AN200621

赛普拉斯 GL-S 和 GL-T Mirrorbit® Flash 非易失性存储器系列 — 自动 ECC

作者：Max Willis

相关器件系列：S29GL-S 和 S29GL-T

AN200621 描述了 MirrorBit®非易失性 Flash 存储器 S29GL-S 和 S29GL-T 系列中的内置自动纠错码（ECC）特性。

目录

1 简介

本文档说明了如何使用 S29GL-S 和 S29GL-T MirrorBit Flash 系列的自动纠错码（ECC）特性，从而使器件最可靠。除了参考本应用笔记外，您还可以查找 S29GL-S 和 S29GL-T 系列数据手册中所提供的自动 ECC 特性说明内容。

2 自动纠错码（ECC）

本部分介绍了自动 ECC 特性，并描述了该特性对擦除命令、编程命令和读取操作的响应。

赛普拉斯 MirrorBit® S29GL-S 和 S29GL-T 系列具有 256 字（512 字节）写缓冲区，用于缓存将被编程到 Flash 中特定 256 字行的数据。Flash 中的每一行包括 16 页，每页 16 字（32 字节）。Flash 器件自动将 ECC 相关的数据编程到每一页相对应的隐藏位置（ECC 信息）。每次执行读取操作时，自动 ECC 逻辑都会检查奇偶校验位，然后纠正在每页中发现的单比特错误。这样做可以确保主系统始终获得正确数据。

图 1 显示的是大小为 16 字（32 字节）的页面及其 ECC 信息。

图 1. 页面数据及其相关联的 ECC 信息

字 0	字 1	字 2	字 3	字 4	…	字 15	奇偶校验位	使能/禁用位

16 字（32 字节）大小的页面	ECC 信息

在写缓冲区编程操作中，写缓冲区与 Flash 存储器阵列中的特定行相关联。同样，在 Flash 读取操作中，器件从 Flash 读取到的页数据被暂时存储在页大小的读缓冲区内，然后这些数据被传送给主系统。器件使用 ECC 信息将相应页数据传送到读缓冲区时，自动 ECC 逻辑将检测并纠正单比特错误。因此，对读缓冲区进行的访问操作总会返回正确数据。

扇区擦除命令将非易失性 Flash 扇区中每一位的状态改为 1 或“被擦除”状态。该操作将擦除扇区中所有用户可访问的位以及主系统不可见的 ECC 信息位。虽然 ECC 使能/禁用位表示 ECC 已经被使能，但是自动 ECC 逻辑在读取被擦除的页面时不会进行纠正操作。

假定某个扇区被擦除，则随后的写缓冲区编程命令会将写缓冲区中由系统提供的页数据复制到合适的行内。自动 ECC 逻辑计算每页的奇偶校验位，这些位被存储在相应的 ECC 信息区中。由于这是对这些页面进行的第一个编程操作，所以 ECC 使能/禁用位会表示自动 ECC 可用于每个编程页。后续的 Flash 读取操作将页数据首次加载到读缓冲区时，ECC 将纠正页中的单比特错误及其 ECC 奇偶校验位。

对于某个已被编程的行，针对该行进行的写缓冲区编程操作会将位的状态从 1 改为 0（不是从 0 改为 1），从而可以将写缓冲区内容复制到该行上。请注意，通过 Flash 擦除操作，可以将以前任何编程操作的 0 位改为 1。

然后自动 ECC 逻辑计算行中每一页的新 ECC 奇偶校验位。如果编程新 ECC 奇偶校验时需要将 0 位转为 1（该操作被禁止），该页的 ECC 使能/禁用位被禁用。这是进行第二次写缓冲区编程最可能发生的结果。要想将 ECC 使能/禁用位的禁用状态改为使能状态，则需要通过扇区擦除操作。

如果编程行中各页的新 ECC 奇偶校验只需要将 1 位改为 0，那么新的 ECC 奇偶校验被编程到 ECC 信息区内，并且 ECC 使能/禁用位仍保持“使能”状态。这样，自动 ECC 可以在某个页的连续几个写缓冲区编程操作过程中保持“使能”状态。这种行为很明显依赖于主系统将要编程的数据，因此很难预测。

字编程命令一直禁用包含该字的页的 ECC。

顾名思义，自动 ECC 特性自动管理各页面的 ECC 使能/禁用情况，不需要系统干预。通过自动 ECC 特性，您可以根据符合数据手册的方式任意使用写缓冲区编程和字编程，而不会影响当前或新的应用。

3 高可靠性用法

虽然使用自动 ECC 功能不需要对应用进行任何更改，但是为了最大化受 ECC 保护的 Flash 存储器阵列，需要使用一定的优化方案。下面两部分将介绍赛普拉斯建议使用的最可靠的 Flash 存储器阵列的 ECC 保护的方法。

如果系统设计要求 ECC 使能/禁用位为“使能”状态的页数量最多，请遵守以下设计规则：

- 在各扇区擦除操作间，每页只能使用一次写缓冲区编程操作
- 不要使用字编程命令。

如果遵循这个设计规则，则自动 ECC 能够 100%保护 Flash 存储器阵列中的页面。这种方法称为 *100% ECC 分数法*。

如果您选用的 Flash 管理软件并非或不能符合这些设计规则，那么为了避免重新设计和重新认证，您可以：

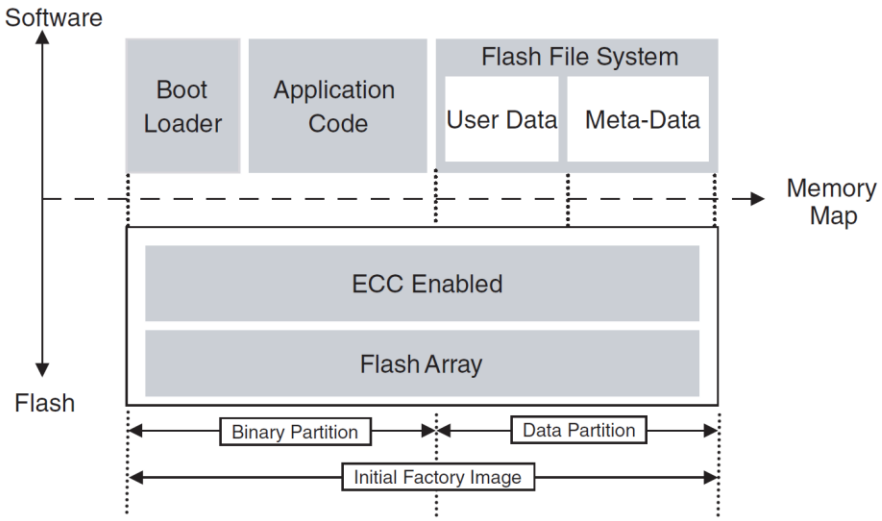
- 将 100% ECC 分数法应用于尽可能多的 Flash 扇区和页面
- 将 100%*有效* ECC 分数法应用于 Flash 中剩下的页面。

100%有效 ECC 分数法仍会禁用字编程，但只要现有的写缓冲区编程操作禁用了某些页面的 ECC 使能/禁用位，您可以使用软件冗余方法（如小型有效载荷软件 ECC 或表决方法）来替换受影响页面上被禁用的自动 ECC。使用这种方法，您所选用的 Flash 管理软件的有效性最大，而风险最小。

3.1 100% ECC 分数法

编程 Flash 器件的最佳实践是使用写缓冲区编程命令对整个编程区域的全部行进行写操作（如图 2 所示）。这样可以确保在每个擦除操作后只能对一行执行一次编程操作。该方法不仅编程速度最快，而且编程区域中的所有页的 ECC 使能/禁用位都保持“使能”状态。

图 2. 100% ECC 分数法的参考存储器映射图



如果您需要编程粒度小于 512 字节行的空间，则器件支持执行大小为 32 字节页倍数的编程粒度。这样，每个编程页的自动 ECC 仍保持“使能”状态。请注意，每个擦除操作后只能对某一页进行一次编程操作。同样，数据分区的最佳实践是将 Flash 文件系统（FFS）软件的编程粒度大小至少设置为 32 字节页，这样才能满足在擦除操作后只对某一页进行一次编程。

这些编程实践能够确保 Flash 阵列得到 100% ECC 分数。ECC 分数的定义如公式 1 所示。

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

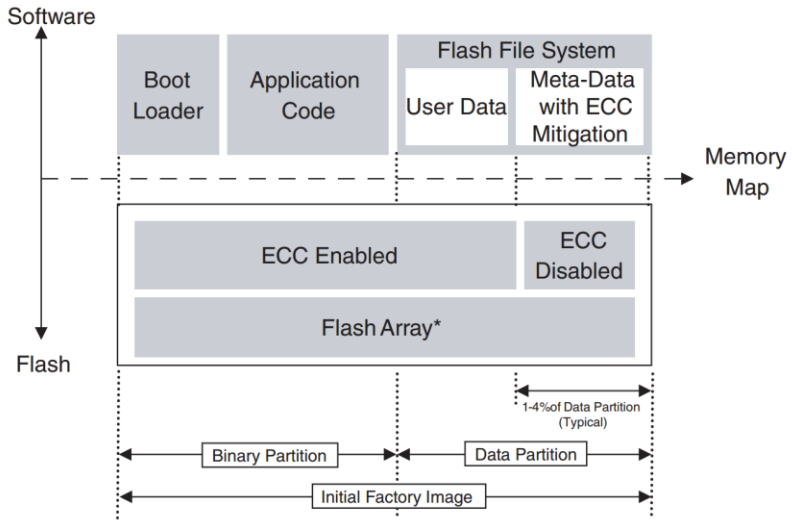
获得 100% ECC 分数的主要困难是 NOR Flash 或串行外设接口（SPI）Flash 的大多数 Flash 文件系统或模块驱动程序并非是针对单次编程模式而设计的。NAND 文件系统符合该操作模式，这是因为很多 NAND 器件规范都有相同的要求。实际上，可以将 NAND 文件系统或模块驱动程序转换成为运行于 NOR 或 SPI Flash 上的部分，从而 Flash 可以获得 100% ECC 分数。该操作主要是为了使所有 Flash 格式元素与 Flash 器件的 GL-S 和 GL-T 系列中的页边界对齐。一旦页面对齐，则用于 NAND 的正常单次编程方法会处理剩余工作。请查阅附录 A：QNX EFTS，参考示例。

还可以调整文件系统和模块驱动程序的某些参数和代码，使其数据格式元素成为页对齐，从而获得 100% ECC 分数。欲了解赛普拉斯针对 Linux 存储器技术器件（MTD）执行该操作的完整源代码修补，请参见附录 B：Linux MTD。

3.2 100%有效 ECC 分数法

如图 3 所示，二进制分区 — 引导加载程序和应用代码 — 可以单次编程，从而确保获得 100% ECC 分数。通常，FFS 中的用户数据可以通过 512 字节写缓冲区编程操作进行单次编程，因此这些页的 ECC 使能/禁用位可以保持“使能”状态。然而，许多值得信赖的 NOR FFS 数据包使用元数据的多次编程，因此“禁用”了这些页的 ECC 使能/禁用位。如果 FFS 被修补，从而向元数据区域添加软件冗余（用于替换当 ECC 使能/禁用位被禁用时丢失的硬件冗余），那么使用软件冗余来替换丢失的硬件 ECC。该方法被称为 *100%有效 ECC 分数法*。

图 3. 100%有效 ECC 分数法的参考存储器映射图



100%有效 ECC 分数法的定义如公式 2 所示：

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

公式 2

这里的替换页是一个特殊页，其中第二个写缓冲区编程操作可能禁用了该页的 ECC 使能/禁用位，但是系统软件提供额外的冗余来代替禁用自动 ECC 功能的好处。

这种方法的基本概念是：整个系统不关心数据存储子系统通过硬件还是软件来管理数据完整性。通过这种方法，可以简单地使用软件管理的备用冗余替代所有子页写入的被禁用自动 ECC 功能。您可以使用任何冗余方法，但只有在用于表示存储在页面中的元数据元素的一组位中出现单比特错误的情况下，该方法才可返回正确的数据。该方法对选用软件的破坏性较小，因为只需要进行很少更改。已选用的 FFS 的核心算法不受影响。

注意：由于更改了 on-Flash 元数据结构，因此，该方法会影响 FFS 与先前的 on-Flash 格式的向后兼容性。

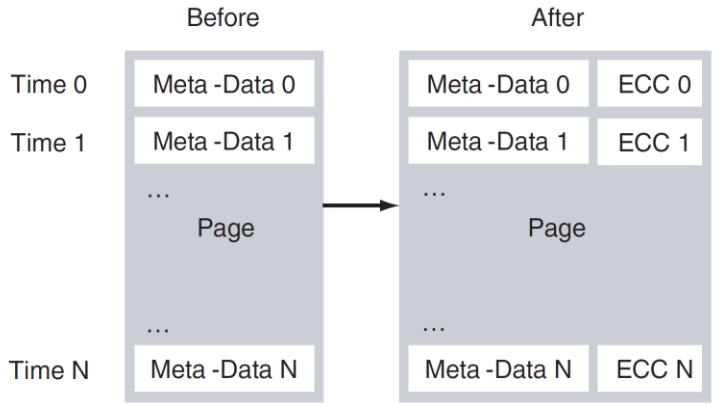
下面的内容将通过两个示例描述了软件冗余方法。这些不是唯一可能的方法，但需要调整 100%有效 ECC 分数法中的文件系统或模块驱动程序时，它们可以提供一个良好的起点。

3.2.1 示例 1：软件 ECC

该示例与记录算法相关，其中 on-Flash 格式是小于页面的元数据元素的简单列表，并且在执行数据存储算法期间，主机系统必须在不同时间单次编程每个元数据元素。该方法将几个 ECC 奇偶校验位简单地附加到每个元数据元素中。主

机系统将 ECC 奇偶校验位与元数据一同写入到 Flash 内。请注意，写入每个有序对（元数据，ECC 奇偶校验）时仅使用单次编程操作。图 4 详细介绍了在有 N 个元数据元素情况下的执行方法。

图 4. 使用软件 ECC 替换被禁用的自动 ECC



该方法使用软件来替换禁用的自动 ECC，并且在元数据元素或其相关联的 ECC 奇偶校验数据中存在单比特错误的情况下能够提供正确数据。数据冗余的开销最小。为了获得所需的比特错误保护级别，软件 ECC 需要的额外位数最少。

与其它方法相比，该方法的软件编码/解码开销可能更大。在实践中，可见的编码和编程开销很小，主要是因为需要软件 ECC 保护的数据很小。赛普拉斯原型机使用该方法，并确认该方法对数据存储效率和编程开销的影响非常小。软件解码步骤却降低了净读速度。

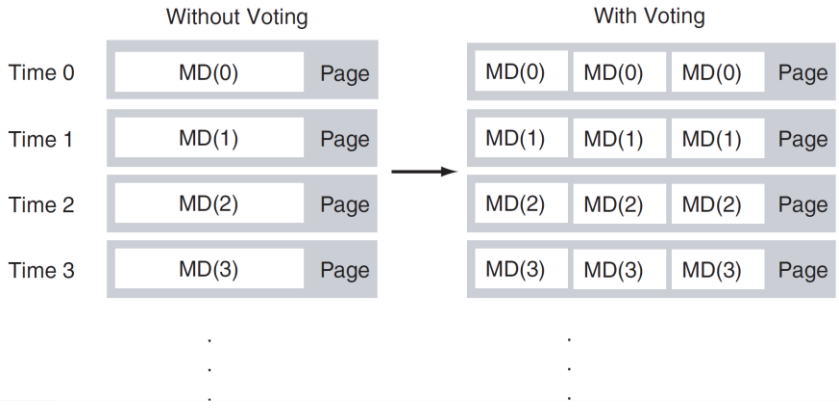
请参考附录 C：小型有效载荷 Hamming ECC，了解如何使用一个 ECC 奇偶校验字节来保护 7 个用户数据字节的 ECC 源代码。

3.2.2 示例 2：三副本表决方法

与软件 ECC 方法相反，三副本表决的方法适用于以下情况：您的算法在位移操作期间单次或多次覆盖了元数据存储位置。该方法可以获取主机系统多次覆盖的元数据元素的单个副本，并将其复制为三个相同的元数据元素副本。每次更新元数据元素，主机系统会使用这三个副本进行覆盖。如果任何两个元数据副本在稍后的读操作期间一致，则主机系统将这两个相同副本中的数据作为元数据元素的“真”值传送给应用。

如图 5 所示，主机系统使用包括三个元数据元素（MD）副本的新 MD 替换原始 MD 中每个数据位。原始元数据元素副本中的每个数据位需要三位空间来存储三个副本。如果这三位中某一个是错误的，通过该方法仍能够返回该位的“真”值。如果三个元数据元素副本采用按位的方式，即使每一位的三个副本中有一个副本出错，该方法仍可返回元数据元素的真值。假定按位方法中的冗余多于替换禁用自动 ECC 的实际要求，通过使用三副本表决方法对元数据值进行解码，可以最小化软件开销。

图 5. 使用三副本表决方法替换禁用的自动 ECC



使用按副本解码方法时，三副本表决方法能够处理该元数据元素的三个副本内的单比特错误。每个元数据元素的数据冗余开销相当大（300%），但是这些类型的元数据元素通常较小（字大小），因此对数据存储效率的影响也会很小。按副本解码算法的软件开销也很小，但是对读速度的影响需要考虑。赛普拉斯原型使用该方法，并确认该方法对数据存储效率和编程开销的影响非常小。

4 高可靠性的系统支持

表 1 提供了截至发布时点的数据存储解决方案列表，这些解决方案支持您的 100%ECC 分数或 100%有效 ECC 分数的目标。

表 1. 截至发布时的数据存储选项

供应商	文件系统	低于 100%的 ECC 分数法	100%有效 ECC 分数法	100% ECC 分数法
赛普拉斯 [1]	赛普拉斯 FFS — NOR/SPI/NAND	-	-	支持
赛普拉斯 [2, 3]	Linux MTD	支持	-	支持
赛普拉斯 [2]	Microsoft Flash PDD（WinCE 6 版本 2，WinCE 7）	-	-	支持
赛普拉斯 [2]	Microsoft FMD（WinCE 5、6 和 7）	支持	-	-
Blunk Microsystems[4]	LiteFS-NOR	-	-	支持
Blunk Microsystems[4]	TargetFAT 和 TargetFTL-NOR	-	-	支持
Datalight[4]	FlashFX 系列	支持	支持	-
Kyoto Software Research（KSR）[4]	Fugue	支持	-	支持
QNX[4]	QNX FFSv3（6.3 和 6.5）	支持	支持	-

注意：

1. *请访问 www.cypress.com，申请获取赛普拉斯 FFS 源代码数据包。*
2. *请访问 www.cypress.com，下载免费的源代码数据包。*
3. *赛普拉斯 MTD 修补还可以配置内核，使其正常工作。请参见附录 B：Linux MTD，了解详细信息。*
4. *更多信息，请联系文件系统供应商。*

5 结论

赛普拉斯 Flash 器件的 GL-S 和 GL-T 系列具有自动 ECC 功能，该功能对 Flash 操作的正常模式完全透明。几乎现有的全部消费和工业应用都可以迁移到 65 nm 或 45 nm Flash，而无需任何特殊软件考虑。本应用笔记还介绍了如何调整汽车或高可靠性应用，以实现 100%ECC 分数或 100%有效 ECC 分数。

6 相关文档

001-98285 — 3.0 V GL-S Flash 存储器系列的 S29GL01GS 1 Gbit、S29GL512S 512 Mbit、S29GL256S 256 Mbit、S29GL128S 128 Mbit 数据手册

001-98286 — 3.0 V GL-S Flash 存储器系列的 S29GL064S 64 Mbit 数据手册

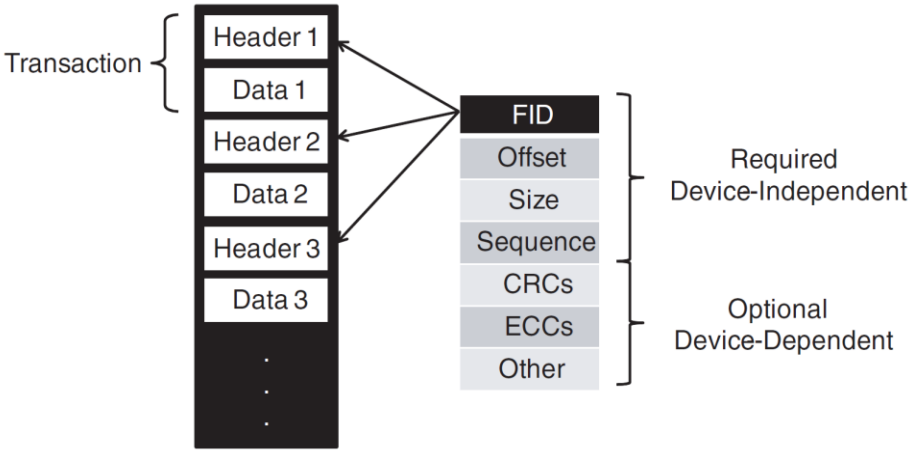
001-98296 — S70GL02GS 2 Gbit（256 Mbyte）3.0 V Flash 存储器数据手册

002-00247 — S29GL01GT 1 Gbit 和 S29GL512T 512 Mbit 并行 NOR Flash 数据手册

A 附录 A：QNX EFTS

ETFS 是基于数据处理事务的 Flash 文件系统，其中每个事务对一个或多个扇区组进行写操作。每个组包括一个头文件段和一个数据段，如图 6 所示。

图 6. EFTS 保存为数据处理事务并且包括多个头文件和数据有序对的扇区组



驱动程序在初始化期间可以设置扇区组的大小。大型页 NAND 的扇区组大小为 2 KB，而适合 GL-S 和 GL-T Flash 存储器的 ETFS 驱动程序的组大小为 1 KB。对于 NAND，数据处理事务的头文件被存储在备用区域内。为了符合 GL-S 和 GL-T Flash，事务头文件被存储在 Flash 扇区的末端，如图 7 所示。

- 这样，每个 128 KB Flash 扇区可以为不支持自动 ECC 的 GL-S 和 GL-T Flash 存储器保存 126 个扇区组。
- 对于支持自动 ECC 的 GL-S 和 GL-T Flash，头文件的页面对齐使每个 128 KB Flash 扇区可以保存的扇区组数量降低到 124 个。

这种调整可使驱动程序在上一代赛普拉斯 GL 系列 Flash 器件上实现最佳的数据存储效率，或者在 45 nm 和 65 nm GL Flash 上实现 100% ECC 分数，但其数据存储效率略微降低。

图 7. 适应 45 nm 或 65 nm 赛普拉斯 Flash 的 EFTS 中的扇区组分配

B 附录 B：Linux MTD

Linux MTD 修补程序（位于下面的方框部分，可从赛普拉斯网站获得）为具有 100% ECC 分数的数据存储解决方案奠定了基础。借助这个修补程序，JFFS2、UBIFS 和其它基于 MTD 的 Flash 文件系统在与 GL-S 和 GL-T Flash 器件配合使用时可满足高可靠性要求。

- 该修补程序修改了 AMD/赛普拉斯驱动程序，从而可以检查工艺技术的 CFI 字段，并为 65 nm 及后续版本的 Flash 器件使能“Sibley 模式”。
- 它还通过 512 字节缓冲区为 65 nm 及更高版本的 Flash 器件使能 FFS 缓冲特性。该特性确保所有写入操作大小都不小于一行。
- 由于原始代码使用字编程执行大小为四字节或更少的写操作，这个修补程序还将编程算法更改为适用于所有大小和所有 Flash 技术的 100%写缓冲区编程。

通过赛普拉斯测试可知道该修补程序与基于 65 nm 及更早技术的赛普拉斯 Flash 器件向后兼容。另外，修补程序支持基于 65 nm 和更新技术的器件实现 100% ECC 分数。

```
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;
+
```

对于 65 nm 和后续赛普拉斯 Flash 器件，您必须将‘写大小’更改为 512 字节，并且禁用‘bit writable’标志。这样可以确保主机通过单次编程操作对 512 字节大小的数据段进行写操作。可以使用其它大小为页倍数的缓冲区，但是赛普拉斯通过测试发现 512 字节的缓冲区可提供最佳吞吐量。

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

以下变更可确保写操作为‘总线对齐’。

```
@@ -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__ );
```

以下更改将删除条件字编程逻辑，并使用无条件写缓冲区编程替换它。

```
@@ -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;
}
```

C 附录 C：小型有效载荷 Hamming ECC

该附录提供的赛普拉斯代码可实现软件 ECC 管理。下面插入的代码表示双函数的组在 8 字节区域中操作，其中代码将 7 个字节分配给用户数据，并且为 ECC 奇偶校验位分配 1 个字节。在正常的文件系统操作中，该代码的执行比较快，并且不易引起注意，因为该算法仅管理八个字节。

```
// (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];
    }
}
```

文档修订记录

文档标题：AN200621 — 赛普拉斯 GL-S 和 GL-T Mirrorbit® Flash 非易失性存储器系列 — 自动 ECC

文档编号： 002-17948

版本	ECN	变更者	提交日期	变更说明
**	5552240	RZZH	01/09/2017	本文档版本号为 Rev. **，译自英文版 002-00621 Rev. *E。
*A	6161311	PRIT	04/30/2018	更新相关文件： 删除了规范 002-03239 的参考。 更新为新模板。 完成日落审查。

全球销售和设计支持

赛普拉斯公司拥有一个由办事处、解决方案中心、厂商代表和经销商组成的全球性网络。要想查找离您最近的办事处，请访问赛普拉斯所在地。

产品

ARM® Cortex® 微控制器	cypress.com/arm
汽车级产品	cypress.com/automotive
时钟与缓冲器	cypress.com/clocks
接口	cypress.com/interface
物联网	cypress.com/iot
存储器	cypress.com/memory
微控制器	cypress.com/mcu
PSoC	cypress.com/psoc
电源管理 IC	cypress.com/pmic
触摸感应	cypress.com/touch
USB 控制器	cypress.com/usb
无线连接	cypress.com/wireless

PSoC® 解决方案

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

赛普拉斯开发者社区

论坛 | WICED IoT 论坛| 项目 | 视频 | 博客 | 培训 | 组件

技术支持

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709