BCH comprises the initials of these three inventors' last names.

There are two basic types of ECC: Block codes and Convolution codes. Block codes are a form of forward error correction (FEC), also known as a channel code, which converts messages into specific codes and sends the information as a block of data with a predetermined length. Larger blocks of data make it easier for the receiving computer to decode the information and correct errors that occur during the transfer. Convolution codes contain ***n***, ***k***, and ***m*** factors which can be implemented with a ***k***-bit input, and an ***n***-bit output, in which the output depends on the current input and the ***m*** preceding inputs. A convolution code is generated by passing the information sequence to be transmitted through a linear finite-state shift register (LFSR).

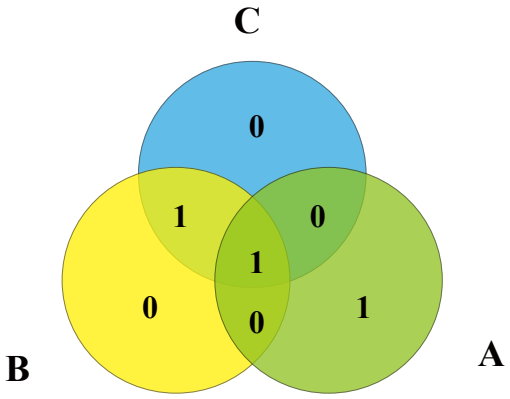
Here is a simple example to explain how ECC works. Suppose we have 3 regions (A, B and C) which intersect each other as shown in the Venn diagram in *"Figure 1."*.

Figure 1.



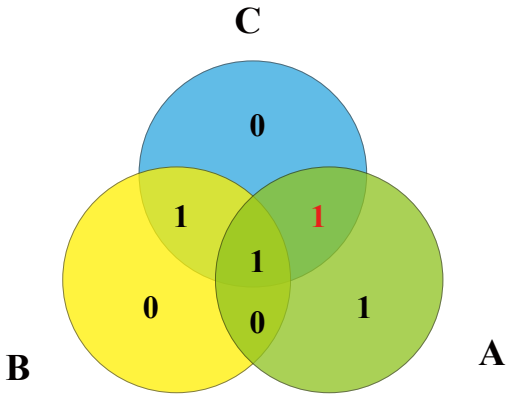
First, we number all of the different areas created by the intersecting circular regions from 1 to 7. Let each of the numbered areas represent a single bit of information. Now assume we have a 4-bit set of data “1001” to be transferred and we place the bits into areas 7, 3, 5, and 6 sequentially. Then we add bits into areas 1, 2, and 4 in order to make all 3 circle regions contain an even numbers of 1s. The three extra bits are ‘100’ as shown in *"Figure 2."*.

Figure 2.



Now we have our code composed of the original message “1001” with the extra “100” “parity” bits attached. If after data transmission we have one incorrect “failing” bit in area 5, then the regions associated with circle A and circle C will now contain an odd number of 1s as shown in *"Figure 3."*. Therefore, we know circle A and circle C contain a data error, but circle B is OK. That means all areas in circle B (2, 3, 6, 7) are correct. Only area 5 is intersected in circle A and circle C but not in circle B. In this case, we know area 5 is wrong.

Figure 3.



From the above simple example, we transferred data “1001” along with the extra ECC parity bits “100”. In this case, the three extra ECC parity bits allow us to do single bit error correction.

Hamming Codes

Now let’s look at Hamming code, invented by Richard Hamming in 1950. Hamming code can correct 1-bit errors and detect 2-bit errors. The above example is Hamming (7, 4), indicating 7 bits of code are transmitted containing 4 bits of data and 3 bits of parity. In mathematical terms, Hamming codes are a class of binary linear codes. To better understand Hamming Codes, we first need to recall modulo2 addition and multiplication (their truth tables are shown in *"Figure 4."*).

Figure 4.

	0	1
0	0	1
1	1	0

Addition

	0	1
0	0	0
1	0	1

Multiplication

The 3 circular regions of the previous section can be realized with the following equations:

$$\begin{cases} X_1 + X_3 + X_5 + X_7 = 0 \\ X_2 + X_3 + X_6 + X_7 = 0 \\ X_4 + X_5 + X_6 + X_7 = 0 \end{cases} \quad \text{where } X_i = \{0, 1\}$$

The limitations of 3 circles become mathematical linear equations. Based on the coefficients of those linear equations, we can re-write the equations as a matrix *H*.

$$H = \begin{pmatrix} 1\,0\,1\,0\,1\,0\,1 \\ 0\,1\,1\,0\,0\,1\,1 \\ 0\,0\,0\,1\,1\,1\,1 \end{pmatrix}$$

Assume *C* is all sets of codewords in this case. All of *C* is the null space of matrix H. The matrix rank is 3. The dimension of *C* is total terms (7) minus matrix rank (3). That is 7-3=4. So we have 2^4 = 16 possible codewords in *C* sets of this case.

Syndrome of Hamming Codes

Now we know all sets of codewords are constructed with the null space of matrix *H*. When a codeword *x* passes through a channel, it could generate an error called error vector *e*. Note that *y* = *x* + *e* where *x* is the original codeword. The error vector *e* will cause an error bit if the relative bit is 1, since 1 + 1 = 0 in a binary arithmetic operation. This enables us to know the position of the errors.

The Syndrome allows us to identify the error that is only relative to error vector *e*. Let's take the same case as an example. For *y* = *x* + *e*, if y = (1011100) = (1001100) + (0010000), then the syndrome is

$$H_y^T = \begin{pmatrix} 1\,0\,1\,0\,1\,0\,1 \\ 0\,1\,1\,0\,0\,1\,1 \\ 0\,0\,0\,1\,1\,1\,1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

From syndrome (1 1 0), we know that bit-3 failed. If there is no error, then the syndrome is (0 0 0). Moreover, if bit-1 failed, the syndrome is (1 0 0). The syndrome indicates the bit position of an error in the codeword (see *"Table 1."*), allowing us to correct the error on the fly during the ECC function.

Table 1.

Failed Bit	Syndrome	Received Data
No error	(0 0 0)	(1 0 0 1 1 0 0)
Bit7	(1 1 1)	(1 0 0 1 1 0 1)
Bit6	(0 1 1)	(1 0 0 1 1 1 0)
Bit5	(1 0 1)	(1 0 0 1 0 0 0)
Bit4	(0 0 1)	(1 0 0 0 1 0 0)
Bit3	(1 1 0)	(1 0 1 1 1 0 0)
Bit2	(0 1 0)	(1 1 0 1 1 0 0)
Bit1	(1 0 0)	(0 0 0 1 1 0 0)

Hamming code can be used with the Macronix MX30LF1208AA and MX30LF1G08AA series SLC NAND products, which require 1-bit error correction per 528 bytes (4224 bits). As shown in *"Table 2."*, *m* parity bits can cover $2^m - 1$ (data plus parity) bits.

Table 2.

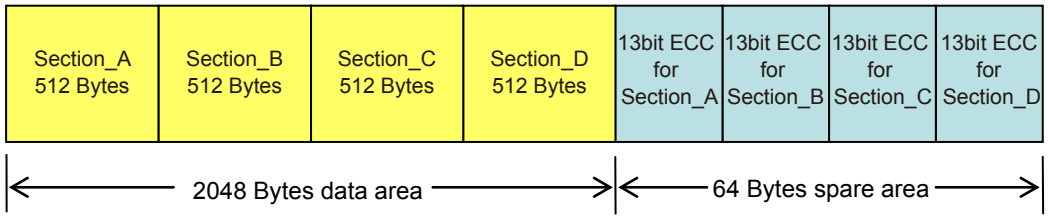
Parity Bits	Data Bits	Total Bits
2	1	3
3	4	7
4	11	15
5	26	31
…	…	…
m	$2^m - m - 1$	$2^m - 1$

Thus, if we use the above to calculate the number of parity bits for the Macronix 1-bit ECC NAND flash:

4224 bits (528 bytes) ≤ $2^m - 1$ → m = 13 parity bits

Therefore, MX30LF1208AA and MX30LF1G08AA need at least 13 parity bits for Hamming code per 528 bytes (512 data bytes + 16 spare bytes). The 13 parity bits can be stored in the spare area as shown in *"Figure 5."*.

Figure 5.

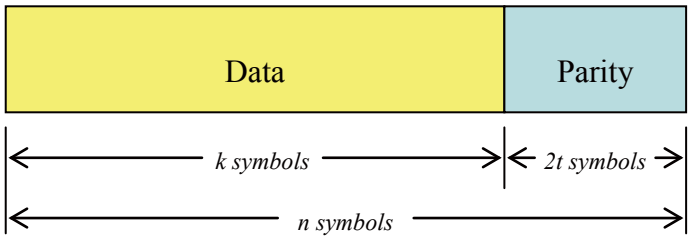


Reed-Solomon Codes

When error rates increase with the advancement of NAND flash technology, the likelihood of multiple bit errors within a single page also increases. To cope with this situation, Hamming codes are used over smaller data blocks, or designers turn to stronger error correction methods to ensure data integrity. Many engineers are familiar with Reed-Solomon codes from communications and other storage applications. The algorithm can also be used for NAND flash error correction. A typical Reed-Solomon code implementation splits the 512B correction blocks into multi-bit symbols (typically 8 or 9 bits) and provides N-symbol corrections. If any number of bits within the symbol are corrupted, the Reed-Solomon code will correct the entire symbol. Since Reed-Solomon codes correct symbols, its strength is in applications where errors tend to be clumped together.

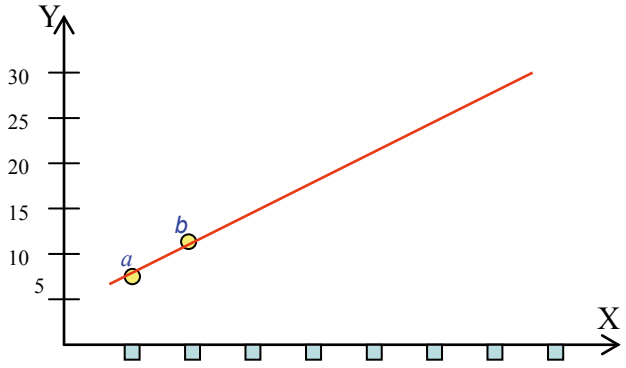
Reed-Solomon code is non-binary code created by Reed and Solomon in Lincoln Laboratory of MIT. Basically, Reed-Solomon code is (***n, k***) code with ***s***-bit symbols. This means that the encoder takes ***k*** data symbols of ***s*** bits each and adds parity symbols to make an ***n***-symbol codeword. There are ***n-k*** parity symbols of ***s*** bits each. A Reed-Solomon decoder can correct up to ***t*** symbols that contain errors in a codeword, where 2***t*** = ***n-k***. *"Figure 6"* shows a typical Reed-Solomon codeword. It is known as a Systematic code because the data is left unchanged and the parity symbols are appended in spare area of a page in NAND device.

Figure 6



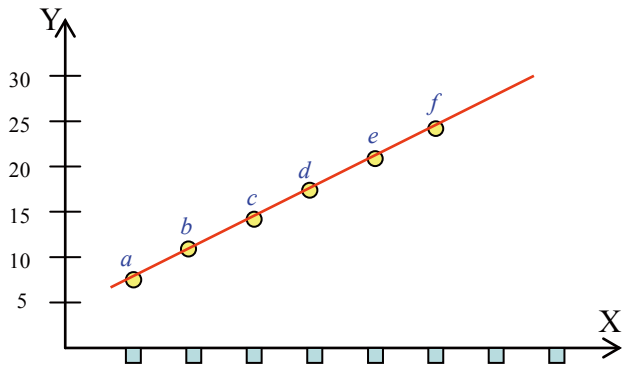
The following example demonstrates the Reed-Solomon concept. We take the s-bit symbol as a number and operate in finite field ***GF***(2^s). The minimum distance of Reed-Solomon code is **n – k + 1**. We assume there are 2 symbols (*a* and *b*) needed to be transferred in a channel. The 2 symbols are not original real numbers in finite field operation. However, in order to assist in understanding the principle, we will draw a straight line to explain how Reed-Solomon code works. *"Figure 7."* plots the 2 symbols and the resulting straight line.

Figure 7.



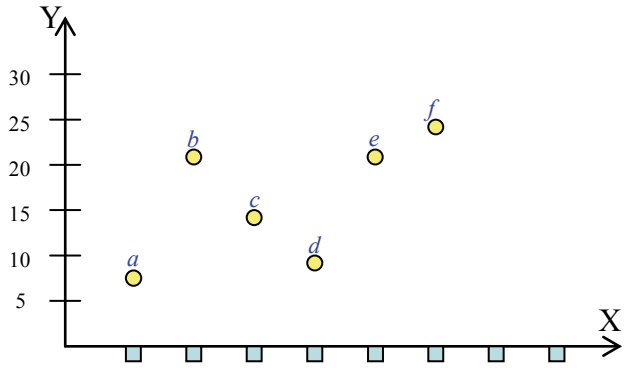
The Y-axis represents the symbol location and the X-axis represents the equal distance between symbols. Two symbols need to be added for each additional error we want to be able to correct. If we need to correct 2 errors during the transferring operation, the Reed-Solomon code requires 4 more symbols appended to the 2 symbols *a* and *b* as shown in *"Figure 8."*.

Figure 8.



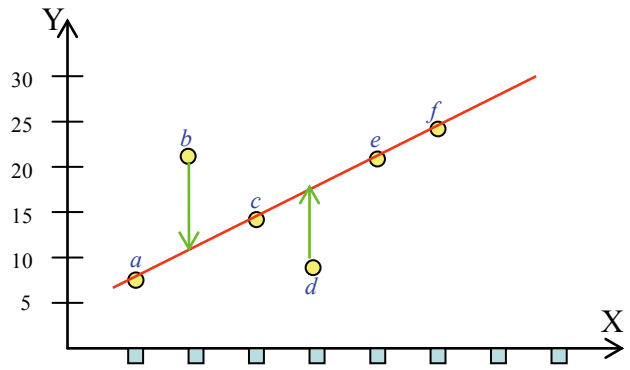
In this case, ***n*** – ***k*** = 2***t*** = 4 can do 2 symbols error correction with distance = 5 (***n*** – ***k*** + 1). Now there are a total of 6 symbols to be transferred in a channel. If there is no error, the 6 symbols will lie along the same straight line. However, if errors occur during the data transfer operation, symbols may be offset from the straight line as shown in *"Figure 9."*.

Figure 9.



To detect the errors, a straight line is drawn connecting the largest number of symbols (in this case to symbols ***a***, ***c***, ***e*** and ***f***). Symbols ***b*** and ***d*** are not on the straight line as shown in Figure 10. Therefore, we know the errors are on symbols ***b*** and ***d***. Reed-Solomon code will detect the 2 error symbols and correct them by pulling them back to the straight line shown in *"Figure 10."*.

Figure 10.

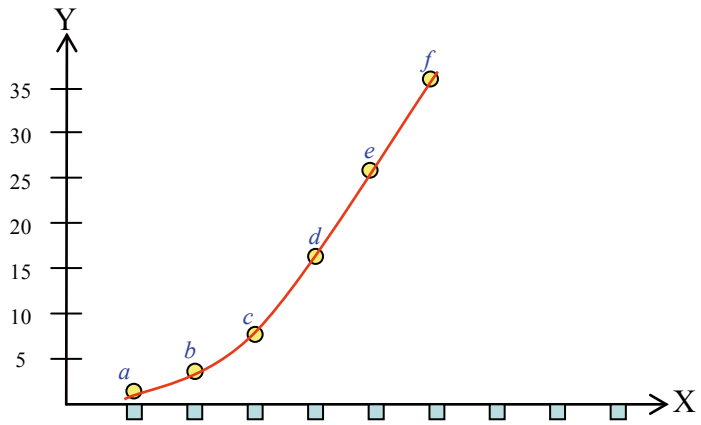


In the above example we wanted to transfer 2 symbols and be able to correct 2 errors. Now assume we want to transfer 3 symbols in a channel and be able to correct 2 errors. The previous example showed 2 symbols defining a straight line. In the current example, we have 3 symbols that can establish a curve of degree 2 such as a parabola. If we still want to correct 2 symbol errors, then we will need to add 4 more symbols to meet Reed-Solomon code requirements such that $n - k = 2t = 4$ with $t = 2$ correction capability. Now we have a total of 7 symbols (which lie on the parabola) that need to be transferred in a channel (*"Figure 11."*).

The correction method is the same as that of previous example. As we did before, we align the parabola to the largest numbers of symbols. For those symbols which do not lie on the parabola, a correction routine will pull them back to the parabola. If there are ***m*** symbols of data to be transferred in a channel, we will find a polynomial of degree ***m*** – 1.

From the polynomial, we can find 2***t*** symbols (parity symbols) to make ***t*** symbols corrections. The generation of 2***t*** parity symbols from m data symbols is known as encoding. Symbol correction is accomplished by decoding the data and parity symbols to identify those symbols which are not on the defined polynomial and must be corrected back to fit in the polynomial. As the polynomial defining the finite field ***GF*** becomes more complex, it may not be easily expressed as a diagram.

Figure 11.



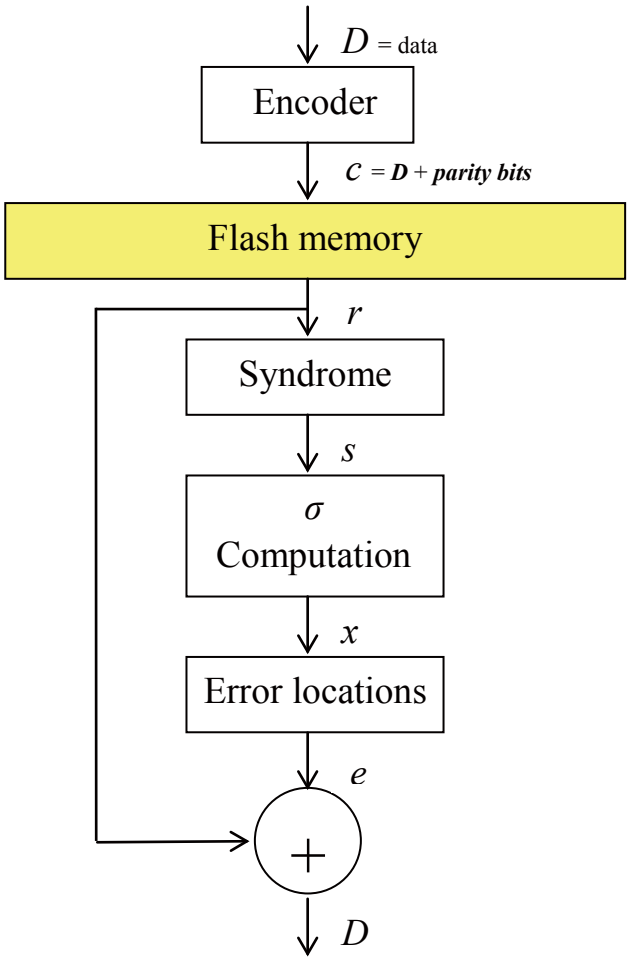
Bose-Chaudhuri-Hocquenghem (BCH) Codes

As error correction requirements for MLC flash memory increase above 4 to 8 bits ECC per 512 bytes of data, the storage requirements for Reed-Solomon parity bytes begin to outgrow the number of bytes available in the spare area of the NAND flash devices. In order to minimize the number of parity bits required, BCH codes are used for the correction. In general, BCH is a more efficient ECC algorithm for NAND flash because NAND flash errors are not correlated (they do not occur in groups or clusters). Instead, we assume that they are randomly distributed across the NAND page.

BCH is more efficient since it targets only single bit errors and can correct more of these bit errors than a Reed-Solomon code when given the same number of parity bits. As NAND error rates continue to grow due to the continuous shrinking/scaling of the technology and higher BCH ECC levels are required, the correction block size can be increased to 1K bytes in order to further improve correction efficiency.
We will use polynomials to describe BCH encoding, syndrome calculation, and error location identification.

General Structure of BCH Code

Let's use a flow chart to describe the BCH code encode and decode procedures.



From above flow chart, we have original data ***D*** to be stored in flash memory. After the encoding procedure, we obtain ***c*** = ***D*** + ***parity bits*** as a codeword stored in flash memory physically. When we read data, we

actually need to read the whole codeword ***r***. Syndrome ***s*** calculated from ***r*** will identify any errors within the codeword if the syndrome is not zero. As usual, there is no error if the syndrome is zero. The decode process includes syndrome generation, ***σ*** computation (error locator polynomial) and ***x*** (error locations within the codeword).

BCH Encoding

For all cyclic codes, including BCH, the encoding operations are performed in the same way. For a better understanding of the operations involved, a simple example follows.

First, we define our message ***D*** *(11010)*, which contains the original data we are going to store in memory. If we need a triple-correction ECC, which means ***t*** = 3, then we need to find the m^{th} power of 2 sufficient to cover message ***D*** plus parity bits (codeword length). The total length of the codeword is ***n*** = 2^m – 1 and the number of parity bits is ***n*** – ***k*** ≤ ***mt*** where ***k*** is the length of the original data. In this case, we have ***k*** = 5. The closest power of 2 which is larger than 5 is 8 (2^3). The minimum distance is 2***t*** + 1 = 7. Thus, the distance between two codewords in a binary code is a minimum of 7 bits in which the two codewords must differ. If we choose ***m*** = 3 with ***n*** = 7 (codeword length), the minimum distance 7 cannot be satisfied. We need to select the next power of two which is 2^4 . With ***m*** = 4 and ***n*** = 15 we can satisfy the minimum distance of 7 among codewords. Therefore, we use BCH(15, 5) in this example based on original data length ***k=5*** and bits correction ***t=3*** requirement.

We have BCH(15, 5) code with ***m*** = 4, ***n*** = 15 and ***k*** = 5. The length of the codeword is 15 bits. From ***GF(2^4)***, which is constructed based on primitive polynomial $p(x) = x^4 + x + 1$, we derive the generator polynomial $g(x) = LCM(f_1(x),f_3(x),f_5(x)) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$ *where the minimal polynomials* $f_1(x) = x^4 + x + 1$, $f_3(x) = x^4 + x^3 + x^2 + x + 1$, $f_5(x) = x^2 + x + 1$ and we can encode the message ***D*** = *(11010)* as follows.

- *Pad message with (n-k) zero bits (10 bits of zeros appended to message):* $X^{n-k}\ D(x)$ = *(11010*0000000000*)*
- *To get the remainder:* $X^{n-k}\ D(x)$ *mod* $g(x)$ = *(*1100100011*)*
- *We have the codeword* $c(x)$ = *(11010*1100100011*)*

After the simple encoding process, the result is c(x) as shown in the following format:

Message	Parity Bits
11010	1100100011

Now we have codeword $c(x)$ = (110101100100011). If we read the codeword without errors, then the syndrome $c(x)$ mod $g(x)$ will be zero. However if errors occurred, the non-zero syndrome will be used as a seed to decode the codeword and find the error locations.

Syndrome

Now if the data read has 3 errors, such as $r(x)$ = *(*1**0**010**0**10010**1**011*)*, the syndrome can be found from the following equations. Due to ***t*** = 3, we have 2***t*** = *6* syndrome elements needed in order to find ***t*** errors.

$$
\begin{cases}
s_1(x) = r(x) \bmod f_1(x) \\
s_2(x) = r(x) \bmod f_2(x) = r(x) \bmod f_1(x) \\
s_3(x) = r(x) \bmod f_3(x) \\
s_4(x) = r(x) \bmod f_4(x) = r(x) \bmod f_1(x) \\
s_5(x) = r(x) \bmod f_5(x) \\
s_6(x) = r(x) \bmod f_6(x) = r(x) \bmod f_3(x)
\end{cases}
$$

In selecting the minimal polynomials, we take advantage of the property of field elements whereby several powers of the generating element have the same minimal polynomial. When *f(x)* is a polynomial over ***GF(2)***, α is an element of ***GF(2^m)***. Then we form a system of equations in α as follows:

$$
\begin{cases}
s_1(\alpha) = r(\alpha) \bmod f_1(\alpha) = \alpha^3 + \alpha^2 + \alpha + 1 \equiv \alpha^{12} \\
s_2(\alpha^2) = r(\alpha^2) \bmod f_2(\alpha^2) = r(\alpha^2) \bmod f_1(\alpha^2) = \alpha^3 + \alpha \equiv \alpha^9 \\
s_3(\alpha^3) = r(\alpha^3) \bmod f_3(\alpha^3) = \alpha^9 + \alpha^6 + \alpha^3 + 1 \equiv \alpha^{12} \\
s_4(\alpha^4) = r(\alpha^4) \bmod f_4(\alpha^4) = r(\alpha^4) \bmod f_1(\alpha^4) = \alpha^{12} + \alpha^8 + \alpha^4 + 1 \equiv \alpha^3 \\
s_5(\alpha_5) = r(\alpha^5) \bmod f_5(\alpha^5) = \alpha^5 \\
s_6(\alpha_6) = r(\alpha^6) \bmod f_6(\alpha^6) = r(\alpha^6) \bmod f_3(\alpha^6) = \alpha^{18} + \alpha^{12} + \alpha^6 + 1 \equiv \alpha^9
\end{cases}
$$

The field we are working in is ***GF(16)***, with the results shown in Table 3.

Table 3. Field of 16 Elements generated with Primitive Polynomial = $x^4 + x + 1$

Power Form	n-Tuple Form	Polynomial Form
0	0000	0
1	0001	1
α	0010	α
α^2	0100	α^2
α^3	1000	α^3
α^4	0011	α + 1
α^5	0110	α^2 + α
α^6	1100	α^3 + α^2
α^7	1011	α^3 + α + 1
α^8	0101	α^2 + 1
α^9	1010	α^3 + α
α^{10}	0111	α^2 + α + 1
α^{11}	1110	α^3 + α^2 + α
α^{12}	1111	α^3 + α^2 + α + 1
α^{13}	1101	α^3 + α^2 + 1
α^{14}	1001	α^3 + 1

Now we have syndrome elements (s_1 s_2 s_3 s_4 s_5 s_6) from the above a system of equations in α.

BCH Decoding

From the example above, the method of decoding the codeword can be reduced to a simple algorithm by Berlekamp that can find ***σ(x)*** (the error locator polynomial) iteratively. First of all, let's make a table for the current example of BCH(15,5) below.

Table 4. Empty BCH (15,5) Decoding Table

μ	$\sigma^{(\mu)}(x)$	d_μ	l_μ	$2\mu - l_\mu$
- 1⁄2	1	1	0	-1
0	1	$s_1 = \alpha^{12}$	0	0
1				
2				
t = 3		-	-	-

Using the algorithm, we need to fill the cyan colored area in the above table and find σ(x). Let's do it one by one as follows.

1) Set $\mu = 0$. We see $d_\mu \neq 0$ (but s_1), so we take $\rho = -½$, and the equation is
$\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x) + d_\mu d_\rho^{-1} x^{2(\mu-\rho)}\ \sigma^{(\rho)}(x) = \sigma^{(0)}(x) + d_0 d_{-½}^{-1} x^{2(0+½)}\ \sigma^{(-½)}(x) = 1 + \alpha^{12}\ (1)\ (x)\ (1) = \alpha^{12}x + 1$

So we get $\sigma^{(1)}(x) = \alpha^{12}x + 1$.

Then, $l_{\mu+1} = L = deg(\sigma^{(\mu+1)}(x)) = \deg(\alpha^{12}x + 1) = 1$.
Now we calculate:
$d_{\mu+1} = s_{2\mu+3} + \sigma_1^{(\mu+1)}\ s_{2\mu+2} + \sigma_2^{(\mu+1)}\ s_{2\mu+1} + \ldots + \sigma_L^{(\mu+1)}\ s_{2\mu+3-L} = s_3 + \sigma_1^{(1)}\ s_2 = \alpha^{12} + \alpha^{12}(\alpha^9) = \alpha^4$

So we get $d_1 = \alpha^4$

2) Set Set $\mu = 1$. We see $d_\mu \neq 0$, so we take $\rho = 0$, and the equation is
$\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x) + d_\mu d_\rho^{-1} x^{2(\mu-\rho)}\ \sigma^{(\rho)}(x) = \sigma^{(1)}(x) + d_1 d_0^{-1} x^{2(1-0)}\ \sigma^{(0)}(x) =$
$(\alpha^{12}x + 1) + \alpha^4\ (\alpha^{12})^{-1} x^2\ (1) = \alpha^{12}x + 1 + \alpha^7 x^2 = \alpha^7 x^2 + \alpha^{12}x + 1$

So we get $\sigma^{(2)}(x) = \alpha^7 x^2 + \alpha^{12}x + 1$.

Then, $l_{\mu+1} = L = \deg(\sigma^{(\mu+1)}(x)) = \deg(\alpha^7 x^2 + \alpha^{12}x + 1) = 2$.
Now we calculate:
$d_{\mu+1} = s_{2\mu+3} + \sigma 1^{(\mu+1)}\ s_{2\mu+2} + \sigma 2^{(\mu+1)}\ s_{2\mu+1} + \ldots + \sigma L^{(\mu+1)}\ s_{2\mu+3-L} =$
$s_5 + \sigma 1^{(2)}\ s_4 + \sigma 2^{(2)}\ s_3 = \alpha^5 + \alpha^{12}(\alpha^3) + \alpha^7(\alpha^{12}) = \alpha^2$

So we get $d_2 = \alpha^2$

3) Set Set $\mu = 2$. We see $d_\mu \neq 0$, so we take $\rho = 1$, and the equation is
$\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x) + d_\mu d_\rho^{-1} x^{2(\mu-\rho)}\ \sigma^{(\rho)}(x) = \sigma^{(2)}(x) + d_2 d_1^{-1} x^{2(2-1)}\ \sigma^{(1)}(x) =$
$(\alpha^7 x^2 + \alpha^{12}x + 1) + \alpha^2\ (\alpha^4)^{-1} x^2\ (\alpha^{12}x + 1) = \alpha^7 x^2 + \alpha^{12}x + 1 + \alpha^{13}x^2(\alpha^{12}x + 1) = \alpha^{25}x^3 + (\alpha^{13} + \alpha^7)x^2 + \alpha^{12}x + 1$
$= \alpha^{25}x^3 + \alpha^5 x^2 + \alpha^{12}x + 1$

So we get $\sigma^{(3)}(x) = \alpha^{25}x^3 + \alpha^5 x^2 + \alpha^{12}x + 1$.

The final error locator polynomial is $\sigma^{(\mu)}(x) = \alpha^{25}x^3 + \alpha^5x^2 + \alpha^{12}x + 1$.

Let's fill in the Table 4 as shown below in Table 5.

Table 5. Completed BCH (15,5) Decoding Table

μ	$\sigma^{(\mu)}(x)$	d_μ	l_μ	$2\mu - l_\mu$
- 1/2	1	1	0	-1
0	1	$s_1 = \alpha^{12}$	0	0
1	$\alpha^{12}x + 1$	α^4	1	1
2	$\alpha^7x^2 + \alpha^{12}x + 1$	α^2	2	2
t = 3	$\alpha^{25}x^3 + \alpha^5x^2 + \alpha^{12}x + 1$	-	-	-

The next step is to find the roots of $\sigma^{(\mu)}(x)$ in ***GF(16)*** by trial and error substitution. The Chien algorithm is an efficient search algorithm for finding the roots. In this case the roots are α^2, α^6 and α^{12}. The bits positions of the error locations correspond **to the inverse of their roots** such as α^2, α^6 and α^{12}. Then the error pattern polynomial is $e(x) = x^{13} + x^9 + x^3$, which is (010001000001000). The recovered data ***D*** is the data read ***r(x)*** exclusive-ORed with error pattern ***e(x)***.

That is, ***D(x)*** = ***r(x)*** $\oplus$ ***e(x)*** = (100100100101011) $\oplus$ (010001000001000) = (110101100100011)

We have recovered the correct data after the ECC decoding process.

ECC Algorithm for 4bit Error Correction

Macronix MX30LF4G18AB and MX30LF2G18AB are 4Gb and 2Gb SLC NAND products that require 4-bit ECC per 528 bytes of data, which is 4224 bits. BCH code requires 52 parity bits while Reed-Solomon code needs 72 parity bits in order to correct 4-bit errors per 4224 data bits. Therefore, if we use the binary BCH code algorithm, we require less of the NAND spare memory space.

We have

$$n = 4096 \text{ (data area)} + 128 \text{ (spare area)} \leq 2^m - 1 \rightarrow \boldsymbol{m} = 13$$

This code supports 8192 codeword bits with 52 parity bits, four bit error correction (***t = 4***) and minimum distance of 9. Therefore, BCH code operates in ***GF(2^{13})*** such as BCH(8191, 8139) with primitive polynomial,

$$\boldsymbol{p(x) = x^{13} + x^4 + x^3 + x} + 1.$$

The minimal polynomials for ***GF(2^{13})*** in binary BCH codes are:

$$\boldsymbol{f_1(x) = x^{13} + x^4 + x^3 + x + 1}$$

$$\boldsymbol{f_3(x) = x^{13} + x^{10} + x^9 + x^7 + x^5 + x^4 + 1}$$

$$\boldsymbol{f_5(x) = x^{13} + x^{11} + x^8 + x^7 + x^4 + x + 1}$$

$$\boldsymbol{f_7(x) = x^{13} + x^{10} + x^9 + x^8 + x^6 + x^3 + x^2 + x + 1}$$

The generator polynomial $\boldsymbol{g(x) = LCM(f_1(x), f_3(x), f_5(x), f_7(x))}$, where

$$\boldsymbol{g(x) = x^{52} + x^{50} + x^{46} + x^{44} + x^{41} + x^{37} + x^{36} + x^{30} + x^{25} + x^{24} + x^{23} + x^{21} + x^{19} + x^{17} + x^{16} + x^{15} + x^{10} + x^9 + x^7 + x^5 + x^3 + x + 1}$$

The implementation of BCH(8191,8139) can be done in either hardware or firmware, depending on the application. The major portions of the binary BCH code flow are the *encoding engine*, the *syndrome calculation* and the *decoding engine*. Each of the major portions in binary BCH code can be implemented in hardware or firmware depending on the resources available in the embedded system. The need for temporary registers in the embedded system is inevitable due to the many finite field calculation operations required within those engines. When the syndromes all work out to zero, we can choose to turn off the decoding engine to reduce system power consumption since there are no error bits to recover. The Berlekamp algorithm in the decoding portion is complex and the throughput of a firmware implementation may not be adequate for systems requiring high data rates or in systems with limited computing power. Therefore, optimization of the algorithm for those applications is very important.

Summary

In this technical note, we introduced Hamming code, Reed-Solomon code and BCH code. Hamming code is used for single error correction in many applications. Reed-Solomon code is a symbol based method that is good for burst errors in embedded systems. Binary BCH code is more efficient, using fewer parity bits than Reed-Solomon code based on single bit errors. The implementation of the ECC function can be done in hardware or firmware, depending on system application with considerations made for system performance, power, and cost. With the Macronix SLC NAND flash memory products, ECC code will further enhance the quality and reliability of your embedded system.

Revision History

Revision No.	Description	Page	Date
REV. 1	Initial Release	ALL	17th, Feb., 2014

Except for customized products which have been expressly identified in the applicable agreement, Macronix's products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only, and not for use in any applications which may, directly or indirectly, cause death, personal injury, or severe property damages. In the event Macronix products are used in contradicted to their target usage above, the buyer shall take any and all actions to ensure said Macronix's product qualified for its actual use in accordance with the applicable laws and regulations; and Macronix as well as it's suppliers and/or distributors shall be released from any and all liability arisen therefrom.

For the contact and order information, please visit Macronix's Web site at: http://www.macronix.com