Nand ECC 校验和纠错 详解(转)

ECC 的全称是 Error Checking and Correction，是一种用于 Nand 的差错检测和修正算法。如果操作时序和电路稳定性不存在问题的话，NAND Flash 出错的时候一般不会造成整个 Block 或是 Page 不能读取或是全部出错，而是整个 Page（例如 512Bytes）中只有一个或几个 bit 出错。ECC 能纠正 1 个比特错误和检测 2 个比特错误，而且计算速度很快，但对 1 比特以上的错误无法纠正，对 2 比特以上的错误不保证能检测。

校验码生成算法：ECC 校验每次对 256 字节的数据进行操作，包含列校验和行校验。对每个待校验的 Bit 位求异或，若结果为 0，则表明含有偶数个 1；若结果为 1，则表明含有奇数个 1。列校验规则如表 1 所示。256 字节数据形成 256 行、8 列的矩阵，矩阵每个元素表示一个 Bit 位。

表格 1　ECC 列校验规则示意图

Byte 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 2	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 3	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 252	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 253	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 254	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 255	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	CP1	CP0	CP1	CP0	CP1	CP0	CP1	CP0
	CP3		CP2		CP3		CP2	
	CP5				CP4			

其中 CP0 ~ CP5 为六个 Bit 位，表示 Column Parity（列极性），

CP0 为第 0、2、4、6 列的极性，CP1 为第 1、3、5、7 列的极性，

CP2 为第 0、1、4、5 列的极性，CP3 为第 2、3、6、7 列的极性，

CP4 为第 0、1、2、3 列的极性，CP5 为第 4、5、6、7 列的极性。

用公式表示就是：CP0=Bit0^Bit2^Bit4^Bit6， 表示第 0 列内部 256 个 Bit 位异或之后再跟第 2 列 256 个 Bit 位异或，再跟第 4 列、第 6 列的每个 Bit 位异或，这样，CP0 其实是 256*4=1024 个 Bit 位异或的结果。CP1 ~ CP5 依此类推。

行校验如下图所示

表格 2　ECC 行校验规则示意图

Byte 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP2	RP4	….	RP14
Byte 1	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				
Byte 2	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP3			
Byte 3	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				
Byte 252	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP2	RP5		RP15
Byte 253	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				
Byte 254	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP3			
Byte 255	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				

其中 RP0 ~ RP15 为十六个 Bit 位，表示 Row Parity（行极性），
RP0 为第 0、2、4、6、….252、254 个字节的极性
RP1-----1、3、5、7……253、255
RP2----0、1、4、5、8、9…..252、253 （处理 2 个 Byte，跳过 2 个 Byte）
RP3---- 2、3、6、7、10、11…..254、255 （跳过 2 个 Byte，处理 2 个 Byte）
RP4---- 处理 4 个 Byte，跳过 4 个 Byte；
RP5---- 跳过 4 个 Byte，处理 4 个 Byte；
RP6---- 处理 8 个 Byte，跳过 8 个 Byte
RP7---- 跳过 8 个 Byte，处理 8 个 Byte；
RP8---- 处理 16 个 Byte，跳过 16 个 Byte
RP9---- 跳过 16 个 Byte，处理 16 个 Byte；
RP10----处理 32 个 Byte，跳过 32 个 Byte
RP11----跳过 32 个 Byte，处理 32 个 Byte；
RP12----处理 64 个 Byte，跳过 64 个 Byte
RP13----跳过 64 个 Byte，处理 64 个 Byte；
RP14----处理 128 个 Byte，跳过 128 个 Byte
RP15----跳过 128 个 Byte，处理 128 个 Byte；
可见，RP0 ~ RP15 每个 Bit 位都是 128 个字节（也就是 128 行）即 128*8=1024 个 Bit 位求异或的结果。
综上所述，对 256 字节的数据共生成了 6 个 Bit 的列校验结果，16 个 Bit 的行校验结果，共 22 个 Bit。在 Nand 中使用 3 个字节存放校验结果，多余的两个 Bit 位置 1。存放次序如下表所示：

表格 3　K9F1208 中 22Bit 校验码的排列规则

ECC	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	RP7	RP6	RP5	RP4	RP3	RP2	RP1	RP0
Byte 1	RP15	RP14	RP13	RP12	RP11	RP10	RP9	RP8
Byte 2	CP5	CP4	CP3	CP2	CP1	CP0	1	1

以 K9F1208 为例，每个 Page 页包含 512 字节的数据区和 16 字节的 OOB 区。前 256 字节数据生成 3 字节 ECC 校验码，后 256 字节数据生成 3 字节 ECC 校验码，共 6 字节 ECC 校验码存放在 OOB 区中，存放的位置为 OOB 区的第 0、1、2 和 3、6、7 字节。

文件:MakeEccTable.rar
校验码生成算法的 C 语言实现
在 Linux 内核中 ECC 校验算法所在的文件为 drivers/mtd/nand/nand_ecc.c，其实现有新、旧两种，在 2.6.27 及更早的内核中使用的程序，从 2.6.28 开始已经不再使用，而换成了效率更高的程序。可以在 Documentation/mtd/nand_ecc.txt 文件中找到对新程序的详细介绍。
首先分析一下 2.6.27 内核中的ECC实现，源代码见:
http://lxr.linux.no/linux+v2.6.27/drivers/mtd/nand/nand_ecc.c

```
static const u_char nand_ecc_precalc_table[] = {
 47         0x00, 0x55, 0x56, 0x03, 0x59, 0x0c, 0x0f, 0x5a, 0x5a, 0x0f,
 0x0c, 0x59, 0x03, 0x56, 0x55, 0x00,
 48         0x65, 0x30, 0x33, 0x66, 0x3c, 0x69, 0x6a, 0x3f, 0x3f, 0x6a,
 0x69, 0x3c, 0x66, 0x33, 0x30, 0x65,
 49         0x66, 0x33, 0x30, 0x65, 0x3f, 0x6a, 0x69, 0x3c, 0x3c, 0x69,
 0x6a, 0x3f, 0x65, 0x30, 0x33, 0x66,
 50         0x03, 0x56, 0x55, 0x00, 0x5a, 0x0f, 0x0c, 0x59, 0x59, 0x0c,
 0x0f, 0x5a, 0x00, 0x55, 0x56, 0x03,
 51         0x69, 0x3c, 0x3f, 0x6a, 0x30, 0x65, 0x66, 0x33, 0x33, 0x66,
 0x65, 0x30, 0x6a, 0x3f, 0x3c, 0x69,
 52         0x0c, 0x59, 0x5a, 0x0f, 0x55, 0x00, 0x03, 0x56, 0x56, 0x03,
 0x00, 0x55, 0x0f, 0x5a, 0x59, 0x0c,
 53         0x0f, 0x5a, 0x59, 0x0c, 0x56, 0x03, 0x00, 0x55, 0x55, 0x00,
 0x03, 0x56, 0x0c, 0x59, 0x5a, 0x0f,
 54         0x6a, 0x3f, 0x3c, 0x69, 0x33, 0x66, 0x65, 0x30, 0x30, 0x65,
 0x66, 0x33, 0x69, 0x3c, 0x3f, 0x6a,
 55         0x6a, 0x3f, 0x3c, 0x69, 0x33, 0x66, 0x65, 0x30, 0x30, 0x65,
 0x66, 0x33, 0x69, 0x3c, 0x3f, 0x6a,
 56         0x0f, 0x5a, 0x59, 0x0c, 0x56, 0x03, 0x00, 0x55, 0x55, 0x00,
 0x03, 0x56, 0x0c, 0x59, 0x5a, 0x0f,
 57         0x0c, 0x59, 0x5a, 0x0f, 0x55, 0x00, 0x03, 0x56, 0x56, 0x03,
 0x00, 0x55, 0x0f, 0x5a, 0x59, 0x0c,
```

```
58          0x69, 0x3c, 0x3f, 0x6a, 0x30, 0x65, 0x66, 0x33, 0x33, 0x66,
0x65, 0x30, 0x6a, 0x3f, 0x3c, 0x69,
59          0x03, 0x56, 0x55, 0x00, 0x5a, 0x0f, 0x0c, 0x59, 0x59, 0x0c,
0x0f, 0x5a, 0x00, 0x55, 0x56, 0x03,
60          0x66, 0x33, 0x30, 0x65, 0x3f, 0x6a, 0x69, 0x3c, 0x3c, 0x69,
0x6a, 0x3f, 0x65, 0x30, 0x33, 0x66,
61          0x65, 0x30, 0x33, 0x66, 0x3c, 0x69, 0x6a, 0x3f, 0x3f, 0x6a,
0x69, 0x3c, 0x66, 0x33, 0x30, 0x65,
62          0x00, 0x55, 0x56, 0x03, 0x59, 0x0c, 0x0f, 0x5a, 0x5a, 0x0f,
0x0c, 0x59, 0x03, 0x56, 0x55, 0x00
            };
```

为了加快计算速度，程序中使用了一个预先计算好的列极性表。这个表中每一个元素都是 unsigned char 类型，表示 8 位二进制数。

表中 8 位二进制数每位的含义：

表格 4　预计算表中每字节的各 Bit 位含义

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	0	行极性	CP5	CP4	CP3	CP2	CP1	CP0

这个表的意思是：对 0~255 这 256 个数，计算并存储每个数的列校验值和行校验值，以数作数组下标。比如 nand_ecc_precalc_table[13]　存储 13 的列校验值和行校验值，13 的二进制表示为 00001101，　其 CP0 = Bit0^Bit2^Bit4^Bit6 = 0;

CP1 = Bit1^Bit3^Bit5^Bit7 = 1;

CP2 = Bit0^Bit1^Bit4^Bit5 = 1;

CP3 = Bit2^Bit3^Bit6^Bit7 = 0;

CP4 = Bit0^Bit1^Bit2^Bit3 = 1;

CP5 = Bit4^Bit5^Bit6^Bit7 = 0;

其行极性 RP = Bit0^Bit1^Bit2^Bit3^Bit4^Bit5^Bit6^Bit7 = 1;

则 nand_ecc_precalc_table[13] 处存储的值应该是 0101 0110，即 0x56.

注意，数组 nand_ecc_precalc_table 的下标其实是我们要校验的一个字节数据。

理解了这个表的含义，也就很容易写个程序生成这个表了。程序见附件中的 MakeEccTable.c 文件。

有了这个表，对单字节数据 dat，可以直接查表 nand_ecc_precalc_table[dat] 得到 dat 的行校验值和列校验值。 但是 ECC 实际要校验的是 256 字节的数据，需要进行 256 次查表，对得到的 256 个查表结果进行按位异或，最终结果的 Bit0 ~ Bit5 即是 256 字节数据的 CP0 ~ CP5.

```
71 int nand_calculate_ecc(struct mtd_info *mtd, const u_char *dat,
72                         u_char *ecc_code)
73 {
74         uint8_t idx, reg1, reg2, reg3, tmp1, tmp2;
75         int i;
78         reg1 = reg2 = reg3 = 0;
81         for(i = 0; i < 256; i++) {
83                 idx = nand_ecc_precalc_table[*dat++];
84                 reg1 ^= (idx & 0x3f);
```

Reg1

表格 5　变量 reg1 中各 Bit 位的含义

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	CP5	CP4	CP3	CP2	CP1	CP0

在这里，计算列极性的过程其实是先在一个字节数据的内部计算 CP0 ~ CP5，每个字节都计算完后再与其它字节的计算结果求异或。而表 1 中是先对一列 Bit0 求异或，再去异或一列 Bit2。 这两种只是计算顺序不同，结果是一致的。 因为异或运算的顺序是可交换的。

行极性的计算要复杂一些。

nand_ecc_precalc_table[] 表中的 Bit6 已经保存了每个单字节数的行极性值。对于待校验的 256 字节数据，分别查表，如果其行极性为 1，则记录该数据所在的行索引（也就是 for 循环的 i 值），这里的行索引是很重要的，因为 RP0 ~ RP15 的计算都是跟行索引紧密相关的，如 RP0 只计算偶数行，RP1 只计算奇数行，等等。

```
86                 /* All bit XOR = 1 ? */
87                 if (idx & 0x40) {
88                         reg3 ^= (uint8_t) i;
89                         reg2 ^= ~((uint8_t) i);
90                 }
91         }
```

这里的关键是理解第 88 和 89 行。Reg3 和 reg2 都是 unsigned char 型的变量，并都初始化为零。
行索引（也就是 for 循环里的 i）的取值范围为 0~255，根据表 2 可以得出以下规律：

RP0 只计算行索引的 Bit0 为 0 的行，RP1 只计算行索引的 Bit0 为 1 的行；
RP2 只计算行索引的 Bit1 为 0 的行，RP3 只计算行索引的 Bit1 为 1 的行；
RP4 只计算行索引的 Bit2 为 0 的行，RP5 只计算行索引的 Bit2 为 1 的行；
RP6 只计算行索引的 Bit3 为 0 的行，RP7 只计算行索引的 Bit3 为 1 的行；
RP8 只计算行索引的 Bit4 为 0 的行，RP9 只计算行索引的 Bit4 为 1 的行；
RP10 只计算行索引的 Bit5 为 0 的行，RP11 只计算行索引的 Bit5 为 1 的行；
RP12 只计算行索引的 Bit6 为 0 的行，RP13 只计算行索引的 Bit6 为 1 的行；
RP14 只计算行索引的 Bit7 为 0 的行，RP15 只计算行索引的 Bit7 为 1 的行；
已经知道，异或运算的作用是判断比特位为 1 的个数，跟比特位为 0 的个数没有关系。如果有偶数个 1 则异或的结果为 0，如果有奇数个 1 则异或的结果为 1。
那么，程序第 88 行，对所有行校验为 1 的行索引按位异或运算，作用便是：
判断在所有行校验为 1 的行中，
属于 RP1 计算范围内的行有多少个------由 reg3 的 Bit 0 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP3 计算范围内的行有多少个------由 reg3 的 Bit 1 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP5 计算范围内的行有多少个------由 reg3 的 Bit 2 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP7 计算范围内的行有多少个------由 reg3 的 Bit 3 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP9 计算范围内的行有多少个------由 reg3 的 Bit 4 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP11 计算范围内的行有多少个------由 reg3 的 Bit 5 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP13 计算范围内的行有多少个------由 reg3 的 Bit 6 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP15 计算范围内的行有多少个------由 reg3 的 Bit 7 指示，0 表示有偶数个，1 表示有奇数个；

所以，reg3 每个 Bit 位的作用如下表所示：
Reg3

表格 6　变量 reg3 中割 Bit 位的含义

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RP15	RP13	RP11	RP9	RP7	RP5	RP3	RP1

第 89 行，对所有行校验为 1 的行索引按位取反之后，再按位异或，作用就是判断比特位为 0 的个数。比如 reg2 的 Bit0 为 0 表示：所有行校验为 1 的行中，行索引的 Bit0 为 0 的行有偶数个，也就是落在 RP0 计算范围内的行有偶数个。所以得到结论：

在所有行校验为 1 的行中，
属于 RP0 计算范围内的行有多少个------由 reg2 的 Bit 0 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP2 计算范围内的行有多少个------由 reg2 的 Bit 1 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP4 计算范围内的行有多少个------由 reg2 的 Bit 2 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP6 计算范围内的行有多少个------由 reg2 的 Bit 3 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP8 计算范围内的行有多少个------由 reg2 的 Bit 4 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP10 计算范围内的行有多少个------由 reg2 的 Bit 5 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP12 计算范围内的行有多少个------由 reg2 的 Bit 6 指示，0 表示有偶数个，1 表示有奇数个；
属于 RP14 计算范围内的行有多少个------由 reg2 的 Bit 7 指示，0 表示有偶数个，1 表示有奇数个；

所以，reg2 每个 Bit 位的作用如下表所示：
Reg2

表格 7 变量 reg2 中各 Bit 位的含义

7	6	5	4	3	2	1	0
RP14	RP12	RP10	RP8	RP6	RP4	RP2	RP0

至此，只用了一个查找表和一个 for 循环，就把所有的校验位 CP0 ~ CP5 和 RP0 ~ RP15 全都计算出来了。下面的任务只是按照表 3 的格式，把这些比特位重新排列一下顺序而已。
从 reg2 和 reg3 中抽取出 RP8~RP15 放在 tmp1 中，抽取出 RP0~RP7 放在 tmp2 中，
Reg1 左移两位，低两位置 1，
然后把 tmp2, tmp1, reg1 放在 ECC 码的三个字节中。
程序中还有
CONFIG_MTD_NAND_ECC_SMC
， 又进行了一次取反操作，暂时还不知为何。

ECC 纠错算法

当往 NAND Flash 的 page 中写入数据的时候，每 256 字节我们生成一个 ECC 校验和，称之为原 ECC 校验和，保存到 PAGE 的 OOB（out-of-band）数据区中。当从 NAND Flash 中读取数据的时候，每 256 字节我们生成一个 ECC 校验和，称之为新 ECC 校验和。
将从 OOB 区中读出的原 ECC 校验和新 ECC 校验和按位异或，若结果为 0，则表示不存在错（或是出现了 ECC 无法检测的错误）；若 3 个字节异或结果中存在 11 个比特位为 1，表示存在一个比特错误，且可纠正；若 3 个字节异或结果中只存在 1 个比特位为 1，表示 OOB 区出错；其他情况均表示出现了无法纠正的错误。
假设 ecc_code_raw[3] 保存原始的 ECC 校验码，ecc_code_new[3] 保存新计算出的 ECC 校验码，其格式如下表所示：

表格 8　K9F1208 中 22Bit 校验码的排列规则

ECC	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	RP7	RP6	RP5	RP4	RP3	RP2	RP1	RP0
Byte 1	RP15	RP14	RP13	RP12	RP11	RP10	RP9	RP8
Byte 2	CP5	CP4	CP3	CP2	CP1	CP0	1	1

对 ecc_code_raw[3] 和 ecc_code_new[3] 按位异或，得到的结果三个字节分别保存在 s0,s1,s2 中，如果 s0s1s2 中共有 11 个 Bit 位为 1，则表示出现了一个比特位错误，可以修正。定位出错的比特位的方法是，先确定行地址（即哪个字节出错），再确定列地址（即该字节中的哪一个 Bit 位出错）。

确定行地址的方法是，设行地址为 unsigned char byteoffs，抽取 s1 中的 Bit7,Bit5,Bit3,Bit1，作为 byteoffs 的高四位， 抽取 s0 中的 Bit7,Bit5,Bit3,Bit1 作为 byteoffs 的低四位， 则 byteoffs 的值就表示出错字节的行地址（范围为 0 ~ 255）。
确定列地址的方法是：抽取 s2 中的 Bit7,Bit5,Bit3 作为 bitnum 的低三位，bitnum 其余位置 0，则 bitnum 的表示出错 Bit 位的列地址 （范围为 0 ~ 7）。

下面以一个简单的例子探索一下这其中的奥妙。
假设待校验的数据为两个字节，0x45（二进制为 0100 0101）和 0x38（二进制为 0011 1000），其行列校验码如下表所示：

表格 9　ECC 定位出错 Bit 的原理示意

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Byte 0	0	1	0	0	0	1	0	1	RP0
Byte 1	0	0	1	1	1	0	0→1	0	RP1
	CP1	CP0	CP1	CP0	CP1	CP0	CP1	CP0	
	CP3		CP2		CP3		CP2		
	CP5				CP4				

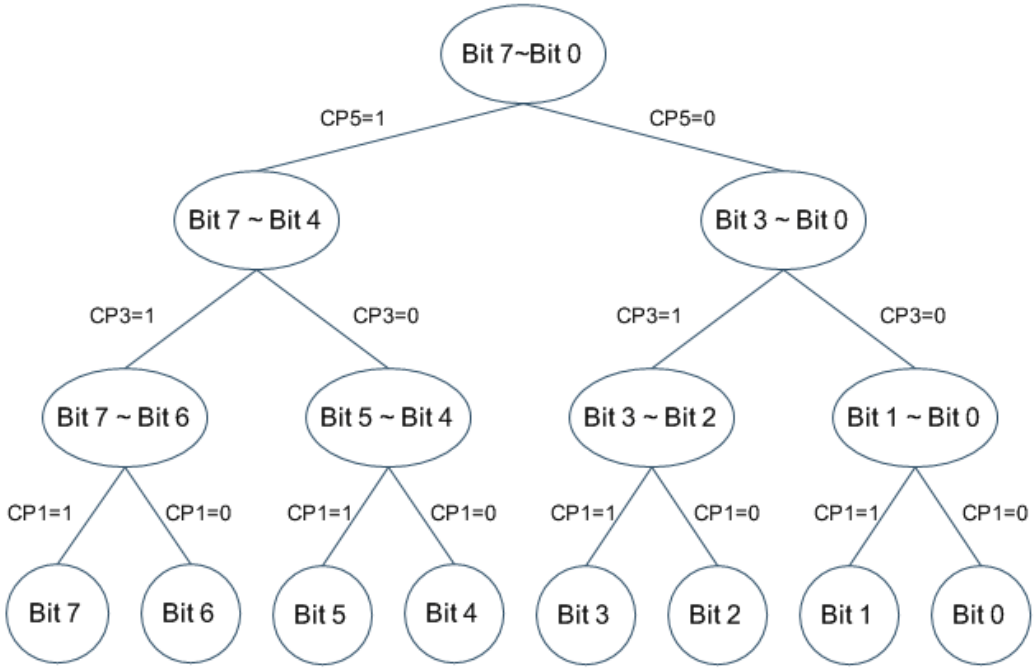
从表中可以计算出 CP5 ~ CP0 的值，列在下表的第一行（原始数据）。假设现在有一个数据位发生变化，0x38 变为 0x3A，也就是 Byte
1 的 Bit 1 由 0 变成了 1，计算得到新的 CP5 ~ CP0 值放在下表第 2 行（变化后数据）。新旧校验码求异或的结果放在下表第三行。

可见，当 Bit
1 发生变化时，列校验值中只有 CP1，CP2，CP4 发生了变化，而 CP0，CP3，CP5 没变化，也就是说 6 个 Bit 校验码有一半发生变化，则求异或的结果中有一半为 1。同理，行校验求异或的结果也有一半为 1。这就是为什么前面说 256 字节数据中的一个 Bit 位发生变化时，新旧 22Bit 校验码求异或的结果中会有 11 个 Bit 位为 1。

表格 10　新旧列校验码及其异或结果

	CP5	CP4	CP3	CP2	CP1	CP0
原始数据	1	1	1	1	0	0
变化后数据	1	0	1	0	1	0
求异或得：	0	1	0	1	1	0

再来看怎么定位出错的 Bit 位。以列地址为例，若 CP5 发生变化（异或后的 CP5=1），则出错处肯定在 Bit 4 ~ Bit 7 中；若 CP5 无变化（异或后的 CP5=0），则出错处在 Bit 0 ~ Bit 3 中，这样就筛选掉了一半的 Bit 位。剩下的 4 个 Bit 位中，再看 CP3 是否发生变化，又选出 2 个 Bit 位。剩下的 2Bit 位中再看 CP1 是否发生变化，则最终可定位 1 个出错的 Bit 位。下面的树形结构更清晰地展示了这个判决过程：



图表 1　出错 Bit 列地址定位的判决树

注意：图中的 CP 指的是求异或之后的结果中的 CP

为什么只用 CP4，CP2，CP0 呢？其实这里面包含冗余信息，因为 CP5=1 则必有 CP4=0，CP5=0 则必有 CP4=1，也就是 CP5 跟 CP4 一定相反，同理，CP3 跟 CP2 一定相反，CP1 跟 CP0 一定相反。所以只需要用一半就行了。

这样，我们从异或结果中抽取出 CP5，CP3，CP1 位，便可定位出错 Bit 位的列地址。比如上面的例子中 CP5/CP3/CP1 = 001，表示 Bit 1 出错。

同理，行校验 RP1 发生变化，抽取 RP1，可知 Byte 1 发生变化。这样定位出 Byte 1 的 Bit 0 出错。

当数据位 256 字节时，行校验使用 RP0 ~ RP15，抽取异或结果的 RP15，RP13，RP11，RP9，RP7，RP5，RP3，RP1 位便可定位出哪个 Byte 出错，再用 CP5,CP3,CP1 定位哪个 Bit 出错。

```
 93          /* Create non-inverted ECC code from line parity */
 94          tmp1  = (reg3 & 0x80) >> 0; /* B7 -> B7 */
 95          tmp1 |= (reg2 & 0x80) >> 1; /* B7 -> B6 */
 96          tmp1 |= (reg3 & 0x40) >> 1; /* B6 -> B5 */
 97          tmp1 |= (reg2 & 0x40) >> 2; /* B6 -> B4 */
 98          tmp1 |= (reg3 & 0x20) >> 2; /* B5 -> B3 */
 99          tmp1 |= (reg2 & 0x20) >> 3; /* B5 -> B2 */
100          tmp1 |= (reg3 & 0x10) >> 3; /* B4 -> B1 */
101          tmp1 |= (reg2 & 0x10) >> 4; /* B4 -> B0 */
103          tmp2  = (reg3 & 0x08) << 4; /* B3 -> B7 */
104          tmp2 |= (reg2 & 0x08) << 3; /* B3 -> B6 */
105          tmp2 |= (reg3 & 0x04) << 3; /* B2 -> B5 */
106          tmp2 |= (reg2 & 0x04) << 2; /* B2 -> B4 */
107          tmp2 |= (reg3 & 0x02) << 2; /* B1 -> B3 */
108          tmp2 |= (reg2 & 0x02) << 1; /* B1 -> B2 */
109          tmp2 |= (reg3 & 0x01) << 1; /* B0 -> B1 */
110          tmp2 |= (reg2 & 0x01) << 0; /* B7 -> B0 */
112          /* Calculate final ECC code */
113#ifdef CONFIG_MTD_NAND_ECC_SMC
114          ecc_code[0] = ~tmp2;
115          ecc_code[1] = ~tmp1;
```

```
116#else
117         ecc_code[0] = ~tmp1;
118         ecc_code[1] = ~tmp2;
119#endif
120         ecc_code[2] = ((~reg1) << 2) | 0x03;
121
122         return 0;
123}
124EXPORT_SYMBOL(nand_calculate_ecc);
125
126static inline int countbits(uint32_t byte)
127{
128         int res = 0;
129
130         for (;byte; byte >>= 1)
131                 res += byte & 0x01;
132         return res;
136 * nand_correct_data - [NAND Interface] Detect and correct bit
error(s)
137 * @mtd:        MTD block structure
138 * @dat:        raw data read from the chip
139 * @read_ecc:   ECC from the chip
140 * @calc_ecc:   the ECC calculated from raw data
```

```
142 * Detect and correct a 1 bit error for 256 byte block
144int nand_correct_data(struct mtd_info *mtd, u_char *dat,
145                       u_char *read_ecc, u_char *calc_ecc)
146{
147        uint8_t s0, s1, s2;
148
149#ifdef CONFIG_MTD_NAND_ECC_SMC
150        s0 = calc_ecc[0] ^ read_ecc[0];
151        s1 = calc_ecc[1] ^ read_ecc[1];
152        s2 = calc_ecc[2] ^ read_ecc[2];
153#else
154        s1 = calc_ecc[0] ^ read_ecc[0];
155        s0 = calc_ecc[1] ^ read_ecc[1];
156        s2 = calc_ecc[2] ^ read_ecc[2];
157#endif
158        if ((s0 | s1 | s2) == 0)
159                return 0;
161        /* Check for a single bit error */
162        if( ((s0 ^ (s0 >> 1)) & 0x55) == 0x55 &&
163            ((s1 ^ (s1 >> 1)) & 0x55) == 0x55 &&
164            ((s2 ^ (s2 >> 1)) & 0x54) == 0x54) {
165
```

```
166                 uint32_t byteoffs, bitnum;
167
168                 byteoffs = (s1 << 0) & 0x80;
169                 byteoffs |= (s1 << 1) & 0x40;
170                 byteoffs |= (s1 << 2) & 0x20;
171                 byteoffs |= (s1 << 3) & 0x10;
172
173                 byteoffs |= (s0 >> 4) & 0x08;
174                 byteoffs |= (s0 >> 3) & 0x04;
175                 byteoffs |= (s0 >> 2) & 0x02;
176                 byteoffs |= (s0 >> 1) & 0x01;
177
178                 bitnum = (s2 >> 5) & 0x04;
179                 bitnum |= (s2 >> 4) & 0x02;
180                 bitnum |= (s2 >> 3) & 0x01;
181
182                 dat[byteoffs] ^= (1 << bitnum);
183
184                 return 1;
185         }
186
```

```
187         if(countbits(s0 | ((uint32_t)s1 << 8) | ((uint32_t)s2 <<16))
== 1)
188                 return 1;
189
190         return -EBADMSG;
191}
192EXPORT_SYMBOL(nand_correct_data);
193
194MODULE_LICENSE("GPL");
195MODULE_AUTHOR("Steven J. Hill <sjhill@realitydiluted.com>");
196MODULE_DESCRIPTION("Generic NAND ECC support");
```