The ECC values are generated based on the most widely used Hamming algorithm for NAND Flash based applications. Understanding the Hamming algorithm and how the generated ECC bits are stored is important for understanding the ECC handling in EZ-USB. The number of ECC1:2 bits used in the two ECC configurations for EZ-USB are:

- 1. One ECC calculated over 512 bytes uses (9+3)x2 = 24 bits i.e. 3 bytes
- 2.Two ECCs, each calculated over 256 bytes (SmartMedia Standard), each uses (8+3)x2 = 22 bits Note: The number of bits needed for ECC using Hamming algorithm for 2^N bits is equal to 2xN

The ECC bits are parity bits calculated over different sets of systematically partitioned data bits. These partitions are halves, fourths, eighths, and so on till the granularity reaches bit level. The size of any set is always half the size of the data. This partitioning scheme generates two sets of ECC bits: even bits (ECCe) and odd bits (ECCo). To understand how partitioning works lets take an example of a set with 8 elements: $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$. For Hamming code calculation these 8 elements are 8 bits of a byte.

Granularity		Partitions								
halves		{0, 1,	2, 3}		{4, 5, 6, 7}					
index		C)			1				
	even halves	: {0, 1, 2, 3}	ECCel	2: 0^1^2^3	odd halves:	{4, 5, 6, 7}	ECCob2: 4^5^6^7			
fourths	{0,	{0, 1}		{2, 3}		{4, 5}		{6, 7}		
index	()	1			2	3	}		
	even fourth	s: {0, 1}, {4, 5	} ECCek	1: 0^1^4^5	odd fourths	s: {2, 3}, {6, 7}	ECCob	1: 2^3^6^7		
eighths	{0}	{1}	{2}	{3}	{4}	{5}	{6}	{7}		
index	0	1	2 3		4	5	6	7		
	even eighth	s: {0},{2},{4},{	[6] ECCek	0: 0^2^4^6	odd eighths: {1},{3},{5},{7}			0: 1^3^5^7		

Figure 1: Partitioning scheme for the hamming code

In figure 1, the elements are divided in halves, fourths, eighths... till granularity (left column) has reached to singleton sets. The union of the even sets and odd sets are separated out for each partitioning level. The even and odd sets can be identified from their index values. Sets that have even indices are used for even parity calculation and vice versa. Ex. The fourths {2, 3} and {6, 7} are indexed 1 and 3 respectively. Hence they form the parity set for the ECCo bit which corresponds to that generated by fourths. Parity calculated is show to the right of the identified sets in figure 1. Thus, the most significant even parity bit ECCeb2 is (b2 is for bit 2) is parity of elements in even half i.e. parity calculated over bits 0, 1, 2, and 3. Figure 1 shows how the rest of the parity bits are calculated. For this example, we get ECCe and ECCo as:

ECCe = {ECCeb2, ECCeb1, ECCeb0} ECCo = {ECCob2, ECCob1, ECCob0}

The data packets in EZ-USB are 256 X 8 bits or 512 X 8 bits across which error correction is implemented. For correction to work properly it is necessary to separate the row and column parity bits. This differentiation helps pin point the address of the erroneous bit correctly. Hence, for our data/packet representation, we make further divisions in each set: column bits and row bits. All it means is that the parity bits generated out of rows are bunched together and that of columns are bunched together for each even and odd parity bit set. In such two dimensional cases, the ECC generation takes two steps. First is calculation of the bit and byte parities. Byte parity is the parity calculated over the bits of a byte. Bit parity is the parity calculated over all the same indexed bits of a packet e.g. all 0th bits or all 8th bits of all the bytes. Then partitioning these bit and byte parities. Finally, ECC parity bits are generated from these bit and byte parity partitions.

The figure 2 details the two dimensional partitions for a 4 byte packet. The data bits are shown with highlighted background. Parity bits are in italics. For calculations, parity bits for bytes B0-B3 are pB0-pB3 and similarly for bits b0-b7 are pb0-pb7. The nomenclature $\#\alpha^*$ is followed for partitions and ECC parity bits. # is the number representing how many bits are in the set (e.g. halves for an 8 bit parity set are denoted by 4, halves for 16 bit parity set are denoted by 8), α is R for rows and C for columns and * is the number representing the index of the partition. * represents even or odd value by e/o for ECC parity bits. The row partitions are calculated over the byte parities and the column parities over bit parities. Figure 2 shows only the partitioning, the equations below show how the ECC parity bits are calculated out of the partitioned parity bits.

	b7	b6	b5	b4	b3	b2	b1	b0	Byte Parity	Row	Partit	tions
В3	0	1	0	0	0	1	1	0	1	1R3	٦	2R1
B2	1	1	0	1	0	1	0	1	1	1R2		ZNI
B1	1	1	0	0	0	0	1	1	0	1R1	ר	2R0
В0	1	1	0	0	1	0	1	1	1	1R0		ZNU
Bit Parity	0	0	0	1	1	0	1	1				
	1C7	1C6	1C5	1C4	1C3	1C2	1C1	1C0				
0.1	L		L		L		L					
Column Partitions	20	C3	20	C2	20	C1	20	00				
T di titionis	L		Γ		L		ı					
		40	C1			40	0					

Figure 2: Bit partitions for 4 byte data packet

Hence,				
2Re	e = 2R0	= pB0 ^ pB1	= 1 ^ 0	= 1
2Ro) = 2R1	= pB2 ^ pB3	= 1 ^ 1	= 0
1Re	e = 1R0 ^ 1R2	= pB0 ^ pB2	= 1 ^ 1	= 0
1Ro) = 1R1 ^ 1R3	= pB1 ^ pB3	= 0 ^ 1	= 1
4Ce	e = 4C0	= pb0 ^ pb1 ^ pb2 ^ pb3	= 1 ^ 1 ^ 0 ^ 1	= 1
4Co) = 4C1	= pb4 ^ pb5 ^ pb6 ^ pb7	= 1 ^ 0 ^ 0 ^ 0	= 1
2Ce	e = 2C0 ^ 2C2	= (pb0 ^ pb1) ^ (pb4 ^ pb5)	= (1 ^ 1) ^ (1 ^ 0)	= 1
2Co) = 2C1 ^ 2C3	= (pb2 ^ pb3) ^ (pb6 ^ pb7)	= (0 ^ 1) ^ (0 ^ 0)	= 1
1Ce	e = 1C0 ^ 1C2 ^ 1C4 ^ 1C6	= pb0 ^ pb2 ^ pb4 ^ pb6	= 1 ^ 0 ^ 1 ^ 0	= 0
1Co	o = 1C1 ^ 1C3 ^ 1C5 ^ 1C7	= pb1 ^ pb3 ^ pb5 ^ pb7	= 1 ^ 1 ^ 0 ^ 0	= 0
ECC	Ce[4:0] = ECCe(old)	= { 4Ce, 2Ce, 1Ce, 2Re, 1Re }		= 11010
ECC	Co[4:0] = ECCo(old)	= { 4Co, 2Co, 1Co, 2Ro, 1Ro }		= 11001

When there is an error in a particular bit, new ECC bits are generated. Figure 3 shows error bit and parity changes in dotted boxes.

	b7	b6	b5	b4	b3	b2	b1	b0	Byte Parity
В3	0	1	0	0	0	1	1	0	1
B2	1	1	0	1	0	1	0	1	1
B1	1	1	0	0	0	0	1	1	0
В0	1	1	0	0	0	0	1	1	0
Bit Parity	0	0	0	1	0	0	1	1	

Figure 3: Bit parities for 4 byte data packet with one bit error

Hence, (new equations)			
2Re = 2R0	= pB0 ^ pB1	= 0 ^ 0	= 0
2Ro = 2R1	= pB2 ^ pB3	= 1 ^ 1	= 0
1Re = 1R0 ^ 1R2	= pB0 ^ pB2	= 0 ^ 1	= 1
1Ro = 1R1 ^ 1R3	= pB1 ^ pB3	= 0 ^ 1	= 1
4Ce = 4C0	= pb0 ^ pb1 ^ pb2 ^ pb3	= 1 ^ 1 ^ 0 ^ 0	= 0
4Co = 4C1	= pb4 ^ pb5 ^ pb6 ^ pb7	= 1 ^ 0 ^ 0 ^ 0	= 1
2Ce = 2C0 ^ 2C2	= (pb0 ^ pb1) ^ (pb4 ^ pb5)	= (1 ^ 1) ^ (1 ^ 0)	= 1
2Co = 2C1 ^ 2C3	= (pb2 ^ pb3) ^ (pb6 ^ pb7)	= (0 ^ 0) ^ (0 ^ 0)	= 0
1Ce = 1C0 ^ 1C2 ^ 1C4 ^ 1C6	= pb0 ^ pb2 ^ pb4 ^ pb6	= 1 ^ 0 ^ 1 ^ 0	= 0
1Co = 1C1 ^ 1C3 ^ 1C5 ^ 1C7	= pb1 ^ pb3 ^ pb5 ^ pb7	= 1 ^ 0 ^ 0 ^ 0	= 1
ECCe[4:0] = ECCe(new)	= { 4Ce, 2Ce, 1Ce, 2Re, 1Re }		= 01001
ECCo[4:0] = ECCo(new)	= { 4Co, 2Co, 1Co, 2Ro, 1Ro }		= 10101

First step in the correction process is to check if there is no error, a single bit correctible error or incorrectible errors.

This ECC check indicates the nature of errors in the data packet. Any result which has a '1' is erroneous and all '1's indicate a correctible single bit error. Here, there is a correctible single bit error detected. The location of the error bit can be found from the old and new ECCo values.

Since we have followed the column[4:2] and row[1:0] convention on the ECC parities, the error bit column address is 011 (= C) and row address is 00 (= R). Flipping the particular bit gives correct data.

$$B[R] = B[R] \wedge (1 << C)$$
; //flipping the error bit

ECC bits in EZ-USB are stored as follows:

ECC1[0]/ECC2[0]

b7	b6	b5	b4	b3	b2	b1	b0
128Ro	128Re	64Ro	64Re	32Ro	32Re	16Ro	16Re

ECC1[1]/ECC2[1]

b7	b6	b5	b4	b3	b2	b1	b0
8Ro	8Re	4Ro	4Re	2Ro	2Re	1Ro	1Re

ECC1[2]/ECC2[2]

b7	b6	b5	b4	b3	b2	b1	b0
4Co	4Ce	2Co	2Ce	1Co	1Ce	256Co	256Ce

Note: b1:0 are used only for ECC1[2] when ECCM = 1, for ECC2[2] they are always unused. The odd and even values are stored side by side for ease of use in error check and correction.

Explanation for the code:

The old ECC value is the ECC value stored in the spare area or that which is received and the new ECC value is the calculated value over the data packet that is read. These values are XORed and stored in three bytes as follows:

```
a = ECC(old)[0] ^ ECC(new)[0];
b = ECC(old)[1] ^ ECC(new)[1];
c = ECC(old)[2] ^ ECC(new)[2];
```

For ECC check, all that needs to be done is XOR the respective even and odd values. As the even and odd values are stored adjacent to each other, the bytes can be XORed with themselves after single bit shift. The even bits of the result holds the ECC check values. The odd bits are 'don't care'.

For ex. Echeck_b will be:

b >> 1	>>	8Ro	8Re	4Ro	4Re	2Ro	2Re	1Ro	1Re
	1								
b	8Ro	8Re	4Ro	4Re	2Ro	2Re	1Ro	1Re	
	\downarrow								
Echeck_b	Х	8Ro^8Re	Х	4Ro^4Re	Х	2Ro^2Re	Х	1Ro^1Re	
	b7	b6	b5	b4	b3	b2	b1	b0	

```
Echeck_a = a ^ (a>>1);

Echeck_b = b ^ (b>>1);

Echeck_c = c ^ (c>>1);

//condition check for one bit correctible error for 512 bytes

if ((Echeck_a & 0x55) == 0x55)&&(Echeck_b & 0x55) == 0x55)&&(Echeck_c & 0x55) == 0x55))

//condition check for one bit correctible error for 256 bytes, b0 of Echeck_c is not used

if ((Echeck_a & 0x55) == 0x55)&&(Echeck_b & 0x55) == 0x55)&&(Echeck_c & 0x54) == 0x54))
```

The odd bits from a, b and c can be stored respectively in index variables for row and column. These variables can be used to directly correct erroneous bit in the data packet.