

Technical Note

Hamming Codes for NAND Flash Memory Devices

For the latest NAND Flash product data sheets, see www.micron.com/products/nand/partlist.aspx.

Overview

NAND Flash memory products have become the technology of choice to satisfy high-density, nonvolatile memory requirements in many applications. NAND Flash technology provides large amounts of storage at a price point lower than any of today's semiconductor alternatives. NAND Flash development has focused on low cost per bit, resulting in a technology that requires significantly more system involvement than other flash technologies. In particular, NAND Flash memory can be expected to experience minor data corruption at some point during normal operation.

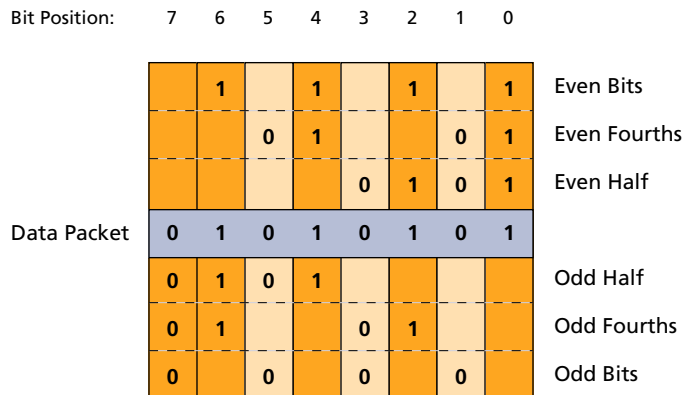
NAND Flash devices use one of two different memory cell technologies. The first cell design is the traditional implementation, where each memory cell represents a single bit of data. The single-bit-per-cell approach is categorized as single-level cell (SLC). The second approach is to program each cell in incremental amounts. With this approach, the data value is determined by how "hard" a cell has been programmed. This multi-level cell (MLC) approach allows each cell to represent 2 bits of data. Historically, SLC NAND Flash devices have provided improved data integrity when compared with their MLC counterparts. The data integrity in MLC requires a significantly more sophisticated error correction scheme than is used for SLC NAND Flash devices.

This technical note describes the use of simple Hamming codes to detect and correct data corruption that occurs during normal SLC NAND Flash device operation. The Hamming algorithm is capable of repairing single-bit data failures and detecting whether 2 bits have become corrupted. The Hamming algorithm is an industry-accepted method for error detection and correction in many SLC NAND Flash-based applications.

Hamming Code Basics

The Hamming algorithm can be described without the complex mathematics found in more sophisticated error correction methods. The most widely used Hamming algorithm for NAND Flash-based applications evaluates a packet of data and calculates two error correction code (ECC) values. Each bit in the two ECC values represents the parity of half the bits in the data packet. The trick is how the data bits are partitioned for each of the parity calculations. To calculate the ECC values, the data bits are first partitioned in halves, fourths, eighths and so on, until granularity has reached the individual bit level (see Figure 1).

Figure 1: Partitioning an 8-Bit Data Packet for Parity Calculations



After the data packet has been partitioned, the parity of each grouping is calculated to generate the two ECC values (see Figure 2). Thus, each calculation generates the parity of one of the data partitions; the response is whether that partition's parity is even or odd. The resulting responses are concatenated to make up the ECC values.

Figure 2: Even and Odd ECC Values Calculation

	Halves	Fourths	Bits		
ECC _e	= 0 ¹	^ 0 ¹	^ 1 ¹	=	000
ECC _o	= 0 ¹	^ 0 ¹	^ 0 ⁰	=	000

The two ECC values generated by the partitioning process are referred to as the even ECC (ECC_e) and odd ECC (ECC_o) values. Note that the most significant bit (MSB) of the resulting ECC values corresponds to the one-half-partition groupings; the next most significant bit corresponds to the one-fourth-partition groupings, and the least significant bit corresponds to the bit-partition groupings. The most-significant to least-significant ordering enables direct identification of a failing bit position when the data packet is analyzed at a later time.

Larger data packets require larger ECC values. Actually, each n-bit ECC value is adequate for a 2ⁿ-bit data packet (i.e., 8 bits for a 256-bit data packet). This Hamming algorithm requires a pair of ECC values, so a total of 2n bits are required to handle 2ⁿ bits of data (i.e., two 8-bit ECC values (16 bits total) for a 256-bit data packet).

Following the calculation, both the data packet and the ECC values are programmed into the NAND Flash device. At a later time, when the data packet is read out of the NAND Flash device, the ECC values are recalculated. Data corruption is indicated when the newly calculated ECC differs from the ECC values previously programmed into the NAND Flash device.

Figure 3 shows the results of the ECC calculation if a single bit in the original data becomes corrupted. The new data value is 01010101 (bit 2 flipped from 0 to 1). It is clear that corruption has occurred because the new ECC values are different from those originally calculated.

Figure 3: ECC Calculation: Corrupted Original Data Bit 2 (01010001 to 01010101)

Halves	Fourths	Bits	
$ECC_e = 0^1 1^0 0^1$	$0^1 1^0 0^1$	$1^1 1^1 1^1$	$= 000$
$ECC_o = 0^1 1^0 0^1$	$0^1 1^0 0^1$	$0^0 0^0 0^0$	$= 000$

If all four ECC values (two old and two new) are exclusive “OR-ed.” it is possible to determine whether a single bit or multiple bits have become corrupted. If the result of the calculation is all 1s (111), a single data bit has become corrupted (see Figure 4). If the result of the calculation is all 0s (000), no data corruption has occurred. If the result of this step is anything other than all 0s or all 1s, then 2 or more bits have become corrupted. Two-bit corruption will always be detected but not repaired because the Hamming code corrects only 1-bit errors. If 3 or more bits are corrupted, not only is it impossible to repair the data packet, but in this situation the Hamming algorithm may also fail to indicate that any corruption has occurred. Two, and certainly three, bits of data corruption are highly unlikely, given the profile of bit failures in SLC NAND Flash devices.

Figure 4: Calculation Determining if Corruption Has Occurred

$$ECC_e(\text{old}) \oplus ECC_o(\text{old}) \oplus ECC_e(\text{new}) \oplus ECC_o(\text{new}) = 101 \oplus 010 \oplus 000 \oplus 000 = 111$$

When it is evident that a single bit has become corrupted, the failing address is identified by exclusive “OR-ing” the old and new ECCo values. The calculation in Figure 5 identifies bit 2 as the problem. The calculation to identify the bad bit uses the ECCo values because they specify the failing bit position directly.

Figure 5: Calculation for Finding Failing Bit Position (Bit 2)

$$ECC_o(\text{old}) \oplus ECC_o(\text{new}) = 010 \oplus 000 = 010$$

When the failing bit is identified, its state is flipped to repair the data packet (Figure 6 on page 4).

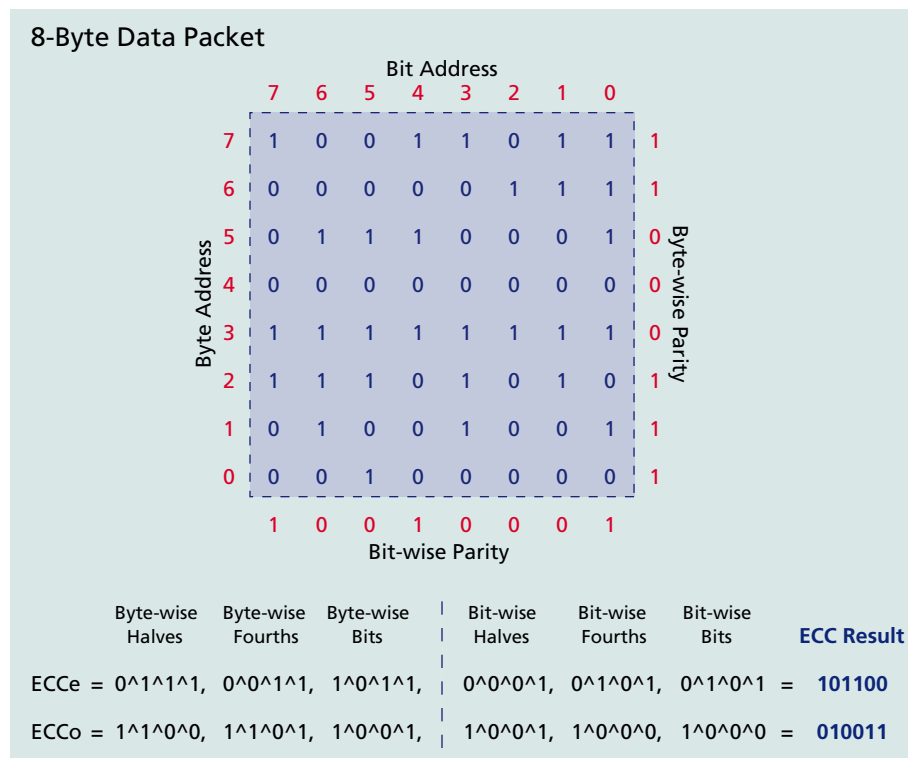
Figure 6: Repairing Corrupted Data to Original State (Bit 2 Flipped)

$$01010101 \oplus 00000100 = 01010001$$

Extension to Larger Data Packets

In the earlier example (“Hamming Code Basics,” Figures 1–5 on pages 2 and 3), 6 bits of ECC are necessary to ensure the integrity of an 8-bit data packet. The overhead in this case is 75 percent—not an overly impressive number. Fortunately, as the size of the data packet grows, the Hamming algorithm becomes increasingly efficient. Each doubling of the data packet requires 2 additional bits of ECC information (i.e., 512 bytes/4,096 bits of data require a pair of 12-bit ECC values—24 bits total). The 24 bits of ECC required for a 512-byte (4,096-bit) data packet will reduce the overhead to 0.06 percent, a much more attractive figure.

Figure 7: ECC Generation on a Byte-Wide Data Packet



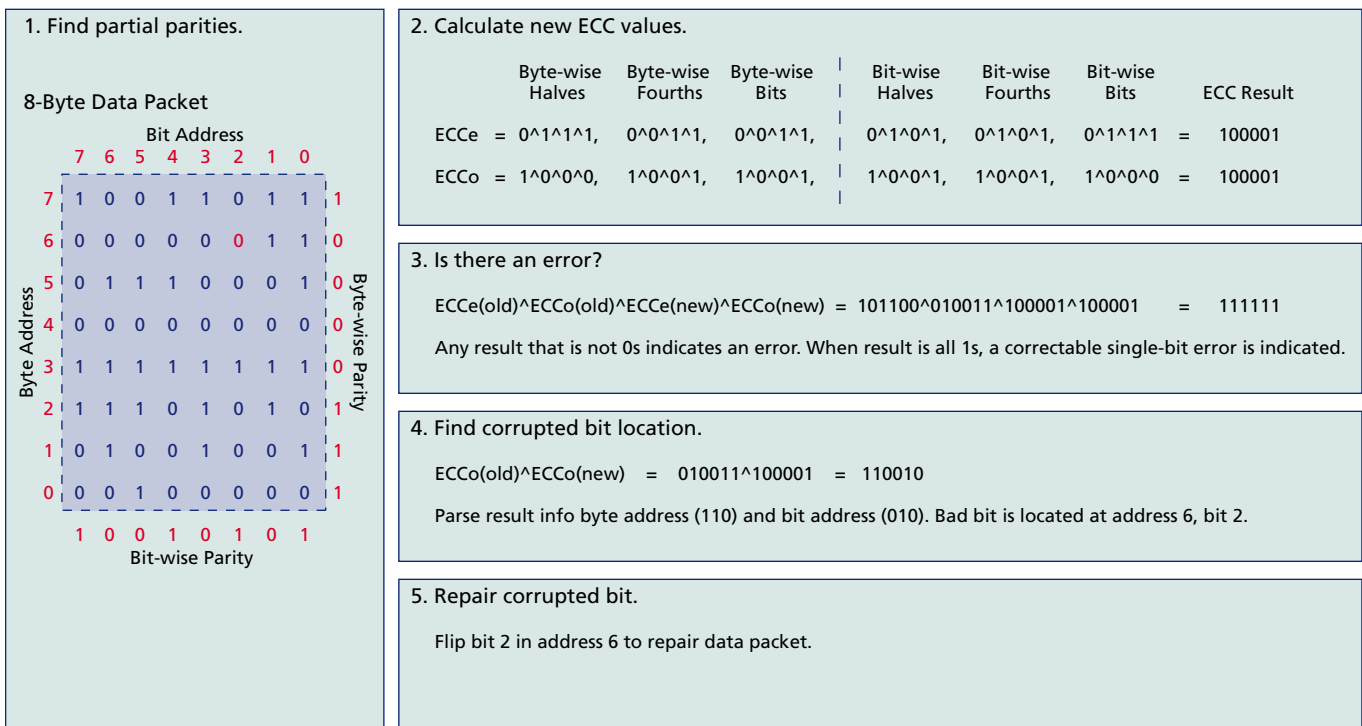
The data packet shown in Figure 7 could be rearranged into a bit-wide data packet and partitioned as shown earlier, but there is a more effective method for data partitioning. Because the XOR function is associative— $(A \wedge B) \wedge C = A \wedge (B \wedge C)$ —the parity calculation can be staged in a more convenient manner.

The staging strategy is implemented with an initial bit-wise and byte-wise calculation before the final ECC values are generated. The byte-wise calculations start with finding the parity of all bits in each byte. The bit-wise calculations determine the parity of the individual bit positions in every byte (i.e., all of the D0s, all of the D1s, etc.). After the bit-

wise and byte-wise parities have been calculated, the data partitioning described in “Hamming Code Basics,” Figures 1–5 on pages 2 and 3, can be performed. The ECC values are then generated from the partitioned bit-wise and byte-wise values. The bit-wise results are positioned in the least significant bits of the resulting ECC values, and the byte-wise results occupy the high-order bits.

Error detection and correction are performed in a manner similar to the method described in “Hamming Code Basics,” Figures 1–5 on pages 2 and 3. In this case, the corrupted bit is identified with a byte-wise address as well as a bit-wise address. Figure 8 shows the process when bit 2 of address 6 (from Figure 7 on page 4) becomes corrupted (1 flipped to 0).

Figure 8: Byte-Wide Data Correction Process



The extension from an 8-byte data packet to a 512-byte packet requires only modifying the size of the data partitions; the algorithm remains the same. Each of the two resulting ECC values will be 12 bits long. The 3 least significant ECC bits represent the 8 bits in each data value, and the 9 most significant ECC bits represent the 512 different addresses.

Accommodating a x16 interface is equally straightforward. Each 16-bit word is partitioned into even and odd groupings of 8, 4, 2, and 1 bit(s) that are used to generate the 4 least significant bits in the generated ECC value. Thus, each ECC value for a 256-word data packet will have 4 bit-wise bits and 8 byte-wise bits (12 bits total, the same as a 512-byte packet).

Conclusion

The use of Hamming codes is relatively straightforward and can easily be implemented in either software or hardware. The limitation to using this algorithm is its limited error correction capabilities. Hamming codes enable single-bit error correction and double-bit error detection. The double-bit error detection capability is useful because it allows for a fall-back strategy to deal with data that is not repairable. Strategies to deal with unrepairable data include repeated reading of the data packet or the use of a secondary ECC algorithm. The random nature of bit failures in NAND Flash devices and the simplicity of the algorithm has made Hamming coding the error correction strategy of choice for SLC NAND Flash-based applications.



8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-3900

prodmktg@micron.com www.micron.com Customer Comment Line: 800-932-4992

Micron, the M logo, and the Micron logo are trademarks of Micron Technology, Inc. All other trademarks are the property of their respective owners.



Revision History

Rev. B	5/07
<ul style="list-style-type: none">• Reworded title and heading.• “Overview” on page 1: Revised description.• “Hamming Code Basics” on page 2: Revised description.• Figure 1 on page 2: Revised values.• Figure 2 on page 2: Revised values.• “Extension to Larger Data Packets” on page 4: Added cross references.• “Conclusion” on page 6: Revised description.	
Rev. A	3/05
<ul style="list-style-type: none">• Initial release.	