MACRONIX INTERNATIONAL CO., LTD.

APPLICATION NOTE

NAND Design-In Notice

1. Introduction

NAND flash process migration is fast. On average, it only takes 1 ~ 2 years before the next process migration is ready to provide cost reductions for general embedded system applications, such as STB, TV, router, printer, etc. As NAND structures shrink in physical size, certain software efforts are required to maintain NAND flash data integrity and design reliability. In general, these software efforts include, but are not limited to, the use of ECC, Bad Black Management, and Partition Tables. This application note will focus on this topic and provide guidelines for NAND usage with software management.

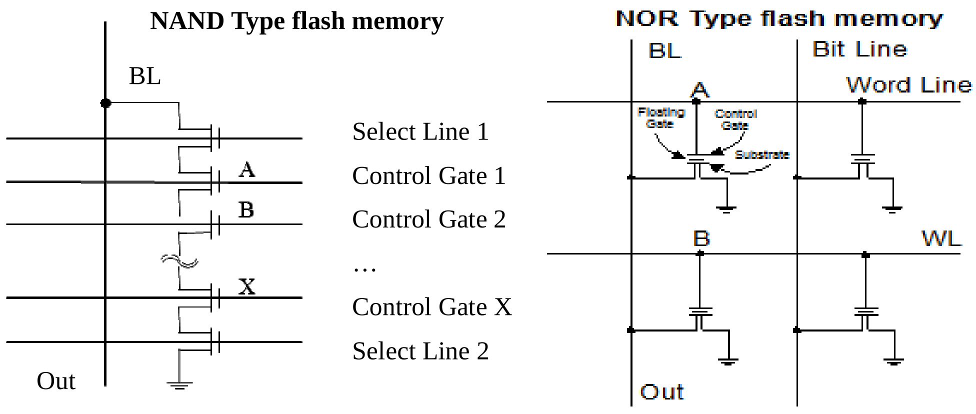
2. Bad Block Management

2-1. Bad Block

In NOR flash, memory cells are independent, but NAND memory cells are joined together to form strings of 32 or 64 cells. See Figure 2-1: Comparison of NAND and NOR Cell Structure. With this structure, a NAND memory cell may become dependent on other member cells of its string. Weak cells in a string may prevent data from being programmed into a cell, prevent cell erasure, cause cell read bit errors that exceed specifications, and so on. Every flash vendor has its own screening method to identify defective cells in a block and mark that block as BB (Bad Block) if the defective cells cannot be corrected with ECC. In general, NAND flash will not ship if the BB count exceeds 2% of the total number of blocks.

Bad blocks (BB) are common in NAND flash, and they must be handled correctly by software. When a user accesses a fresh NAND device for the first time, it is important to scan all bad block markers and build a BBT (Bad Block Table) in order to prevent using any of the bad blocks listed in the BBT. Additional bad blocks may be found during normal NAND usage, and they should be handled by BBM (Bad Block Management).

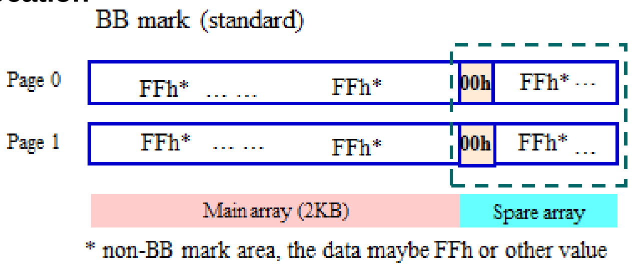
Figure 2-1: Comparison of NAND and NOR Cell Structure



2-2. Bad Block Marker

NAND Flash vendors always place bad block markers at fixed locations to identify bad blocks found prior to shipping. See Figure 2-2: BB Marker Location. The datasheet describes the bad block marker as follows: The BB mark is a non-FFh (or 00h) byte written to the 1st address of the 1st and 2nd page of the spare area of the BB. Note: The designer must take care not erase any bad block marks as they cannot be easily recoved if they are inadvertantly erased.

Figure 2-2: BB Marker Location



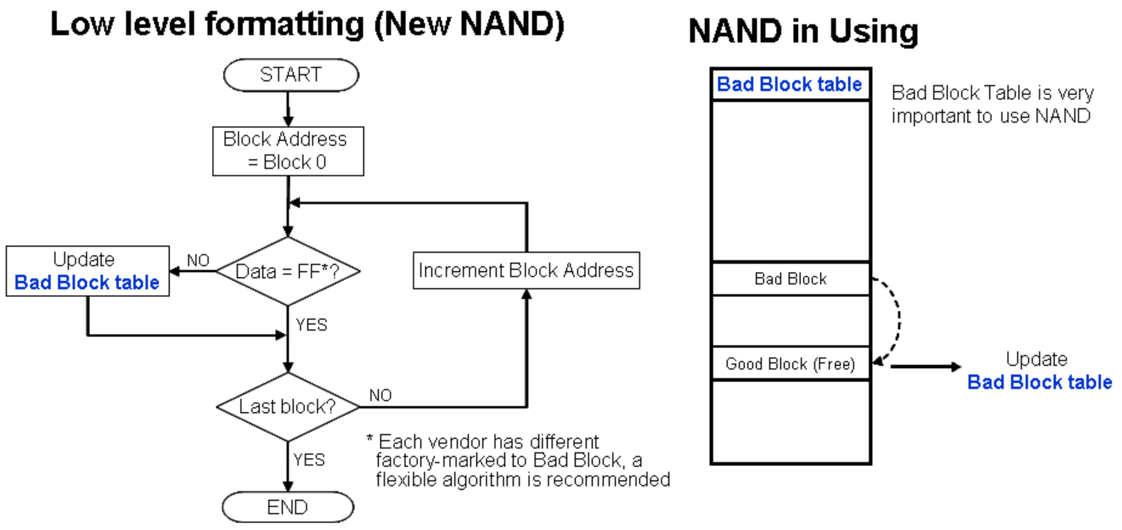
Note: 1. In factory rework flow, it will process erase flow before reprogram, so it is important to not erase BB marker.

2. Some system data structures are (512B+16B) from the beginning of page to the end. The new BB marker locates in the main array area, instead of the spare area. Because some programmers can't scan the BBT, rework flow may misidenitfy identify BB and treat BB as good block to store data.

2-3. Bad Block Management (BBM)

Software is necessary to achieve block management with NAND Flash designed into a system, and it also needs to provide the ECC function if the Host Processor does not support hardware ECC. Additionally, Bad Block Management(BBM) is a must-have to provide data storage integrity of NAND Flash and to maintain the Bad Block Table(BBT). See Figure 2-3: BBM – BB Scan and BBT is one of common methods being used by BBM. It supports NAND flash usage by the Host Processor in system. It also stores and builds up a backup table in initial good block. It not only keeps NAND initial bad block information, but it also records bad block replacement during NAND usage.

Figure 2-3: BBM – BB Scan and BBT

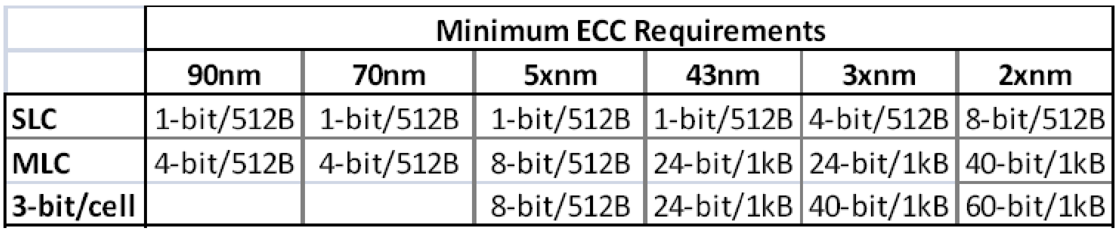


3. ECC bit Implement

In NAND flash, even good blocks might not be perfect and they may have a few weak bits. It requires ECC (Error Correction Code) to prevent weak bits from affecting data integrity. Each NAND flash vendor defines the ECC bit requirement in their datasheet (e.g., 4-bit ECC per 528 byte). In general, many Host Processors with embedded NAND Flash control units have hardware ECC that support multiple ECC algorithms. For example, the Hamming code algorithm handles 1-bit ECC and the BCH code algorithm handles over 4-bit ECC.

Furthermore, weak bits are related to the process technology, and the advanced process technologies with smaller geometries will statistically have more weak bits. Taking SLC NAND Flash as an example, one memory cell stores one bit and using the 4xnm process needs 1-bit/528Byte ECC capability per check (sector); the 3xnm process needs 4-bit/528Byte; and the 2xnm process node needs 8-bit/528Byte. Table 3-1 shows ECC bits and process technology relationship.

Table 3-1: ECC bits Requirement vs Process



	Minimum ECC Requirements					
	90nm	70nm	5xnm	43nm	3xnm	2xnm
SLC	1-bit/512B	1-bit/512B	1-bit/512B	1-bit/512B	4-bit/512B	8-bit/512B
MLC	4-bit/512B	4-bit/512B	8-bit/512B	24-bit/1kB	24-bit/1kB	40-bit/1kB
3-bit/cell			8-bit/512B	24-bit/1kB	40-bit/1kB	60-bit/1kB

4. Partition Table

A Partition Table should also be an integral part of the BBM. It defines different segments for code, data usage, or file system. It skips the initial bad block in each partition, and it also reserves some valid blocks beforehand for use during bad block replacement. Therefore, it is important to reserve enough valid blocks to avoid Partition Table crashes. Recommendations are as follows:

Table 4-1 shows a recommended Partition Table, and each partition reserves enough blocks for bad block replacement.

Table 4-1: Recommended Partition Table

Partition	Partition Range	Partition Size	Code/Data Size	Allow Bad Blocks
Partition 1	Block 0	1 block	1	0
Partition 2	Block 1 ~ Block 285	285 blocks	123	162
Partition 3	Block 286 ~ Block 585	300 blocks	202	98
Partition 4	Block 745 ~ Block 759	160 blocks	72	88

Table 4-2 shows an improper Partition Table with dangerous risks. There are not enough reserved valid blocks from Partition 2 to Partition 6. Once a partition has two initial bad blocks, this Partition Table will crash and cause system operation issues. Since BB distribution is completely random, two consecutive BBs are entirely possible. Therefore, reserving 3 or more good blocks per partition is highly recommended.

Table 4-2: Unsuitable Partition Table

Partition	Partition Range	Partition Size	Code/Data Size	Allow Bad Blocks
Partition 1	Block 0 ~ Block 7	8 blocks	6	2
Partition 2	Block 8 ~ Block 31	24 blocks	23	1
Partition 3	Block 32 ~ Block 54	23 blocks	22	1
Partition 4	Block 55 ~ Block 56	2 blocks	1	1
Partition 5	Block 57 ~ Block 78	22 blocks	21	1
Partition 6	Block 79 ~ Block 83	5 blocks	4	1
Partition 7	Block 84 ~ Block 386	303 blocks	283	20
Partition 8	Block 387 ~ Block 847	461 blocks	427	34

Note: BB quantity is max. 2% total blocks in each vendor, and the distribution isn't smooth or linear.

5. File System Usage

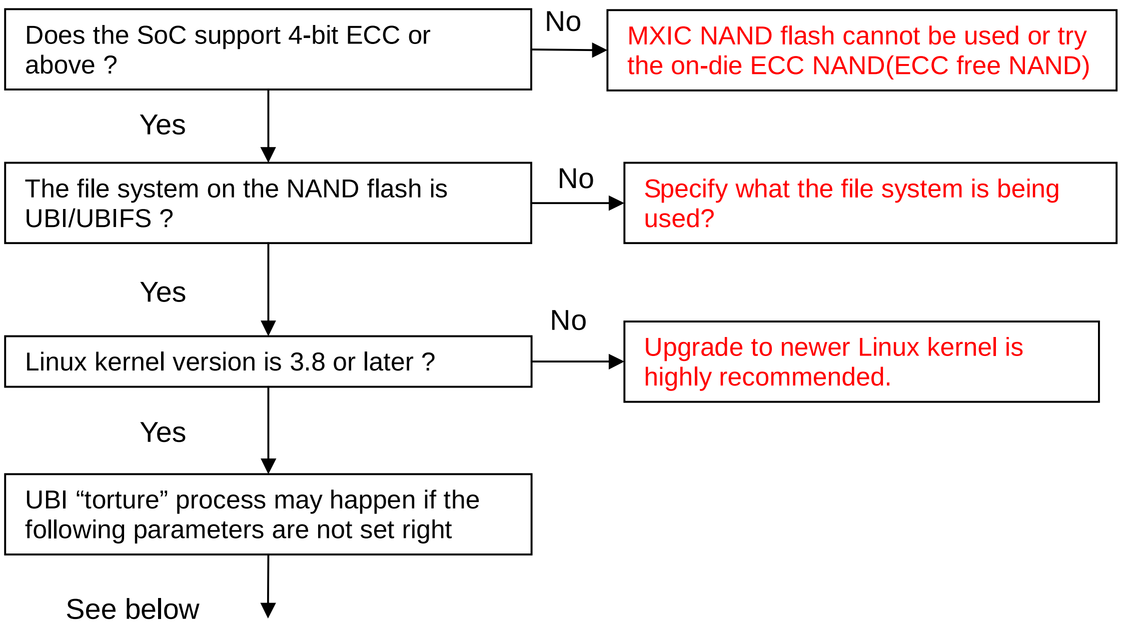
There are various file systems used by applications, like FAT32, UBIFS, YAFFS2, JFFS2, and so on. In NAND application, UBIFS and JFFS2 are two of most popular ones. File system structure may cause unexpected action in NAND usage. The sections below discuss how to maintain normal NAND usage under those two file systems.

UBI/UBIFS/JFFS2 is now the most popular file system for NAND flash devices in the Linux world. But failed cases do happen when some specific characteristics of NAND flash are ignored. Most are related to ECC and/or BB not being handled correctly. There are some additional points that must checked during the design phase, otherwise unexpected negative consequences may occur. For example, a torture test or empty scan can cause a rapid increase in bad blocks.

For those NOR-flash replacement customers or those updating legacy systems, which only support 1-bit ECC, we recommend using ECC free NAND flash instead of raw NAND flash.

5-1. UBI/UBIFS File System for Usage Example

The following is the checking diagram that the user must go through to make sure the target system can work smoothly with our NAND flash.


Does the SoC support 4-bit ECC or above ?
No
MXIC NAND flash cannot be used or try the on-die ECC NAND(ECC free NAND)
Yes
The file system on the NAND flash is UBI/UBIFS ?
No
Specify what the file system is being used?
Yes
Linux kernel version is 3.8 or later ?
No
Upgrade to newer Linux kernel is highly recommended.
Yes
UBI “torture” process may happen if the following parameters are not set right
See below

```
int ubi_io_read(…)          // in mtd/ubi/io.c
{
    ……
    err = mtd_read(…, buf);
    if (err) {
        if (mtd_is_bitflip(err)){
            …
            return UBI_IO_BITFLIPS; // this will start torture_peb()
        }
        …
    }
    return err;
}


int mtd_read(…)        // in mtd/mtdcore.c
{
    ……
    ret_code = mtd->_read(…);
    ……
    return ret_code >= mtd->bitflip_threshold ? –EUCLEAN : 0;
}
```

Note :
The “`bitflip_threshold`” value (defined in mtd.h as follows) must be set in the LLD (Low Level Driver) the same as ECC bits (ex. 4) required by the NAND flash in the driver probing stage.

```
// in  include/linux/mtd/mtd.h (after Linux version 3.8)
struct mtd_info {
     ……
   /*
   * read ops return -EUCLEAN if max number of bitflips corrected on any
   * one region comprising an ecc step equals or exceeds this value.
   * Settable by driver, else defaults to ecc_strength.  User can override
   * in sysfs.  N.B. The meaning of the -EUCLEAN return code has changed;
   * see Documentation/ABI/testing/sysfs-class-mtd for more detail.
   */
   unsigned int bitflip_threshold;
     ……
    /* max number of correctible bit errors per ecc step */
     unsigned int ecc_strength;
     ……
};
```

Note. Refer to the section 8. Appendix for more detail and information.

5-2. ECC-free NAND Usage Example in JFFS2

There is no ECC overhead for the SoC, but the data must be written sequentially and the NOP (Number Of Programming) can only be once for each ECC chunk (sector), though SLC NAND usually allows the NOP = 4 per page.

The file system may be guilty of violating the NOP specification.   For example, the JFFS2 file system violates this specification when it writes the “cleanmark” in the OOB area right after erasing the block to denote the erasing process is perfectly completed. When data is written into the page later, the NOP becomes twice and this violates the ECC-free NAND rule. This will corrupt the internal ECC and the written data will be damaged. So if the customer tries to use the JFFS2 file system along with ECC-free NAND, they should Google for the solution on how to skip the “cleanmark” for JFFS2 and adhere to the NOP=1 criteria. Because new ECC-free NAND flash all follow this rule, there are solutions already provided on the Internet. Notes: Refer to the Section 8. Appendix 8-4 JFFS2 for more detail and information.

6. Check List for NAND Usage

In order to avoid any unnecessary or extra design overhead, be sure to complete below questionnaire and respond back to our sales and/or technical people before starting your design.

☐ What is the Corechip being adopted in the project? (Optional)

☐ What is the Application? (Optional)

☐ How many the ECC bits can be handled by the Host Processor or Software? (Must)

☐ What is the Initial Bad Block requirement? in general is <2% of total blocks. (Must)

☐ It is highly suggested sharing or discussing with us the built-up Partition Table in project! (Desirable)

☐ Does the Bad Block Management be implemented in the project? (Must)

☐ What is the Linux Kernel version? Over ver. 3.8? (Must)

☐ What is the File System being used in the project? (Must)

7. Error Log output from UBI Torture & Empty Scan

If you find the following message in the Linux console, it means that it has detected some bit flips and the torture process is going to happen sooner or later.

```
UBI: fixable bit-flip detected at PEB xxxx
```

Similarly, if you find the following message, it means that some erased block contains non-0xFF data, which is also bit flip from 0xFF to other value (ex: 0xFE, 0x9F, …), This is what we call empty-scan problem that will be explained in Appendix 8-2.

```
UBIFS error (pid xxxx) : corrupt empty space LEB xxxx:xxxx, corruption starts at   xxxx
```

This will be helpful to quickly identify the problem in your system.

8. Summary

In NAND Flash usage, to guarantee the data storage integrity and reliable system design, the file system, ECC, BBM and Partition table considerations and implantations are the most important key before starting the design. Of course, responding to us with the completed questionnaire will be a good start to achieving the design target smoothly and simply.

8. Appendix

8-1. UBI/UBIFS Torture process in UBI Layer

Macronix SLC NAND flash is manufactured by 36nm fab technology to maximize the memory capacity, but at the cost of increasing the requirement of ECC from 1-bit to 4-bit(or 8-bit). Since most other vendors' SLC NAND flash only need 1-bit ECC, the Linux NAND driver of various SoC only implement 1-bit ECC when they find SLC NAND flash on board. Therefore, some modifications to the driver is required to increase the ECC to 4 bits when Macronix 4-bit ECC SLC NAND is found on board, or they can read the ONFI table in the flash to get the ECC requirement information. But if the SoC only supports 1-bit ECC, our flash is not suitable for this kind of platform.

UBI/UBIFS is based upon the MTD NAND driver layer. This file system really takes good care of the NAND flash by watching the health condition of each memory block every time when reading or writing the NAND flash. In older Linux architecture (before 3.8), the underlying NAND driver will report how many bits of data is corrected by ECC to the UBI layer when reading any data. And since the UBI layer also assumes all SLC NAND only needs 1-bt ECC, when the driver reports that there is any ECC correction applied, UBI worries that this block of memory is becoming unreliable and starts the "torturing" process by erasing and writing some pattern data to the whole block and read back the result to see if any ECC is required and repeat the process for 3 times. If the block failed (even when it is ECC correctable), the UBI will mark the block as BAD. But for those 4-bit ECC or even MLC flash, 1 or 2 bits of correctable errors doesn't mean the block is running out of its life cycle. So in later versions of Linux (after 3.8), the driver layer added ECC threshold parameters (bitflip_threshold and ecc_strength). Only when the correctable errors bit exceeds the threshold then report to the upper layer to take other actions if needed. So setting the ECC threshold parameters based on the real ECC requirement (and of course using the Linux kernel after version 3.8) is very critical for our NAND flash. Otherwise, the system will waste a lot of performance on running the "torture" process and more and more blocks are unnecessarily marked as bad, which will eventually cause the flash space to run out.

8-2. UBI/UBIFS Empty (erased) block checking in UBIFS

When a block is erased completely, all data is reset to 0xFF in every byte in the block. But in our flash, some bit flips are still possible, although the error bit number is still under the ECC correctable range. So if later the data is written along with ECC, it will still be ECC correctable. BUT, the UBIFS assumes that a sudden power loss could occur during the erase operation leaving random garbage data in the block. It automatically takes steps to verify that the erase operation completed successfully by doing an empty scan to see if the data of an erase block are all 0xFF. But when the driver is reading an erased block (no ECC was written and the ECC area are all 0xFF too) any bit flip is considered to be ECC uncorrectable (the ECC for all 0xFF data is not all 0xFF). And this will make the UBIFS go to some recovery process and sometimes it will crash the file system. Now it needs the driver to "intelligently" distinguish the data situation under no ECC condition. Usually, a driver knows that if all the data is 0xFF, this is an empty page, but now it has to allow some (under 4 bits total, for example) bytes that are not 0xFF and still consider it as an empty page and correct those non-0xFF data to 0xFF and return to upper layer.

The following is some portion of the UBIFS source code which the output error message could be seen if this empty scan fail occurs:

```
// in Linux/fs/ubifs/recovery.c
struct ubifs_scan_leb *ubifs_recover_leb(struct ubifs_info *c, int lnum,
                                         int offs, void *sbuf, int jhead)
{
```

```
            ......
        } else if (!is_empty(buf, len)) {
                if (!is_last_write(c, buf, offs)) {
                        int corruption = first_non_ff(buf, len);
                        /*
                         * See header comment for this file for more
                         * explanations about the reasons we have this check.
                         */
                        ubifs_err(c, "corrupt empty space LEB %d:%d, corruption starts at %d", lnum, offs, corruption);
                        /* Make sure we dump interesting non-0xFF data */
                        offs += corruption;
                        buf += corruption;
                        goto corrupted;
                }
        }
     .....
corrupted:
        ubifs_scanned_corruption(c, lnum, offs, buf);
        err = -EUCLEAN;
error:
        ubifs_err(c, "LEB %d scanning failed", lnum);
        ubifs_scan_destroy(sleb);
        return ERR_PTR(err);
}
```

The solution is already provided by Linux after verion 4.4. You can reference the follow source code to fix the problem.

in Linux/drivers/mtd/nand/nand_base.c (after ver 4.4)

```
int nand_check_erase_ecc_chunk(void *data, int datalen, void *ecc, int ecclen, void *extraoob, int extrooblen, int
bitflips_threshold)
{
     Int data_bitflips = 0, ecc_bitflips = 0, extraoob_birflips = 0;

     data_bitflips = nand_check_erased_buf(data, datalen, bitflips_threshold);
     if (data_bitflips < 0)
          return data_bitflips;
     bitflips_threshold    -= data_bitflips;

     ecc_bitflips = nand_check_erased_buf(ecc, ecclen, bitflips_threshold);
     if (ecc_bitflips < 0)
          return ecc_bitflips;
     bitflips_threshold    -= ecc_bitflips;

     extraoob_bitflips = nand_check_erased_buf(extraoob, extraoob len, bitflips_threshold);
     if (extraoob _bitflips < 0)
          return extraoob _bitflips;

     if (data_bitflips)
          memset(data, 0xff, datalen);

     if (ecc_bitflips)
          memset(ecc, 0xff, ecclen);

     if (extraoob_bitflips)
          memset(extraoob, 0xff, extraooblen);
```

```
        return data_bitflips + ecc_bitflips + extraoob_bitflips;
}
```

8-3. UBI/UBIFS Linux Source Code

```
/* in mtd/ubi/io.c */
int ubi_io_read(const struct ubi_device *ubi, void *buf, int pnum, int offset, int len)
{
	int err, retries = 0;
	size_t read;
	loff_t addr;

	dbg_io("read %d bytes from PEB %d:%d", len, pnum, offset);

	ubi_assert(pnum >= 0 && pnum < ubi->peb_count);
	ubi_assert(offset >= 0 && offset + len <= ubi->peb_size);
	ubi_assert(len > 0);

	err = self_check_not_bad(ubi, pnum);
	if (err)
		return err;

	/*
	 * Deliberately corrupt the buffer to improve robustness. Indeed, if we
	 * do not do this, the following may happen:
	 * 1. The buffer contains data from previous operation, e.g., read from
	 *    another PEB previously. The data looks like expected, e.g., if we
	 *    just do not read anything and return - the caller would not
	 *    notice this. E.g., if we are reading a VID header, the buffer may
	 *    contain a valid VID header from another PEB.
	 * 2. The driver is buggy and returns us success or -EBADMSG or
	 *    -EUCLEAN, but it does not actually put any data to the buffer.
	 *
	 * This may confuse UBI or upper layers - they may think the buffer
	 * contains valid data while in fact it is just old data. This is
	 * especially possible because UBI (and UBIFS) relies on CRC, and
	 * treats data as correct even in case of ECC errors if the CRC is
	 * correct.
	 *
	 * Try to prevent this situation by changing the first byte of the
	 * buffer.
	 */
	*((uint8_t *)buf) ^= 0xFF;

	addr = (loff_t)pnum * ubi->peb_size + offset;
retry:
	err = mtd_read(ubi->mtd, addr, len, &read, buf);
	if (err) {
		const char *errstr = mtd_is_eccerr(err) ? " (ECC error)" : "";

		if (mtd_is_bitflip(err)) {
			/*
			 * -EUCLEAN is reported if there was a bit-flip which
			 * was corrected, so this is harmless.
			 *
			 * We do not report about it here unless debugging is
			 * enabled. A corresponding message will be printed
```

```
                         * later, when it is has been scrubbed.
                         */
                        ubi_msg(ubi, "fixable bit-flip detected at PEB %d",
                                pnum);
                        ubi_assert(len == read);
                        return UBI_IO_BITFLIPS;
                }

                if (retries++ < UBI_IO_RETRIES) {
                        ubi_warn(ubi, "error %d%s while reading %d bytes from PEB %d:%d, read only %zd bytes, retry",
                                 err, errstr, len, pnum, offset, read);
                        yield();
                        goto retry;
                }

                ubi_err(ubi, "error %d%s while reading %d bytes from PEB %d:%d, read %zd bytes",
                        err, errstr, len, pnum, offset, read);
                dump_stack();

                /*
                 * The driver should never return -EBADMSG if it failed to read
                 * all the requested data. But some buggy drivers might do
                 * this, so we change it to -EIO.
                 */
                if (read != len && mtd_is_eccerr(err)) {
                        ubi_assert(0);
                        err = -EIO;
                }
        } else {
                ubi_assert(len == read);

                if (ubi_dbg_is_bitflip(ubi)) {
                        dbg_gen("bit-flip (emulated)");
                        err = UBI_IO_BITFLIPS;
                }
        }

        return err;
}
/* in mtd/mtdcore.c */
int mtd_read(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf)
{
        int ret_code;
        *retlen = 0;
        if (from < 0 || from >= mtd->size || len > mtd->size - from)
                return -EINVAL;
        if (!len)
                return 0;

        ledtrig_mtd_activity();
        /*
         * In the absence of an error, drivers return a non-negative integer
         * representing the maximum number of bitflips that were corrected on
         * any one ecc region (if applicable; zero otherwise).
         */
        ret_code = mtd->_read(mtd, from, len, retlen, buf);
        if (unlikely(ret_code < 0))
```

```
		return ret_code;
	if (mtd->ecc_strength == 0)
		return 0;		/* device lacks ecc */
	return ret_code >= mtd->bitflip_threshold ? -EUCLEAN : 0;
}

/* in mtd/ubi/io.c */
static uint8_t patterns[] = {0xa5, 0x5a, 0x0};
static int torture_peb(struct ubi_device *ubi, int pnum)
{
	int err, i, patt_count;

	ubi_msg(ubi, "run torture test for PEB %d", pnum);
	patt_count = ARRAY_SIZE(patterns);
	ubi_assert(patt_count > 0);

	mutex_lock(&ubi->buf_mutex);
	for (i = 0; i < patt_count; i++) {
		err = do_sync_erase(ubi, pnum);
		if (err)
			goto out;

		/* Make sure the PEB contains only 0xFF bytes */
		err = ubi_io_read(ubi, ubi->peb_buf, pnum, 0, ubi->peb_size);
		if (err)
			goto out;

		err = ubi_check_pattern(ubi->peb_buf, 0xFF, ubi->peb_size);
		if (err == 0) {
			ubi_err(ubi, "erased PEB %d, but a non-0xFF byte found",
				pnum);
			err = -EIO;
			goto out;
		}

		/* Write a pattern and check it */
		memset(ubi->peb_buf, patterns[i], ubi->peb_size);
		err = ubi_io_write(ubi, ubi->peb_buf, pnum, 0, ubi->peb_size);
		if (err)
			goto out;

		memset(ubi->peb_buf, ~patterns[i], ubi->peb_size);
		err = ubi_io_read(ubi, ubi->peb_buf, pnum, 0, ubi->peb_size);
		if (err)
			goto out;

		err = ubi_check_pattern(ubi->peb_buf, patterns[i],
						ubi->peb_size);
		if (err == 0) {
			ubi_err(ubi, "pattern %x checking failed for PEB %d",
				patterns[i], pnum);
			err = -EIO;
			goto out;
		}
	}

	err = patt_count;
```

```
        ubi_msg(ubi, "PEB %d passed torture test, do not mark it as bad", pnum);

out:
        mutex_unlock(&ubi->buf_mutex);
        if (err == UBI_IO_BITFLIPS || mtd_is_eccerr(err)) {
                /*
                 * If a bit-flip or data integrity error was detected, the test
                 * has not passed because it happened on a freshly erased
                 * physical eraseblock which means something is wrong with it.
                 */
                ubi_err(ubi, "read problems on freshly erased PEB %d, must be bad",
                        pnum);
                err = -EIO;
        }
        return err;
}
```

8-4. JFFS2

Here is the link related to section 6-2:

http://lists.infradead.org/pipermail/linux-mtd/2007-December/020047.html

Some portion of the patch content to JFFS2 as follows:

```
diff --git a/fs/jffs2/erase.c b/fs/jffs2/erase.c
index a1db918..65d7dd7 100644
--- a/fs/jffs2/erase.c
+++ b/fs/jffs2/erase.c
@@ -415,6 +415,7 @@ static void jffs2_mark_erased_block(struct jffs2_sb_info *c, struct jffs2_eraseb
        /* Cleanmarker in oob area or no cleanmarker at all ? */
        if (jffs2_cleanmarker_oob(c) || c->cleanmarker_size == 0) {

+               /* We only write cleanmarker in case of SLC NAND */
                if (jffs2_cleanmarker_oob(c)) {
                        if (jffs2_write_nand_cleanmarker(c, jeb))
                                goto filebad;
diff --git a/fs/jffs2/fs.c b/fs/jffs2/fs.c
index ee192af..cbeb822 100644
--- a/fs/jffs2/fs.c
+++ b/fs/jffs2/fs.c
@@ -656,7 +656,9 @@ void jffs2_gc_release_page(struct jffs2_sb_info *c,
 static int jffs2_flash_setup(struct jffs2_sb_info *c) {
        int ret = 0;
```

9. Revision History

Revision	Description	Date
1.0	Initial Release	Feb. 14, 2017
2.0	1. Add UBI/UBIFS error log to identify the torture and empty scan symptom 2. Modify the solution to the empty scan symptom from the new Linux NAND device driver source.	Sep. 25, 2017

NAND Design-In Notice