Infineon GL-S and GL-T MirrorBit™ Flash Nonvolatile Memory Families – Automatic ECC

About this document

Scope and purpose

This application note provides information on the Automatic Error Correction Code (ECC) feature built into the S29GL-S and S29GL-T families of MirrorBit™ nonvolatile Flash memory devices.

Table of contents

1 Introduction

This document shows how to use the Automatic Error Correction Code (ECC) feature to achieve maximum reliability with the S29GL-S and S29GL-T MirrorBit Flash families. Use this application note in conjunction with the Automatic ECC feature description provided in the S29GL-S and S29GL-T family datasheets.

2 Automatic ECC

This section describes the Automatic ECC feature and shows how the feature responds to erase commands, programming commands, and read operations.

The MirrorBit™ S29GL-S and S29GL-T families have a 256-word (512-byte) Write Buffer for caching data for programming to a particular 256-word Line in the Flash. Each Line in the Flash contains 16 Pages, where each Page contains 16 words (32 bytes). The Flash device automatically programs ECC-related data into hidden locations associated with each Page (ECC Information). For each read operation, the Automatic ECC logic checks the Parity Bits and corrects any single-bit error found within the Page. This helps to ensure that the host system always obtains correct data.

Figure 1 shows a 16-word (32-byte) Page and its ECC Information.

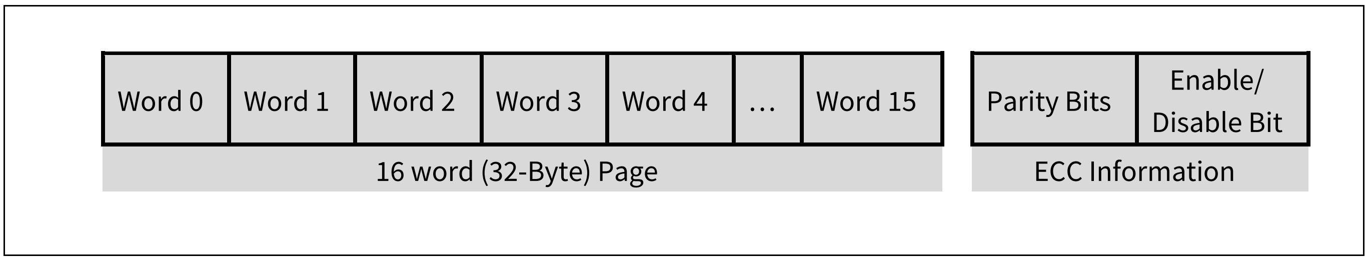

Figure 1 Page data with associated ECC Information

During a Write Buffer Programming operation, the Write Buffer is associated with a particular Line of the Flash memory array. Similarly, during Flash read operations, a Page-sized Read Buffer temporarily holds the Page data that the device reads from the Flash, at which point the device passes the data from the Read Buffer to the host system. The Automatic ECC logic detects and corrects single-bit errors when the device transfers the data from the Page to the Read Buffer using the associated ECC Information. In this way, each system access to the Read Buffer always returns the corrected data.

The Sector Erase command changes each nonvolatile Flash bit in the Sector to the ‘1’ or ‘erased’ state. This operation erases all user-accessible bits within the Sector as well as the ECC Information bits that are not visible to the host system. While the ECC Enable/Disable bit shows that ECC is ‘enabled’ for the page, the Automatic ECC logic performs no corrections upon reading erased Pages.

Assuming that a Sector is in the erased state, a subsequent Write Buffer Programming command copies system-supplied Pages from the Write Buffer to the appropriate Line. The logic computes the ECC Parity Bits for each Page, which are stored in the associated ECC Information area. As this is the first programming operation on these Pages, the ECC Enable/Disable bits show that Automatic ECC remains ‘enabled’ for each programmed Page. When subsequent Flash reads trigger the initial load of the Page to the Read Buffer, ECC corrects, at most, a single-bit error in the Page data and its ECC Parity.

Continuing with a Line that has been programmed, a subsequent Write Buffer programming operation applied to that Line copies the Write Buffer contents to that Line by only changing '1' bits to '0' bits , not '0' bits to ‘1’ bits. Note that '0' bits from any prior programming operation can only be changed to '1' bits via a Flash erase operation.

The logic then computes the new ECC Parity for the Pages within that Line. If programming the new ECC parity requires any forbidden '0' bit to '1' bit transitions, the ECC Enable/Disable bit is ‘disabled’ for that Page. This is the most likely result for the second Write Buffer Programming operation on a Page. A Sector erase is required to change ECC Enable/Disable bits from ‘disabled’ to ‘enabled’.

Otherwise, if programming the new ECC Parity for the Pages within that Line requires only '1' bit to '0' bit transitions, then the updated ECC Parity is programmed to the ECC Information and the ECC Enable/Disable bit remains ‘enabled’. In this way, Automatic ECC may remain enabled for several sequential Write Buffer Programming operations on a Page. This behavior is highly dependent upon the data the host system will program so it is difficult to predict.

The Word Programming command always disables ECC for the Page containing the word.

As the name implies, the Automatic ECC feature automatically manages the Pages where ECC is enabled or disabled and no system intervention is required. The Automatic ECC feature enables you to use Write Buffer Programming and Word Programming in any way that complies with the datasheet with no impact to your existing or new applications.

3 High-Reliability Usage

While no changes to your application are required to use the Automatic ECC feature, there are best practices you may follow if you wish to maximize the fraction of the Flash memory array that has ECC protection. The next two sections describe methods recommended by Infineon to maximize ECC coverage on the Flash memory array.

If the system design goal is to maximize the number of Pages with ECC Enable/Disable bits set to 'enabled', then the design rule is to:

- Never use Write Buffer Programming more than once per Page between Sector erases, and to
- Never use the Word Programming command.

If one follows this design rule, Automatic ECC protects 100% of the Pages in the Flash memory array. This approach is called the *100% ECC Fraction* method.

If your trusted Flash management software does not or cannot comply with these design rules, to avoid costly re-design and re-qualification of that trusted software you may use:

- The 100% ECC Fraction method on as many Flash Sectors and Pages as you can, and then use
- The 100% *Effective* ECC Fraction method on the remaining Flash Pages.

The 100% Effective ECC Fraction method still forbids the use of Word Programming, but insofar as your existing Write Buffer Programming usage may 'disable' the ECC Enable/Disable bits for some Pages, the essence of the method is to use software redundancy methods (like small-payload software ECC or voting) to replace the disabled Automatic ECC on only the affected Pages. This method delivers maximum value for minimum risk to your trusted Flash management software.

3.1 100% ECC Fraction

The best practice for programming the Flash device is to use the Write Buffer Programming command to write full Lines for the entire span of the programmed region as illustrated in **Figure 2**, taking care to program each Line only once after the prior erase. Not only does this practice deliver the fastest programming speed, but it also keeps the ECC Enable/Disable bits 'enabled' for every Page within the programmed region.

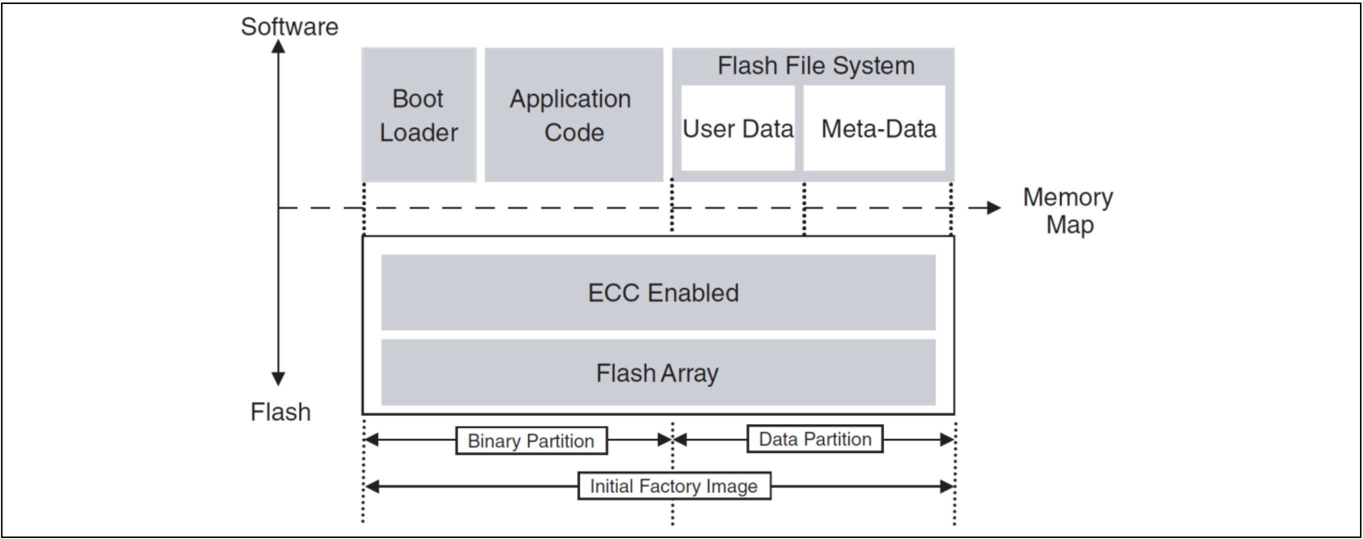

Figure 2 Reference Memory Map for the Case of 100% ECC Fraction

If you need to program data at granularities smaller than a 512-byte Line, programming in granularities that are multiples of a 32-byte Page allows Automatic ECC to remain enabled on each programmed Page. Take care to program each Page only once after the prior erase. Similarly, the best practice for the Data Partition is for your Flash File System (FFS) software to program at granularities no smaller than a 32-byte Page, where the software programs each Page only once after the prior erase.

These programming practices ensure 100% ECC Fraction for the Flash array, where Equation 1 defines ECC Fraction.

$$ECC\ Fraction = \frac{\#\ of\ Pages\ with\ ECC\ enabled}{\#\ of\ Pages\ in\ the\ Flash\ device} \times 100 \qquad \text{Eq. 1}$$

The main challenge with achieving 100% ECC Fraction is that not many Flash file systems or block drivers for NOR or Serial Peripheral Interface (SPI) Flash are designed to operate in single-pass programming mode. NAND file systems comply with this operating mode because many NAND device specifications require it. Indeed it is possible to convert a NAND file system or block driver to run on NOR or SPI Flash so that the Flash can be operated with 100% ECC Fraction. The main task for this effort is aligning all of the Flash format elements with the Page boundaries in the GL-S and GL-T families of Flash devices. Once Page-aligned, the normal single-pass programming methods used for NAND handle the rest. As an example, see **Appendix A: QNX EFTS**.

It is also possible to adapt some file systems and block drivers to achieve 100% ECC Fraction by adjusting some parameters and code to realign the data format elements to be Page-aligned. For the full source code patch showing how Infineon did this for the Linux Memory Technology Device (MTD), see **Appendix B: Linux MTD**.

3.2 100% Effective ECC Fraction

As shown in **Figure 3**, the Binary Partition – the Boot Loader and Application Code – may be programmed in a single pass to ensure 100% ECC Fraction. Typically, the User Data in the FFS can be programmed in a single pass via 512-byte Write Buffer Programming operations, so ECC Enable/Disable bits remain ‘enabled’ for those Pages. However, many trustworthy NOR FFS packages use multi-pass programming for the Meta Data, thereby ‘disabling’ the ECC Enable/Disable bits for those pages. If the FFS is patched to add software redundancy to the Meta Data regions to replace the hardware redundancy that is lost when the ECC Enable/Disable bit is ‘disabled’, then the missing hardware ECC is mitigated with software redundancy. This method is called *100% Effective ECC Fraction*.

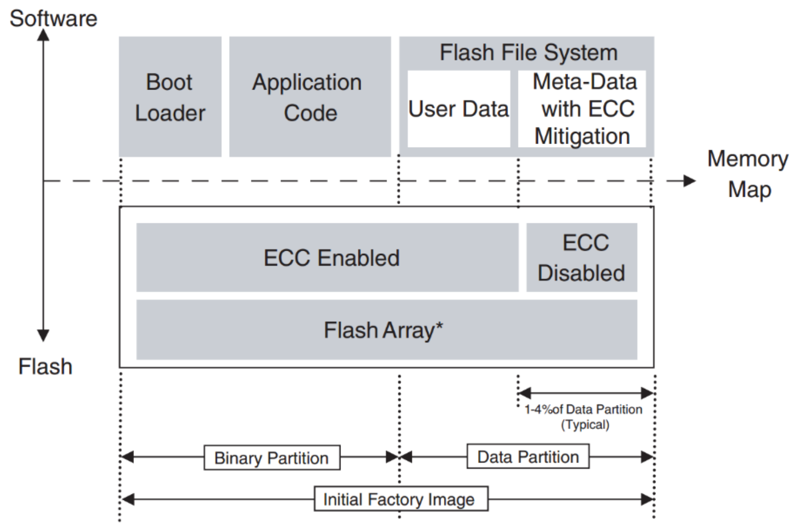

Figure 3 **Reference Memory Map for the Case of 100% Effective ECC Fraction**

Equation 2 defines the Effective ECC Fraction:

$$Effective\ ECC\ Fraction = \frac{\#\ of\ Pages\ with\ ECC\ enabled + \#\ of\ Mitigated\ Pages}{\#\ of\ Pages\ in\ the\ Flash\ device} \times 100 \qquad \text{Eq. 2}$$

Here, a Mitigated Page is a Page where a second Write Buffer Programming operation may have disabled the ECC Enable/Disable bit for that Page, but where the system software provides additional redundancy that replaces the benefit of the disabled Automatic ECC function.

The basic concept underlying this approach is that the overall system does not care whether the data storage subsystem manages the data integrity in hardware or in software. This approach simply replaces the disabled Automatic ECC function with an alternative redundancy managed by the software for all sub-Page writes. You may use any redundancy method so long as the method can return the correct data in the event of a single-bit error within the set of bits used to represent the Meta Data element stored within a Page. This approach can be less disruptive to trusted software as only a minor patch needs to be re-qualified; the core algorithms of the trusted FFS are unaffected.

Note: *This approach affects the backward compatibility of the FFS with prior on-Flash formats due to changes to on-Flash Meta Data structures.*

The discussion that follows illustrates the software redundancy approach with two examples. These are not the only possible methods, but they provide a good starting point for adapting a file system or block driver to 100% Effective ECC Fraction.

3.2.1 Example 1: Software ECC

This example is relevant for logging algorithms where the on-Flash format is a simple list of Meta Data elements, all smaller than a Page, and the host system must write each Meta Data element in a single pass at different times during the execution of a data storage algorithm. This method simply appends a few bits of ECC parity to each Meta Data element. The host system writes the ECC parity bits to the Flash along with the Meta Data, taking care to use only single-pass programming for each ordered pair (Meta Data, ECC Parity). **Figure 4** shows the method in more detail for the case of 'N' Meta Data elements.

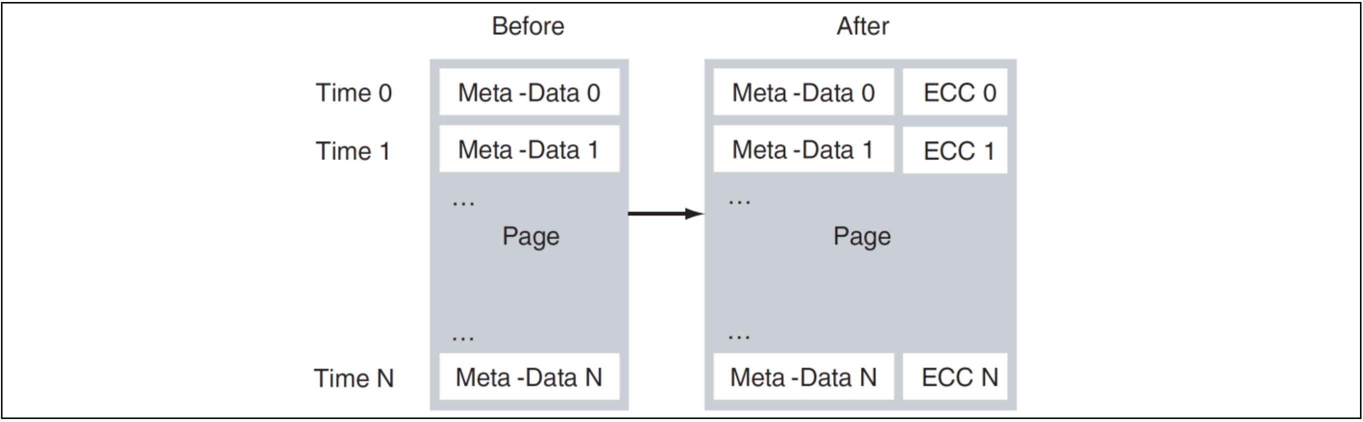

Figure 4 **Mitigating Disabled Automatic ECC with Software ECC**

This approach uses software to replace the disabled Automatic ECC and is able to deliver the correct data in the event of a single-bit error in the Meta Data element or its associated ECC parity data. The data redundancy overhead is minimal given that the software ECC requires the fewest number of extra bits to achieve the required level of bit error protection.

The software encode/decode overhead can be larger for this method compared to others. In practice, the observable encode and programming overhead is small, mainly because the size of the data under software ECC protection is small. Infineon prototypes using this method confirm that impacts to both data storage efficiency and programming overhead can be virtually undetectable. However, the software decode step may degrade the net read speed.

See **Appendix C: Small Payload Hamming ECC** for the ECC source code that uses one byte of ECC parity to protect seven bytes of user data.

3.2.2 Example 2: Three Copy Voting

In contrast to the Software ECC method, the Three Copy Voting method is appropriate for cases where your algorithm overwrites a Meta Data storage location one or more times during bit-walking operations. This method takes a single copy of the Meta Data element that the host system overwrites multiple times, and transforms it into three identical copies of the Meta Data element. The host system overwrites the three copies for each update to the Meta Data element. If any two of the Meta Data copies agree during a later read operation, then the host system passes the value contained in these two identical copies to the application as the "true" value of the Meta Data element.

As shown in **Figure 5**, the host system replaces each bit of the data content in the original Meta Data element (MD) with a new Meta Data element comprising three copies of MD. These contain three bits of storage for each bit of data content in the original single copy of the Meta Data element. If any one of these three bits is erroneous, the method is capable of returning the "true" value of that bit. When applied bit-wise to the three copies of the Meta Data element, one copy of every bit in the three-copy Meta Data element can be erroneous and the method still returns the "true" value of the Meta Data element. Given that the bit-wise approach delivers much more redundancy than is actually required to replace the disabled Automatic ECC, software overhead can be minimized by decoding the Meta Data value via the copy-wise method.

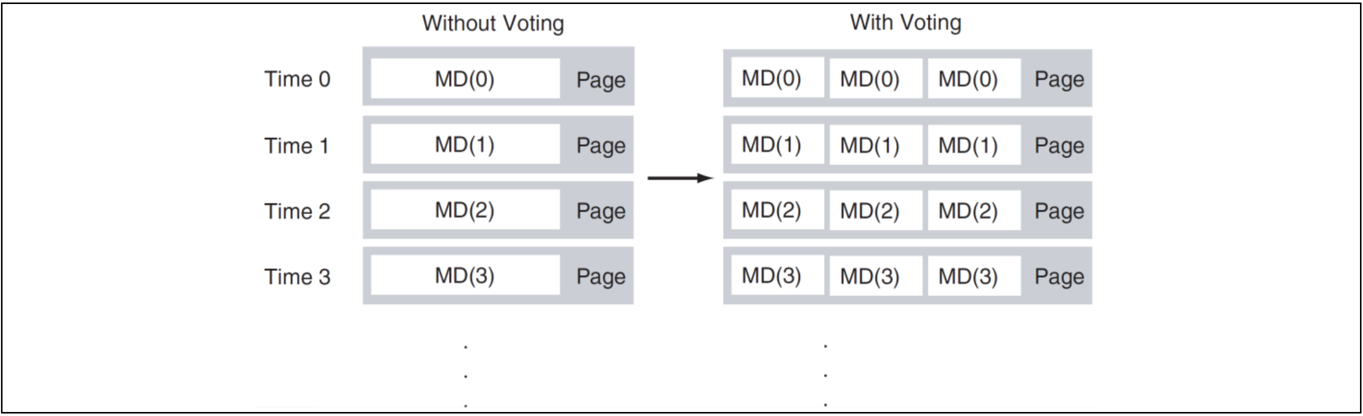

Figure 5 Mitigating Disabled Automatic ECC with Three Copy Voting

When the copy-wise decoding method is used, the Three Copy Voting method is capable of withstanding a single-bit error within the scope of the three copies of that Meta Data element. The data redundancy overhead per metadata element is rather large at 300%, but these types of Meta Data elements are normally small (word-sized) so the impact to data storage efficiency is likely to be small. The software overhead for a copy-wise decoding algorithm is also small, but the impact to read speed may be noticeable. Infineon prototypes using this method confirm that impacts on both data storage efficiency and programming overhead can be virtually undetectable.

4 High-Reliability Ecosystem Support

Table 1 provides a list of data storage solutions that, at the time of publication, support your goal of 100% ECC Fraction or 100% Effective ECC Fraction.

Table 1 Data Storage Options as of the Time of Publication

Supplier	File System	<100% ECC Fraction	100% Effective ECC Fraction	100% ECC Fraction
Infineon[1]	FFS – NOR/SPI/NAND	-	-	Available Now
Infineon[2,3]	Linux MTD	Available Now	-	Available Now
Infineon[2]	Microsoft Flash PDD (WinCE 6 Rev. 2, WinCE 7)	-	-	Available Now
Infineon[2]	Microsoft FMD (WinCE 5, 6, and 7)	Available Now	-	-
Blunk Microsystems[4]	LiteFS-NOR	-	-	Available Now
Blunk Microsystems[4]	TargetFAT and TargetFTL-NOR	-	-	Available Now
Datalight[4]	FlashFX Family	Available Now	Available Now	-
Kyoto Software Research (KSR)[4]	Fugue	Available Now	-	Available Now
QNX[4]	QNX FFSv3 (6.3 and 6.5)	Available Now	Available Now	-

Note:

1. *Visit **www.cypress.com** to register for the FFS source code package.*
2. *Visit **www.cypress.com** to download the free source code package.*
3. *The MTD patch also configures the kernel for proper operation. See **Appendix B: Linux MTD** for details.*
4. *Check with the file system supplier for more information.*

5 Conclusion

The GL-S and GL-T families of Flash devices from Infineon have an Automatic ECC feature that is completely transparent for normal modes of Flash operation. Nearly all existing consumer and industrial applications can migrate to the 65-nm or 45-nm Flash without any special software considerations. This application note also shows how to adapt automotive or high-reliability applications to achieve either 100% ECC Fraction or 100% Effective ECC Fraction.

6 Appendix A: QNX EFTS

ETFS is a transaction-based Flash file system where each transaction consists of writing one or more clusters. Each cluster consists of a Header segment and a Data segment, as shown in **Figure 6**.

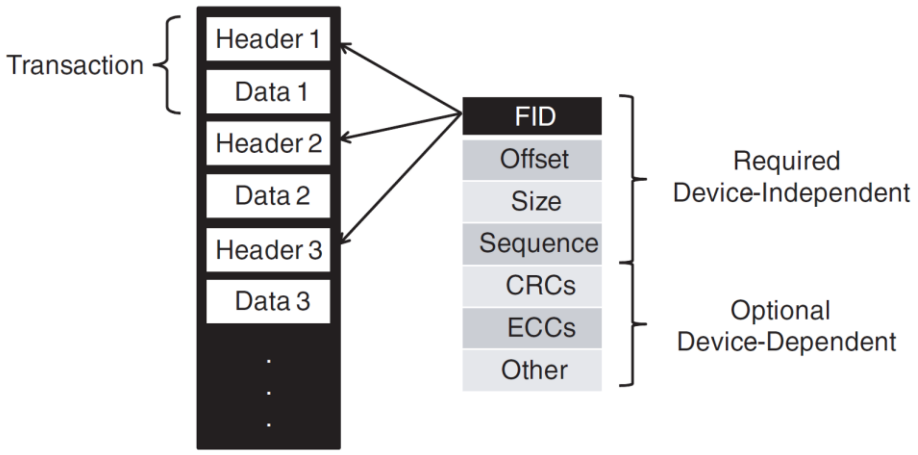

Figure 6 Clusters Consisting of Ordered Pairs of Header and Data Saved by EFTS as Transactions

The driver can set the cluster size at the initialization time. The cluster size for a large-page NAND is 2 KB, and the cluster size for the ETFS driver adapted to GL-S and GL-T Flash memories is 1 KB. On NAND, transaction headers are stored in the spare area. For adaptation to GL-S and GL-T Flash, transaction headers are stored at the end of the Flash sectors as shown in **Figure 7**.

- This enables each 128-KB Flash sector to hold 126 clusters for GL-S and GL-T Flash memories without Automatic ECC, whereas
- Page-aligning the headers reduces the cluster count to 124 per a 128-KB Flash sector for GL-S and GL-T Flash memories with Automatic ECC.

The net result of this adaptation is a driver that can work with optimal data storage efficiency on older generations of Infineon GL Family Flash devices and with slightly reduced data storage efficiency, but 100% ECC Fraction, on 45-nm and 65-nm GL Flash devices.

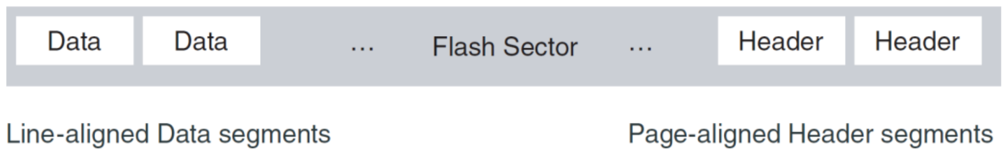

Figure 7 Cluster Arrangement for EFTS Adapted to 45-nm or 65-nm Flash

7 Appendix B: Linux MTD

The Linux MTD patch, provided in the boxed section below and available on the website, lays the foundation for a data storage solution with 100% ECC Fraction. The patch enables JFFS2, UBIFS, and other MTD-based Flash file systems to meet high-reliability requirements when used with GL-S and GL-T Flash devices.

- This patch modifies the AMD/Cypress driver so that the CFI field for the process technology is checked, and “Sibley Mode” is enabled for 65-nm and later Flash devices.
- The patch also enables FFS buffering via a 512-byte buffer for 65-nm and later Flash devices; this ensures that all writes are no smaller than a Line.
- Because the original code uses Word Programming for writes of four bytes or less, this patch also changes the programming algorithms to 100% Write Buffer Programming for writes for all sizes and for all Flash technologies.

Infineon tests show this patch is backward-compatible with Flash devices based on 65-nm and earlier technologies. In addition, the patch enables 100% ECC Fraction for devices based on 65-nm and later technologies.

```
Diff -rupN linux-2.6.23.i586/drivers/mtd/chips/cfi_cmdset_0002.c linux-2.6.23.i586-
gls/drivers/mtd/chips/cfi_cmdset_0002.c
--- linux-2.6.23.i586/drivers/mtd/chips/cfi_cmdset_0002.c2007-10-09
22:31:38.000000000 +0200
+++ linux-2.6.23.i586-gls/drivers/mtd/chips/cfi_cmdset_0002.c2009-03-17
12:13:03.000000000
+0100
@@ -161,9 +161,17 @@ static void fixup_use_write_buffers(stru
{
   struct map_info *map = mtd->priv;
   struct cfi_private *cfi = map->fldrv_priv;
+ struct cfi_pri_amdstd *extp = cfi->cmdset_priv;
+
   if (cfi->cfiq->BufWriteTimeoutTyp) {
   DEBUG(MTD_DEBUG_LEVEL1, "Using buffer write method\n" );
   mtd->write = cfi_amdstd_write_buffers;
+
```

For 65-nm and later Flash devices, you must change the 'write size' to 512 bytes and disable the 'bit writable' flag. This ensures that the host writes data in 512-byte segments via single-pass programming. You can use other buffer sizes in Page multiples; however, Infineon testing finds that 512 bytes provides the best throughput.

```
+ if (extp->SiliconRevision >= 0x1C) {
+   mtd->writesize = 512;
+   mtd->flags &= ~MTD_BIT_WRITEABLE;
+   printk(KERN_INFO "Enabling Spansion 65nm mode, writesize = 512 bytes\n");
+ }
  }
```

```
}
```

The following changes ensure the writes are 'bus-aligned'.

```
@@ -1232,9 +1240,6 @@ static int cfi_amdstd_write_words(struct
}

-/*
- * FIXME: interleaved mode not tested, and probably not supported!
- */
   static int __xipram do_write_buffer(struct map_info *map, struct flchip *chip,
              unsigned long adr, const u_char *buf,
              int len)
@@ -1245,8 +1250,8 @@ static int __xipram do_write_buffer(stru
   unsigned long uWriteTimeout = ( HZ / 1000 ) + 1;
   int ret = -EIO;
   unsigned long cmd_adr;
-  int z, words;
-  map_word datum;
+  int z, words, prolog, epilog, buflen = len;
+  map_word datum, pdat, edat;

adr += chip->start;
   cmd_adr = adr;
@@ -1267,6 +1272,20 @@ static int __xipram do_write_buffer(stru
   ENABLE_VPP(map);
   xip_disable(map, chip, cmd_adr);
+  /* If start is not bus-aligned, prepend old contents of flash */
+  prolog = (adr & (map_bankwidth(map)-1));
+  if (prolog) {
+  adr -= prolog;
+  len += prolog;
+  pdat = map_read(map, adr);
+  }
+  /* If end is not bus-aligned, append old contents of flash */
+  epilog = ((adr + len) & (map_bankwidth(map)-1));
+  if (epilog) {
+  len += map_bankwidth(map)-epilog;
+  edat = map_read(map, adr + len - map_bankwidth(map));
+  }
+
   cfi_send_gen_cmd(0xAA, cfi->addr_unlock1, chip->start, map, cfi,
   cfi->device_type, NULL);
   cfi_send_gen_cmd(0x55, cfi->addr_unlock2, chip->start, map, cfi,
   cfi->device_type, NULL);
   //cfi_send_gen_cmd(0xA0, cfi->addr_unlock1, chip->start, map, cfi,
   cfi->device_type,NULL);
@@ -1281,16 +1300,24 @@ static int __xipram do_write_buffer(stru
   map_write(map, CMD(words - 1), cmd_adr);
   /* Write data */
   z = 0;
+  if (prolog) {
+      datum = map_word_load_partial(map, pdat, buf, prolog,
+              min_t(int, buflen, map_bankwidth(map) - prolog));
+      map_write(map, datum, adr);
+
+      z += map_bankwidth(map);
+      buf += map_bankwidth(map) - prolog;
+  }
   while(z < words * map_bankwidth(map)) {
-      datum = map_word_load(map, buf);
+      if (epilog && z >= (words-1) * map_bankwidth(map))
+              datum = map_word_load_partial(map, edat, buf, 0, epilog);
+      else
+              datum = map_word_load(map, buf);
      map_write(map, datum, adr + z);

      z += map_bankwidth(map);
      buf += map_bankwidth(map);
   }
-  z -= map_bankwidth(map);
-
-  adr += z;

   /* Write Buffer Program Confirm: GO GO GO */
   map_write(map, CMD(0x29), cmd_adr);
@@ -1331,8 +1358,10 @@ static int __xipram do_write_buffer(stru

   /* reset on all failures. */
   map_write( map, CMD(0xF0), chip->start );
+  cfi_send_gen_cmd(0xAA, cfi->addr_unlock1, chip->start, map, cfi,
   cfi->device_type, NULL);
+  cfi_send_gen_cmd(0x55, cfi->addr_unlock2, chip->start, map, cfi,
   cfi->device_type, NULL);
+ cfi_send_gen_cmd(0xF0, cfi->addr_unlock1, chip->start, map, cfi,
  cfi->device_type, NULL);
```

```
    xip_enable(map, chip, adr);
-   /* FIXME - should have reset delay before continuing */
     printk(KERN_WARNING "MTD %s(): software timeout\n",
       __func__ );
```

The following change removes the conditional Word Programming logic and replaces it with unconditional Write Buffer Programming.

```
@@ -1364,36 +1393,12 @@ static int cfi_amdstd_write_buffers(stru
    chipnum = to >> cfi->chipshift;
    ofs = to - (chipnum << cfi->chipshift);

-   /* If it's not bus-aligned, do the first word write */
-   if (ofs & (map_bankwidth(map)-1)) {
-        size_t local_len = (-ofs)&(map_bankwidth(map)-1);
-        if (local_len > len)
-               local_len = len;
-        ret = cfi_amdstd_write_words(mtd, ofs + (chipnum<<cfi->chipshift),
-                      local_len, retlen, buf);
-        if (ret)
-          return ret;
-        ofs += local_len;
-        buf += local_len;
-        len -= local_len;
-
-        if (ofs >> cfi->chipshift) {
-           chipnum ++;
-           ofs = 0;
-           if (chipnum == cfi->numchips)
-               return 0;
-        }
-   }
-
-   /* Write buffer is worth it only if more than one word to write... */
-   while (len >= map_bankwidth(map) * 2) {
+   while (len) {
        /* We must not cross write block boundaries */
        int size = wbufsize - (ofs & (wbufsize-1));

        if (size > len)
               size = len;
-       if (size % map_bankwidth(map))
-              size -= size % map_bankwidth(map);

               ret = do_write_buffer(map, &cfi->chips[chipnum],
                      ofs, buf, size);
@@ -1413,16 +1418,6 @@ static int cfi_amdstd_write_buffers(stru
               }
        }

-  if (len) {
-       size_t retlen_dregs = 0;
-
-       ret = cfi_amdstd_write_words(mtd, ofs + (chipnum<<cfi->chipshift),
-                      len, &retlen_dregs, buf);
-
-       *retlen += retlen_dregs;
-       return ret;
-  }
-
   return 0;
}
```

8 Appendix C: Small Payload Hamming ECC

This appendix provides the code used to perform software ECC management. The code inserted below shows that this set of two functions operates on an 8-byte region where the code allocates 7 bytes to user data, and allocates 1 byte for ECC parity bits. Execution of the code is fast and not noticeable in a normal file system operation because this algorithm manages only eight bytes.

```
// (63,57) Hamming code routines:
// ==============================
// Data: 57 bit (1..7 byte), parity: 6 bit, 1 bit error correction

#ifndef __F3S_HAMMING_H_INCLUDED
#define __F3S_HAMMING_H_INCLUDED

//
// Compute and return Hamming parity, len specifies the byte count (1..7)
//
unsigned char compute_hamming(const void* data, int len);

//
// Correct Hamming data and parity, len specifies the data byte count
//
void correct_hamming(void* data, int len, unsigned char* parity);


#endif /* __F3S_HAMMING_H_INCLUDED */

unsigned char xor[57] = {
   0x03, 0x05, 0x06, 0x07, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x11,
   0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d,
   0x1e, 0x1f, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a,
   0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36,
   0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f };

unsigned char p0[8] = { 0x00, 0xfc, 0xe1, 0xe1, 0xe7, 0xef, 0xf7, 0xff };

unsigned char errorbit[64][2] = {
   { 0, 0x00 }, { 8, 0x01 }, { 8, 0x02 }, { 0, 0x01 }, { 8, 0x04 }, { 0, 0x02 },
   { 0, 0x04 }, { 0, 0x08 }, { 8, 0x08 }, { 0, 0x10 }, { 0, 0x20 }, { 0, 0x40 },
   { 0, 0x80 }, { 1, 0x01 }, { 1, 0x02 }, { 1, 0x04 }, { 8, 0x10 }, { 1, 0x08 },
   { 1, 0x10 }, { 1, 0x20 }, { 1, 0x40 }, { 1, 0x80 }, { 2, 0x01 }, { 2, 0x02 },
   { 2, 0x04 }, { 2, 0x08 }, { 2, 0x10 }, { 2, 0x20 }, { 2, 0x40 }, { 2, 0x80 },
   { 3, 0x01 }, { 3, 0x02 }, { 8, 0x20 }, { 3, 0x04 }, { 3, 0x08 }, { 3, 0x10 },
   { 3, 0x20 }, { 3, 0x40 }, { 3, 0x80 }, { 4, 0x01 }, { 4, 0x02 }, { 4, 0x04 },
   { 4, 0x08 }, { 4, 0x10 }, { 4, 0x20 }, { 4, 0x40 }, { 4, 0x80 }, { 5, 0x01 },
   { 5, 0x02 }, { 5, 0x04 }, { 5, 0x08 }, { 5, 0x10 }, { 5, 0x20 }, { 5, 0x40 },
   { 5, 0x80 }, { 6, 0x01 }, { 6, 0x02 }, { 6, 0x04 }, { 6, 0x08 }, { 6, 0x10 },
   { 6, 0x20 }, { 6, 0x40 }, { 6, 0x80 }, { 7, 0x01 } };


unsigned char compute_hamming(const void* data, int len)
{
   unsigned char d, *x = xor, p = p0[len];
   int i;

   while (len--) {
       d = *(unsigned char*)data++;
       for (i = 0; i < 8; i++) {
              if (d & 0x01)
                     p ^= *x;
              x++;
                     d >>= 1;
              }
       }
       return p;
}
void correct_hamming(void* data, int len, unsigned char* parity)
{
   unsigned char *tdata = (unsigned char*)data;
   unsigned char d, *x = xor, p = p0[len] ^ (*parity |= 0xc0);
   int i;

   while (len--) {
       d = *(unsigned char*)data++;
       for (i = 0; i < 8; i++) {
              if (d & 0x01)
                     p ^= *x;
              x++;
                     d >>= 1;
       }
   }

   if (p) {
       if (errorbit[p][0] == 8)
              *parity ^= errorbit[p][1];
       else
              *(tdata + errorbit[p][0]) ^= errorbit[p][1];
       }
}
```

References

[1] **001-98285** - 3.0V GL-S Flash Memory Family, S29GL01GS 1 Gbit, S29GL512S 512 Mbit, S29GL256S 256 Mbit, S29GL128S 128 Mbit Datasheet

[2] **001-98286** - 3.0V GL-S Flash Memory, S29GL064S 64 Mbit Datasheet

[3] **001-98296** - S70GL02GS 2 Gbit (256 Mbyte) 3.0V Flash Memory Datasheet

[4] **002-00247** - S29GL01GT 1 Gbit and S29GL512T 512 Mbit Parallel NOR Flash Datasheet

Revision history

Document version	Date of release	Description of changes
**	2011-05-16	New Spec.
*A	2012-04-17	Updated Programming Guide Example 1: Software ECC. Example 2: Three Copy Voting.
*B	2013-03-18	Updated 'Data Storage Options as of Publication' table.
*C	2016-03-28	Data Storage Options as of Publication table: Added a Blunk Microsystems File System.
*D	2015-12-09	Updated to template.
*E	2016-10-25	Added Associated Part Family as "S29GL-S and S29GL-T" in page 1. Added S29GL-T related information in all instances across the document. Updated to new template.
*F	2017-07-14	Updated logo and copyright.
*G	2018-04-30	Updated Related Documents: Removed reference of spec 002-03239. Updated to new template. Completing Sunset Review.
*H	2021-04-23	Updated to Infineon template

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2021-04-23

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to www.cypress.com/support

Document reference

002-00621 Rev. *H