Technical Note

Patching the Linux Kernel and U-Boot for Micron® M29 Flash Memory

Introduction

This application note provides a guide for modifying the memory technology device (MTD) layer software for the purpose of correctly using Micron® M29 family Flash memory devices in a Linux environment.

This document is also useful for all Linux operating system users who are migrating from Spansion® GL™ parts to Micron M29 family Flash memory devices (M29W and M29EW). The document briefly outlines the primary specification differences between both families of devices. For a deeper analysis of hardware differences, please refer to the specific migration guide available on the Micron website at www.Micron.com. The section “Reference Documentation” on page 25 provides a URL for locating related migration guides.

This document also describes the modifications that are required to make a Linux environment work with M29 Flash memory devices.

Comparison of Spansion GL and Micron M29

The Micron M29 Flash memory devices are pin-compatible devices for the S29GL Flash memory device on leading 65nm lithography. Table 1 provides a comparison of the primary features of each device. For more detailed information on the compatibility of Micron and Spansion memory devices, please refer to the specific migration guide available the Micron website (for a list of URLs, see the section “Reference Documentation” on page 25).

Table 1: Spansion GL and M29 Feature Comparison

Features	M29EW	M29W	S29GL-P™	S29GL-N™
Process technology	65nm FG	65nm FG	90nm MirrorBit™	110nm MirrorBit™
Package	56-TSOP 64-Fortified BGA	56-TSOP 64-Fortified BGA	56-TSOP 64-Fortified BGA	56-TSOP 64-Fortified BGA
Block architecture	Uniformed 128KB	Uniformed 128KB	Uniformed 128KB	Uniformed 128KB
Page read size	16 words (x16) 32 bytes (x8)	8 words (x16) 16 bytes (x8)	8 words (x16) 16 bytes (x8)	8 words (x16) 16 bytes (x8)
Program buffer size	512 words (x16) 256 bytes (x8)	32 words (x16) 64 bytes (x8) 256 word (enhanced program)	32 words (x16) 64 bytes (x8)	16 words (x16) 32 bytes (x8)
Typical average program speed with full buffer	1.46 MB/s	0.7 MB/s	0.148 MB/s	0.148 MB/s
Support for common Flash interface	Yes	Yes	Yes	Yes

Table 1: Spansion GL and M29 Feature Comparison

Features	M29EW	M29W	S29GL-P™	S29GL-N™
Hardware protection of top and bottom sectors	Yes	Yes	Yes	Yes
Software protect and password protect	Yes	Yes	Yes	Yes
Password access	Yes	No	No	No

Enabling Buffered Programing Functionality in 2.4.x Kernels

Buffered programming features are not included in the MTD for 2.4.x Linux kernels. To take advantage of performance resulting from using the buffered programming feature, a new function must be implemented and included in the Linux Flash driver. The following code provides a possible implementation of buffered programing (validated for the 2.4.21 kernel version) to be inserted in the cfi_cmdset_0002.c, which is the low-level driver for Flash memory compliant with the 002 command set.

```
static inline int do_write_buffer(struct map_info *map, struct
flchip *chip,unsigned long adr, const u_char *buf, int len)
{
  unsigned long timeo = jiffies + HZ;
  unsigned int status;
  unsigned int dq7, dq5, dq1;
  struct cfi_private *cfi = map->fldrv_priv;
  DECLARE_WAITQUEUE(wait, current);
  int ret = 0;
  int z;
  __u32 datum = 0;
#ifdef CONFIG_TANGO2
  unsigned int newv, oldv;
  unsigned int mask = ((cfi_buswidth_is_2()) ? 0xffff : 0xff);
#endif

  if( (!cfi_buswidth_is_2() && !cfi_buswidth_is_4()) ||
      !len || (len % CFIDEV_BUSWIDTH) )
          return -EINVAL;

retry:
  cfi_spin_lock(chip->mutex);

  if (chip->state != FL_READY) {
      set_current_state(TASK_UNINTERRUPTIBLE);
      add_wait_queue(&chip->wq, &wait);

      cfi_spin_unlock(chip->mutex);

      schedule();
      remove_wait_queue(&chip->wq, &wait);
      timeo = jiffies + HZ;
```

```
        goto retry;
    }

    chip->state = FL_WRITING_TO_BUFFER;
#ifdef CONFIG_TANGO2
    if (adr == 0)
        oldv = get_unaligned((__u32*)buf);
    else
        oldv = *(volatile unsigned int *)map->map_priv_1;
#endif

    adr += chip->start;
    ENABLE_VPP(map);

    /* write buffers algorithm taken from Am29LV641MH/L manual */
    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_write(map, CMD(0x25), adr);
    cfi_write(map, CMD(len/CFIDEV_BUSWIDTH-1), adr); /* word count
*/

    /* Write data */
    for (z = 0; z < len; z += CFIDEV_BUSWIDTH) {
        if (cfi_buswidth_is_2()) {
            datum = *((__u16*)buf);
            buf += sizeof(__u16);
                cfi_write(map, datum, adr + z);
            //map->write16 (map, datum, adr+z);
        } else if (cfi_buswidth_is_4()) {
                datum = *((__u32*)buf);
            buf += sizeof(__u32);
                cfi_write(map, datum, adr + z);
                //map->write32 (map, datum, adr+z);
        }
    }
```

```
  /* start program */
  cfi_write(map, CMD(0x29), adr);

  cfi_spin_unlock(chip->mutex);
  cfi_udelay(chip->buffer_write_time);
  cfi_spin_lock(chip->mutex);

  /* use data polling algorithm */
  dq1 = CMD(1<<1);
  dq5 = CMD(1<<5);
  dq7 = CMD(1<<7);

  timeo = jiffies + ((((chip->buffer_write_time << cfi->cfiq-
>BufWriteTimeoutMax) * HZ) / 1000000) == 0 ?
          (HZ/10) /* setting timeout to 100ms */ :
          (((chip->buffer_write_time << cfi->cfiq->BufWriteTime-
outMax) * HZ) / 1000000) + 1);

  z -= CFIDEV_BUSWIDTH;/* go to last written address */
  do {
      status = cfi_read(map, adr+z);

#ifdef CONFIG_TANGO2
      newv = *(volatile unsigned int *)map->map_priv_1;
      if ((oldv & mask) == (newv & mask)) {
#endif
          if( (dq7 & status) == (dq7 & datum) )
              break;
          if( ((dq5 & status) == dq5) ||
              ((dq1 & status) == dq1) ) {
              status = cfi_read( map, adr+z );
              if( (dq7 & status) != (dq7 & datum) )
              {
                  ret = -EIO;
                  break;
              } else break;
          }
#ifdef CONFIG_TANGO2
          }
#endif


      if (need_resched()) {
              cfi_spin_unlock(chip->mutex);
              yield();
          cfi_spin_lock(chip->mutex);
      } else
              udelay(1);


  } while( !time_after(jiffies, timeo) );


  if( !ret && time_after( jiffies, timeo ) )
  {
          printk(KERN_WARNING "Waiting for write to complete
timed out in do_write_buffer.");


              ret = -EIO;
  }


  if( ret == -EIO ) {
      if( (dq1 & status) == dq1 ) {
          printk( "Flash write to Buffer aborted @ 0x%lx =
0x%x\n", adr, status );
          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);
      } else {
              printk( "Flash write to buffer failed @ 0x%lx =
0x%x\n", adr, status );
          cfi_write(map, CMD(0xF0), chip->start);
      }
  }


  DISABLE_VPP(map);
  chip->state = FL_READY;
  wake_up(&chip->wq);
  cfi_spin_unlock(chip->mutex);

  return ret;
}
```

To export the new write buffer feature to the MTD user, the following modifications must be applied to the `cfi_cmdset_0002.c` file:

```
@@ -33,6 +33,9 @@
#include <linux/mtd/cfi.h>

#define AMD_BOOTLOC_BUG
+#define DEBUG_CFI_FEATURES
+#define MAXCIR
+//#define FORCE_SINGLE_WRITE

#ifdef CONFIG_MTD_CFI_AMDSTD_RETRY

@@ -82,6 +85,7 @@ do { \

static int cfi_amdstd_read (struct mtd_info *, loff_t, size_t,
size_t *, u_char *);
static int cfi_amdstd_write(struct mtd_info *, loff_t, size_t,
size_t *, const u_char *);
+static int cfi_amdstd_write_buffers(struct mtd_info *, loff_t,
size_t, size_t *, const u_char *);
static int cfi_amdstd_erase_chip(struct mtd_info *, struct
erase_info *);
static int cfi_amdstd_erase_onesize(struct mtd_info *, struct
erase_info *);
static int cfi_amdstd_erase_varsize(struct mtd_info *, struct
erase_info *);
@@ -362,7 +367,18 @@ static struct mtd_info
*cfi_amdstd_setup(struct map_info *map)
          else
              mtd->erase = cfi_amdstd_erase_onesize;
          mtd->read = cfi_amdstd_read;
-             mtd->write = cfi_amdstd_write;
+#ifndef FORCE_SINGLE_WRITE
+             if( cfi->cfiq->BufWriteTimeoutTyp )
+               {
+                       printk( "Using buffer write method\n" );
+                       mtd->write = cfi_amdstd_write_buffers;
+               } else {
+#endif
+                       printk( "Using word write method\n" );
+                       mtd->write = cfi_amdstd_write;
+#ifndef FORCE_SINGLE_WRITE
+               }
+#endif
                        break;

  default:
@@ -968,6 +1129,81 @@ static int cfi_amdstd_write (struct
mtd_info *mtd, loff_t to , size_t len, size_
  return 0;
}

+static int cfi_amdstd_write_buffers (struct mtd_info *mtd,
loff_t to , size_t len, size_t *retlen, const u_char *buf)
+{
+         struct map_info *map = mtd->priv;
+         struct cfi_private *cfi = map->fldrv_priv;
+         int wbufsize = CFIDEV_INTERLEAVE << cfi->cfiq->MaxBuf-
WriteSize;
+         int ret = 0;
+         int chipnum;
+         unsigned long ofs;
+
+         /* code derived from
cfi_cmdset_0001.c:cfi_intelext_write_words */
+         *retlen = 0;
+         if (!len)
+         return 0;
+
+         chipnum = to >> cfi->chipshift;
+         ofs = to  - (chipnum << cfi->chipshift);
+
+         /* If it's not bus-aligned, do the first word write */
+         if (ofs & (CFIDEV_BUSWIDTH-1)) {
+             size_t local_len = (-ofs)&(CFIDEV_BUSWIDTH-1);
+             if (local_len > len)
+             local_len = len;
+         ret = cfi_amdstd_write(mtd, to, 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 > CFIDEV_BUSWIDTH) {
+             /* We must not cross write block boundaries */
+         int size = wbufsize - (ofs & (wbufsize-1));
+
+             if (size > len)
+                 size = len & ~(CFIDEV_BUSWIDTH-1);
+             ret = do_write_buffer(map, &cfi->chips[chipnum],
+                           ofs, buf, size);
+             if (ret)
+             return ret;
+
+             ofs += size;
+             buf += size;
+             (*retlen) += size;
+             len -= size;
+
+              if (ofs >> cfi->chipshift) {
+              chipnum ++;
+              ofs = 0;
+              if (chipnum == cfi->numchips)
+              return 0;
+              }
+              }
+
+              /* ... and write the remaining bytes */
+              if (len > 0) {
+              size_t local_retlen;
+              ret = cfi_amdstd_write(mtd, ofs + (chipnum << cfi-
>chipshift),
+              len, &local_retlen, buf);
+              if (ret)
+              return ret;
+              (*retlen) += local_retlen;
+              }
+
+              return 0;
+}

/*
* Handle devices with one erase region, that only implement
```

Enabling 1KB Buffered Programing for M29EW Devices

The M29EW device has a larger buffer size than the S29GL. As summarized in Table 1 on page 1, the size of the buffer is respectively 1KB for the M29EW and 32 bytes or 64 bytes for the S29. Typically, larger buffer sizes result in increased performance. The MTD driver automatically supports the 1KB buffer size (when the Flash memory is used in 16-bit mode) in kernel version 2.6.13 and later. Enabling the larger buffer size in older versions of the kernel requires a code modification, which is shown in the following example.

Apply this path in the `mtd/cfi.h` file:

```
- static inline map_word cfi_build_cmd(u_char cmd, struct
map_info *map, struct cfi_private *cfi)


+ static inline map_word cfi_build_cmd(u_long cmd, struct
map_info *map, struct cfi_private *cfi)
```

Enabling 1KB Buffered Programing for M29EW in U-Boot

As previously described, M29EW devices have a larger buffer size than S29GL devices. The size of the buffer is respectively 1KB for M29EW and 32 bytes or 64 bytes for the S29 (see Table 1 on page 1). Typically, larger buffer sizes result in increased performance.

Enabling the larger buffer size in U-Boot requires a code modification, which is shown in the following example.

Micron has developed several patches for the different versions of U-Boot that are available on demand for customers. However, the logic behind the modification is similar for each version, and it is possible to port one available patch for a specific version of U-Boot into a different version. The following code defines a patch developed for U-Boot 1.3.1.

```
diff -rupN a/drivers/mtd/cfi_flash.c b/drivers/mtd/cfi_flash.c
--- a/drivers/mtd/cfi_flash.c2007-12-06 10:21:19.000000000 +0100
+++ b/drivers/mtd/cfi_flash.c2011-03-21 14:41:49.000000000 +0100
@@ -184,8 +184,8 @@ flash_info_t flash_info[CFG_MAX_FLASH_BA
 typedef unsigned long flash_sect_t;

  static void flash_add_byte (flash_info_t * info, cfiword_t * cword, uchar c);
-static void flash_make_cmd (flash_info_t * info, uchar cmd, void *cmdbuf);
-static void flash_write_cmd (flash_info_t * info, flash_sect_t sect, uint off-
                set, uchar cmd);
+static void flash_make_cmd (flash_info_t * info, ulong cmd, void *cmdbuf);
+static void flash_write_cmd (flash_info_t * info, flash_sect_t sect, uint off-
                set, ulong cmd);
  static void flash_unlock_seq (flash_info_t * info, flash_sect_t sect);
  static int flash_isequal (flash_info_t * info, flash_sect_t sect, uint offset,
                uchar cmd);
  static int flash_isset (flash_info_t * info, flash_sect_t sect, uint offset,
                uchar cmd);
@@ -903,7 +903,7 @@ static void flash_add_byte (flash_info_t
 /*-----------------------------------------------------------------------
  * make a proper sized command based on the port and chip widths
  */
-static void flash_make_cmd (flash_info_t * info, uchar cmd, void *cmdbuf)
+/*static void flash_make_cmd (flash_info_t * info, uchar cmd, void *cmdbuf)
 {
      int i;
      uchar *cp = (uchar *) cmdbuf;
@@ -914,12 +914,33 @@ static void flash_make_cmd (flash_info_t
      for (i = 1; i <= info->portwidth; i++)
 #endif
      *cp++ = (i & (info->chipwidth - 1)) ? '\0' : cmd;
+}*/
```

```
+
+static void flash_make_cmd (flash_info_t * info, ulong cmd, void *cmdbuf)
+{
+     int i;
+     int cword_offset;
+     int cp_offset;
+     uchar val;
+     uchar *cp = (uchar *) cmdbuf;
+
+     for (i = info->portwidth; i > 0; i--){
+     cword_offset = (info->portwidth-i)%info->chipwidth;
+#if defined(__LITTLE_ENDIAN) || defined(CFG_WRITE_SWAPPED_DATA)
+         cp_offset = info->portwidth - i;
+         val = *((uchar*)&cmd + cword_offset);
+#else
+         cp_offset = i - 1;
+         val = *((uchar*)&cmd + sizeof(ulong) - cword_offset - 1);
+#endif
+         cp[cp_offset] = (cword_offset >= sizeof(ulong)) ? 0x00 : val;
+     }
  }

 /*
  * Write a proper sized command to the correct address
  */
-static void flash_write_cmd (flash_info_t * info, flash_sect_t sect, uint off-
                set, uchar cmd)
+static void flash_write_cmd (flash_info_t * info, flash_sect_t sect, uint off-
                set, ulong cmd)
 {

  volatile cfiptr_t addr;
@@ -1496,7 +1517,7 @@ static int flash_write_cfibuffer (flash_
          break;
      case FLASH_CFI_16BIT:
          cnt = len >> 1;
-         flash_write_cmd (info, sector, 0,  (uchar) cnt - 1);
+         flash_write_cmd (info, sector, 0,  cnt - 1);
          while (cnt-- > 0) *dst.wp++ = *src.wp++;
          break;
      case FLASH_CFI_32BIT:
```

Enabling Buffered Programming for M29EW in x8 Mode

M29EW Flash memory can work in two modes: x8 mode and x16 mode. The two modes refer to the Flash data bus size of the Flash, which is respectively 8 bits and 16 bits. The behavior of the two modes is similar, except for the buffer size, which for the x8 mode is set to 256 bytes instead of 1024 bytes. This causes a problem because in the CFI, the value related to the buffer size is set to 1024 bytes independently on the data bus size. This means that when M29EW is used in x8 mode, the Linux probe function reads the buffer size from the CFI and sets the internal structures to perform a program that fills 1024 bytes of buffer. The program fails as a result. A code modification is required to avoid this issue.

Micron has developed several patches for different kernels that are available on demand for customers. However, the logic behind the modification is similar for each kernel, and it is possible to port one available patch for a specific kernel version into a different kernel. The patch is also submitted to the Linux communities, and upon the approval it is made available for all future kernel deliveries. The following code defines a patch developed and validated for the 2.6.30 Linux kernel:

```
From: Massimo Cirillo <maxcir@gmail.com>

This patch fixes a problem related to an incorrect value con-
tained in the CFI

of M29EW devices family. The incorrect CFI field is MaxBufWrite-
Size that

should be 0x8 if the device is used in 8bit mode, whereas the
value read

out from CFI is 0xA.


Signed-off-by: Massimo Cirillo <maxcir@gmail.com>

---

diff --git a/drivers/mtd/chips/cfi_probe.c b/drivers/mtd/chips/
cfi_probe.c

old mode 100644

new mode 100755

index e63e674..5730201

--- a/drivers/mtd/chips/cfi_probe.c

+++ b/drivers/mtd/chips/cfi_probe.c

@@ -158,6 +158,9 @@ static int __xipram cfi_chip_setup(struct
map_info *map,

  __u32 base = 0;

  int num_erase_regions = cfi_read_query(map, base + (0x10 +
28)*ofs_factor);

  int i;

+ int extendedId1 = 0;

+ int extendedId2 = 0;

+ int extendedId3 = 0;
```

```
  xip_enable(base, map, cfi);
#ifdef DEBUG_CFI
@@ -195,6 +198,15 @@ static int __xipram cfi_chip_setup(struct
map_info *map,
  cfi->mfr = cfi_read_query16(map, base);
  cfi->id = cfi_read_query16(map, base + ofs_factor);

+ /* Get device ID cycle 1,2,3 for Micron/ST devices */
+ if ((cfi->mfr == CFI_MFR_NMX || cfi->mfr == CFI_MFR_ST)
+ && ((cfi->id & 0xff) == 0x7e)
+ && (le16_to_cpu(cfi->cfiq->P_ID) == 0x0002)) {
+ extendedId1 = cfi_read_query16(map, base + 0x1 * ofs_factor);
+ extendedId2 = cfi_read_query16(map, base + 0xe * ofs_factor);
+ extendedId3 = cfi_read_query16(map, base + 0xf * ofs_factor);
+ }
+
  /* Get AMD/Spansion extended JEDEC ID */
  if (cfi->mfr == CFI_MFR_AMD && (cfi->id & 0xff) == 0x7e)
  cfi->id = cfi_read_query(map, base + 0xe * ofs_factor) << 8 |
@@ -213,6 +225,16 @@ static int __xipram cfi_chip_setup(struct
map_info *map,
  cfi->cfiq->InterfaceDesc = le16_to_cpu(cfi->cfiq->Interface-
Desc);
  cfi->cfiq->MaxBufWriteSize = le16_to_cpu(cfi->cfiq->MaxBufWri-
teSize);

+   /* If the device is a M29EW used in 8-bit mode, adjust buffer
size */
+ if ((cfi->cfiq->MaxBufWriteSize > 0x8) && (cfi->mfr ==
CFI_MFR_NMX ||
+ cfi->mfr == CFI_MFR_ST) && (extendedId1 == 0x7E) &&
+ (extendedId2 == 0x22 || extendedId2 == 0x23 || extendedId2 ==
0x28) &&
+ (extendedId3 == 0x01)) {
+ cfi->cfiq->MaxBufWriteSize = 0x8;
+ pr_warning("Adjusted buffer size on Micron Flash M29EW fam-
ily");
+ pr_warning("in 8 bit mode\n");
+    }
```

```
+
#ifdef DEBUG_CFI
  /* Dump the information therein */
  print_cfi_ident(cfi->cfiq);
diff --git a/include/linux/mtd/cfi.h b/include/linux/mtd/cfi.h
old mode 100644
new mode 100755
index 88d3d8f..43d6a77
--- a/include/linux/mtd/cfi.h
+++ b/include/linux/mtd/cfi.h
@@ -522,6 +522,7 @@ struct cfi_fixup {
#define CFI_MFR_ATMEL 0x001F
#define CFI_MFR_SAMSUNG 0x00EC
#define CFI_MFR_ST  0x0020 /* STMicroelectronics */
+#define CFI_MFR_NMX 0x0089 /* Micron */

void cfi_fixup(struct mtd_info *mtd, struct cfi_fixup* fixups);
```

Note: No issues related to buffered programing have been experienced with M29W memory devices.

Enabling Buffered Programming for M29EW in x8 Mode in U-Boot

As previously described, the buffer size of M29EW Flash memory in x8 mode is 256 bytes instead of 1024 bytes. However, in the CFI the value related to the buffer size is set to 1024 bytes independently on the data bus size. As a result, if an M29EW device is used in x8 mode, the U-Boot probe function, which reads the buffer size from the CFI, will set internal structures to perform a program that fills 1024 bytes of buffer. The program fails as a result. A code modification is required to avoid this issue.

Micron has developed several patches for the different versions of U-Boot that are available on demand for customers. However, the logic behind the modification is similar for each version, and it is possible to port one available patch for a specific version of U-Boot into a different version. The following code defines a patch developed for U-Boot 1.3.1.

```
diff -rupN a/drivers/mtd/cfi_flash.c b/drivers/mtd/cfi_flash.c
--- a/drivers/mtd/cfi_flash.c2007-12-06 10:21:19.000000000 +0100
+++ b/drivers/mtd/cfi_flash.c2011-03-21 11:56:54.000000000 +0100
@@ -1320,6 +1320,15 @@ ulong flash_get_size (ulong base, int ba
      if ((info->interface == FLASH_CFI_X8X16) && (info->chipwidth ==
                FLASH_CFI_BY8)) {
          info->portwidth >>= 1;/* XXX - Need to test on x8/x16 in parallel. */
      }
+
+     /* M29EW256M: buffer size workaround in x8 mode */
+     if (info->chipwidth == FLASH_CFI_BY8
+      && info->manufacturer_id == 0x89
+      && info->device_id == 0x7E
+      && (info->device_id2 == 0x2201 || info->device_id2==0x2301 || info-
                >device_id2==0x2801)
+      && info->buffer_size > 256) {
+         info->buffer_size = 256;
+     }
  }

  flash_write_cmd (info, 0, 0, info->cmd_reset);
```

0xFF Command Intolerance for M29W128G

M29W128G devices do not recognize the 0xFF command (software reset command for 0001 command-set-compliant Flash) as a valid command. In some systems, when 0xFF is issued to the device, the M29W128G memory device will enter an unexpected state. As a result, it is necessary to add a 0xF0 command systematically after a 0xFF command.

To support this device, a software modification at the MTD level is required. In the Cfi_util.c file, the function cfi_qry_mode_off(), which resets the device after the autoselect mode, must have a 0xF0 command after the 0xFF command.

Note: This fix makes the fixup_M29W128G_write_buffer() no longer necessary (it has been included in the kernel release since version 2.6.30). Thus, it can be commented out.

The following code provides a patch for the 2.6.30 kernel. Similar modifications can be done to backport the modifications to earlier kernels.

```
From: Massimo Cirillo <maxcir@gmail.com>

The M29W128G Micron Flash devices are intolerant to any 0xFF com-
mand:

in the Cfi_util.c the function cfi_qry_mode_off() (that resets
the device

after the autoselect mode) must have a 0xF0 command after the
0xFF command.

This fix solves also the cause of the
fixup_M29W128G_write_buffer() fix,

that can be removed now.

The following patch applies to 2.6.30 kernel.


Signed-off-by: Massimo Cirillo <maxcir@gmail.com>

Acked-by: alexey Korolev <akorolev@infradead.org>

---

diff --git a/drivers/mtd/chips/cfi_cmdset_0002.c b/drivers/mtd/
chips/cfi_cmdset_0002.c

old mode 100644

new mode 100755

index 61ea833..94bb61e

--- a/drivers/mtd/chips/cfi_cmdset_0002.c

+++ b/drivers/mtd/chips/cfi_cmdset_0002.c

@@ -282,16 +282,6 @@ static void fixup_s29gl032n_sectors(struct
mtd_info *mtd,

  }

}


-static void fixup_M29W128G_write_buffer(struct mtd_info *mtd,
void *param)

-{

- struct map_info *map = mtd->priv;
- struct cfi_private *cfi = map->fldrv_priv;
- if (cfi->cfiq->BufWriteTimeoutTyp) {
- pr_warning("Don't use write buffer on ST Flash M29W128G\n");
- cfi->cfiq->BufWriteTimeoutTyp = 0;
- }
-}
-
static struct cfi_fixup cfi_fixup_table[] = {
  { CFI_MFR_ATMEL, CFI_ID_ANY, fixup_convert_atmel_pri, NULL },
#ifdef AMD_BOOTLOC_BUG
@@ -308,7 +298,6 @@ static struct cfi_fixup cfi_fixup_table[] =
{
  { CFI_MFR_AMD, 0x1301, fixup_s29gl064n_sectors, NULL, },
  { CFI_MFR_AMD, 0x1a00, fixup_s29gl032n_sectors, NULL, },
  { CFI_MFR_AMD, 0x1a01, fixup_s29gl032n_sectors, NULL, },
- { CFI_MFR_ST,  0x227E, fixup_M29W128G_write_buffer, NULL, },
#if !FORCE_WORD_WRITE
  { CFI_MFR_ANY, CFI_ID_ANY, fixup_use_write_buffers, NULL, },
#endif
diff --git a/drivers/mtd/chips/cfi_util.c b/drivers/mtd/chips/
cfi_util.c
old mode 100644
new mode 100755
index 34d40e2..8b87652
--- a/drivers/mtd/chips/cfi_util.c
+++ b/drivers/mtd/chips/cfi_util.c
@@ -81,6 +81,10 @@ void __xipram cfi_qry_mode_off(uint32_t base,
{
  cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type,
NULL);
  cfi_send_gen_cmd(0xFF, 0, base, map, cfi, cfi->device_type,
NULL);
+ /* M29W128G devices require an additional reset command
+ when exit qry mode */
+ if ((cfi->mfr == CFI_MFR_ST) && (cfi->id == 0x227E || cfi->id
== 0x7E))
+ cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type,
NULL);
}
EXPORT_SYMBOL_GPL(cfi_qry_mode_off);
--
```

Correcting Erase Suspend Hang Ups

Some revisions of the M29EW suffer from erase suspend hang ups. In particular, it can occur when the sequence *Erase Confirm* -> *Suspend* -> *Program* -> *Resumesequence* causes a lockup due to internal timing issues. The consequence is that the erase cannot be resumed without inserting a dummy command after programming and prior to resuming. If the erase suspend is not required, the user can enqueue a READ or PROGRAM operation that is required when the Flash device is erasing a block. This is done by applying the following code modification into the get_chip function in the cfi_cmdset_002.c file:

```
case FL_ERASING:

- if (mode == FL_WRITING)

+ if ((mode == FL_WRITING)|| (mode == FL_READY))

goto sleep;
```

This patch can be applied to all 2.6.x versions of the kernel. The erase suspend feature is not enabled when using a 2.4.x version of the kernel.

If the erase feature is required, the work-around is to issue a dummy write cycle that writes an F0 command code before the RESUME command.

The following code, which is applied to the cfi_cmdset_0002.c file, is a patch validated for kernel version 2.6.23.17:

```
diff --git a/drivers/mtd/chips/cfi_cmdset_0002.c b/drivers/mtd/
chips/cfi_cmdset_0002.c

old mode 100644

new mode 100755

index 1f64458..c06da03

--- a/drivers/mtd/chips/cfi_cmdset_0002.c

+++ b/drivers/mtd/chips/cfi_cmdset_0002.c

@@ -551,6 +551,11 @@ static int get_chip(struct map_info *map,
struct flchip *chip, unsigned long adr

               * there was an error (so leave the erase

               * routine to recover from it) or we trying to

               * use the erase-in-progress sector. */

+ /* before resume, insert a dummy 0xF0 cycle for Micron M29EW
devices */

+ if ( (cfi->mfr == 0x0089) &&

+ (((cfi->device_type == CFI_DEVICETYPE_X8) && ((cfi->id & 0xff)
== 0x7e))

+ || ((cfi->device_type == CFI_DEVICETYPE_X16) && (cfi->id ==
0x227e))) )

+ map_write(map, CMD(0xF0), chip->in_progress_block_addr);

              map_write(map, CMD(0x30), chip-
>in_progress_block_addr);

              chip->state = FL_ERASING;
              chip->oldstate = FL_READY;
@@ -600,6 +605,11 @@ static void put_chip(struct map_info *map, struct flchip *chip, unsigned long ad
  switch(chip->oldstate) {
  case FL_ERASING:
      chip->state = chip->oldstate;
+ /* before resume, insert a dummy 0xF0 cycle for Micron M29EW devices */
+ if ( (cfi->mfr == 0x0089) &&
+ (((cfi->device_type == CFI_DEVICETYPE_X8) && ((cfi->id & 0xff) == 0x7e))
+ || ((cfi->device_type == CFI_DEVICETYPE_X16) && (cfi->id == 0x227e))) )
+ map_write(map, CMD(0xF0), chip->in_progress_block_addr);
      map_write(map, CMD(0x30), chip->in_progress_block_addr);
      chip->oldstate = FL_READY;
      chip->state = FL_ERASING;
@@ -743,6 +753,11 @@ static void __xipram xip_udelay(struct map_info *map, struct flchip *chip,
          local_irq_disable();


          /* Resume the write or erase operation */
+ /* before resume, insert a dummy 0xF0 cycle for Micron M29EW devices */
+ if ( (cfi->mfr == 0x0089) &&
+ (((cfi->device_type == CFI_DEVICETYPE_X8) && ((cfi->id & 0xff) == 0x7e))
+ || ((cfi->device_type == CFI_DEVICETYPE_X16) && (cfi->id == 0x227e))) )
+ map_write(map, CMD(0xF0), adr);
          map_write(map, CMD(0x30), adr);
          chip->state = oldstate;
          start = xip_currtime();
```

Resolving the Delay After Resume Issue

Some revisions of the M29EW (for example, A1 and A2 step revisions) are affected by a problem that could cause a hang up when an ERASE SUSPEND command is issued after an ERASE RESUME operation without waiting for a minimum delay. The result is that once the ERASE seems to be completed (no bits are toggling), the contents of the Flash memory block on which the erase was ongoing could be inconsistent with the expected values (typically, the array value is stuck to the 0xC0, 0xC4, 0x80, or 0x84 values), causing a consequent failure of the ERASE operation.

The occurrence of this issue could be high, especially when file system operations on the Flash are intensive. As a result, it is recommended that a patch be applied. Intensive file system operations can cause many calls to the garbage routine to free Flash space (also by erasing physical Flash blocks) and as a result, many consecutive SUSPEND and RESUME commands can occur.

The problem disappears when a delay is inserted after the RESUME command by using the udelay (…) function available in Linux.

The DELAY value must be tuned based on the customer's platform. The maximum value that fixes the problem in all cases is 500µs. But, in our experience, a delay of 30µs to 50µs is sufficient in most cases.

When there is suspicion that the root cause of a customer's problem can be this issue, we recommend to:

1. Set the delay to the maximum value.
2. Check if the problem occurs again.
3. If the problem does not occur again, try a lower DELAY value.

Patch for M29EW Devices

The following patch for M29EW devices is compliant for the 2.6.28 kernel release:

```
--- a/drivers/mtd/chips/cfi_cmdset_0002.c

+++ b/drivers/mtd/chips/cfi_cmdset_0002.c

@@ -682,6 +682,7 @@ static void put_chip(struct map_info *map,
struct flchip *chip, unsigned long ad

  case FL_ERASING:

      chip->state = chip->oldstate;

      map_write(map, CMD(0x30), chip->in_progress_block_addr);

+ udelay(DELAY);

      chip->oldstate = FL_READY;

      chip->state = FL_ERASING;

      break;
```

0x554 Command Tolerance

Flash memory devices that are compliant with the 0002 command set can work in 8-bit and 16-bit modes. The Flash offsets for the first two write cycles of the autoselect command sequence are 0xAAA/0x555 for the 8-bit mode and 0x555/2AA for the 16-bit mode. Kernel versions 2.6.11 and older were used to write the autoselect commands at the 16-bit mode addresses for both modes (0x555/2AA addresses). The unique package for these types of Flash devices results in the addresses being shifted by 1 when the 8-bit mode is used (the data pin DQ15 becomes the A-1 address pin for the 8-bit mode). As a result, the internal address 0x555/2AA became 0xAAA/0x554.

Typically, Micron Flash memory devices do not accept 0x554 as a valid command address for the autoselect sequence, resulting in a Flash operation failure. To avoid this problem, a patch must be applied to the cfi.h file. In particular, the implementation of the cfi_build_cmd_addr function must be modified as specified in the following text:

```
static inline uint32_t cfi_build_cmd_addr(uint32_t cmd_ofs,
struct map_info *map, struct cfi_private *cfi)
{
          unsigned bankwidth = map_bankwidth(map);
          unsigned interleave = cfi_interleave(cfi);
          unsigned type = cfi->device_type;
          uint32_t addr;

          addr = (cmd_ofs * type) * interleave;

  /* Modify the unlock address if we are in compatiblity mode.
  * For 16bit devices on 8 bit busses
  * and 32bit devices on 16 bit busses
  * set the low bit of the alternating bit sequence of the
address.
  */
          if (((type * interleave) > bankwidth) &&
((uint8_t)cmd_ofs == 0xaa))
                  addr |= (type >> 1)*interleave;

          return  addr;
}
```

Summary of Available Patches

Table 2 provides a list of available patches related to the primary MTD incompatibilities.

Table 2: Summary of Available Patches

Issue	Affected Kernel	Patch Availability
Buffered programing enablement	2.4.x	2.4.21
1KB buffered programming	Version 2.6.13 and prior	2.6.12.6 2.4.30 2.4.25
x8 mode enabling	Version 2.6.30 and prior	2.6.27 2.6.14
0xFF tolerance	Version 2.6.30 and prior	2.6.8 2.6.14 2.6.22
Erase suspend hang up	2.4.x and 2.6.x	2.6.23
Delay after resume	2.4.x and 2.6.x	2.6.28 2.6.31

Reference Documentation

For additional information, refer to the migration guides for the Micron M29 Flash memory devices, which are available here: http://www.micron.com/products/nor-flash/parallel-nor-flash.

Conclusion

Micron recognizes the value of open source and the importance of supporting the open source community. Support for Micron Flash memory devices are enabled by contributing regular patches and updates to the Linux MTD and Linux file systems.

To request support for specific Linux issues, software or incompatibility with Micron Flash memory devices, contact your Micron representative or submit a request from www.micron.com.

Revision History

Rev. F **01/12**

- Updated hyperlinks

Rev. E **04/11**

- Added the “Enabling 1KB Buffered Programing for M29EW in U-Boot” section
- Added the “Enabling Buffered Programming for M29EW in x8 Mode in U-Boot” section
- Rebranded document as a technical note

Rev. D **10/10**

- Corrected title on inside pages.

Rev. C **07/10**

- Applied branding and formatting.

Rev. 2 . . . . **02/10**

- Added the Resolving the Delay After Resume Issue section.
- Updated the Summary of Available Patches section.

Rev. 1 . . . . **10/09**

Initial release authored by Giuseppe Russo and Massimo Cirillo.