Technical Note

Enabling On-Die ECC NAND with JFFS2

Introduction

The Micron NAND Flash memory with on-die ECC is specifically designed to work with application processors that have either no ECC engine or only a 1-bit ECC engine. The on-die ECC engine is capable to correct up to four errors, effectively lengthening device life span.

JFFS2 is a mature, well maintained Flash file system, tightly coupled with the MTD NAND interface. The MTD NAND interface includes a common library and NAND-specific drivers to provide an abstraction layer that enables development of many NAND device types. Because of unique features in the on-die ECC NAND, both the JFFS2 and the NAND drivers require slight alterations. The alterations, described here, are a proven design using an identical board, the same Linux build, and a similar NAND device (2KB + 64-byte per page) but without the on-die ECC.

While the MTD is well defined and adapted, the MTD/NAND interface layer covers all the variations across the large number of micro-architectures, NAND devices, and application designs. The on-die ECC features essentially eliminate the write OOB feature which MTD uses to provide to FS. This technical note focuses on altering the JFFS2/MTD protocol slightly, also provide guidance on the MTD/NAND layer implementation.

Terminology

JFFS2: Flash file system

MTD: Interface module between the Flash file system and the Flash memory

OOB: Spare area on each Flash memory page

BBM: Bad block marker field on OOB

Write-Buffer: JFFS2 feature that delays writes to NAND page size before committing to NAND

Clean-Marker: JFFS2 feature that marks a block after erase.

Enabling and Disabling On-Die ECC

Table 1: On-Die ECC Settings

Command	Command Sequence	Settings	Description
SET FEATURES (EFh)	EFh - 90h - 08h[1] - 00h - 00h - 00h - WAIT	08h = Enabled	Enables internal on-die ECC.
	EFh - 90h - 00h[1] - 00h - 00h - 00h - WAIT	00h = Disabled	Disables internal on-die ECC
GET FEATURES (EEh)	EEh - 90h - WAIT - READ[2] - READ - READ - READ	08h/00h	Reads internal on-die ECC state, executing 6-bit detection and 4-bit error correction. Determines whether on-die ECC is enabled/ disabled.

Notes:
1. In the command sequence, the command code is followed by address 90h and then four data bytes, of which only the first is used to enable/disable the internal on-die ECC state.
2. When the READ operation is complete, read status bit 0 must be checked for occurrence of errors larger than four bits, after which the device must be returned to read mode by issuing the 00h command.

OOB Area Layout and Limitations

The ECC NAND Flash device internal ECC enables 5-bit detection and 4-bit error correction for 512 bytes (x8) or 256 words (x16) of the main area and 4 bytes of metadata I in the OOB area. The metadata II area, which consists of two bytes (x8), is not ECC protected. See following figure for the OOB area mapping.

Table 2: OOB Area Mapping

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Note:
1. Gray = ECC parity bytes; light blue = Metadata 2 unprotected bytes; dark blue = Metadata 1 protected bytes; White = Reserved bytes

While the ECC engine provides enhanced error correction and longer product life, the benefit comes with limitations: 1) Each ECC protected area (512+4 bytes) must be written as one partial-page program; 2) The separately writable OOB area (total 4x2 bytes) is small and not ECC protected; each 2KB page allows up to four error-protected WRITE operations of (512+4) bytes each. The four WRITE operations could be combined into one, two, or three to reduce NAND cell stress and to allow extra WRITE commands to the unprotected 8 bytes.

Because of these limitations, on-die ECC NAND with JFFS2 functions differently than NAND without the on-die ECC: four ECC units of (512 bytes data + 3 bytes ECC) and

four OOB units of (8 metadata + 3 bytes ECC) per page will not work. There is not enough memory for the 12 bytes required by the Clean Marker functionality, and the available 8 bytes, which are not error-protected, are unreliable.

JFFS2 Configuration

Developed originally for NOR flash, JFFS2 is capable of operating without OOB areas. However, JFFS2 key capabilities, Write-Buffer, Summary, and No-Clean-Marker, belong to opposite configuration modes: the first two belong to NAND mode while the last one belongs to NOR mode. Plus, NAND requires bad block management that does not exist in NOR.

Therefore, JFFS2 must be reconfigured to enable Write-Buffer and Summary functionality and disable Clean Marker functionality, while retaining its dependence on MTD bad block marking.

Following are additional reconfiguration parameters:

- Using Linux Ubuntu kernel for OMAP, git tag v3.9-rc8 (See details and source code at http://omappedia.org/wiki/OMAP_Ubuntu_Main).
- Reconfiguration can be applied on any JFFS2 including code changes between v3.0 and v3.9-rc8.
- MTD nandsim module enabled to simulate 2KB + 64 bytes per page NAND device, so the architecture/board-specific NAND driver implementation can be focused on JFFS2 configuration (For details on NAND driver implementation with the on-die ECC device, refer to *TN-29-56: Enabling On-Die ECC for OMAP3 on Linux/Android OS Introduction*).
- Code changes represented loosely in patch format. Lines beginning with "-" and "+" mean "change from" and "change to", respectively.

Kernel Configuration File

The following JFFS2 reconfiguration settings enable the Write Buffer and Summary capabilities.

+CONFIG_JFFS2_FS=y

+CONFIG_JFFS2_SUMMARY=y //enable summary

+CONFIG_JFFS2_FS_XATTR=y

+CONFIG_JFFS2_COMPRESSION_OPTIONS=y

+CONFIG_JFFS2_LZO=y

+CONFIG_JFFS2_FS_WRITEBUFFER=y //enable write buffer

+CONFIG_JFFS2_RUBIN=y

JFFS2: fs/jffs2/os-linux.h

In the Write Buffer enabled corresponding section, disable jffs2_cleanmarker_oob. Some places in code using this macro must be changed to use (c->mtd->type == MTD_NANDFLASH).

#ifndef CONFIG_JFFS2_FS_WRITEBUFFER

...

#else

...

- #define jffs2_cleanmarker_oob(c) (c->mtd->type == MTD_NANDFLASH)

+ #define jffs2_cleanmarker_oob(c) (0) //disable clean marker

...

JFFS2: fs/jffs2/erase.c

Use the original jffs2_cleanmarker_oob macro, which is(c->mtd->type == MTD_NAND-FLASH).

static void jffs2_erase_failed(struct jffs2_sb_info *c, struct jffs2_eraseblock

...

- if (jffs2_cleanmarker_oob(c) && (bad_offset != (uint32_t)MTD_FAIL_ADDR_UN-KNOWN)) {

+ if ((c->mtd->type == MTD_NANDFLASH) && (bad_offset != (uint32_t)MTD_FAIL_ADDR_UNKNOWN)) { //use mtd->type instead of jffs2_clean-marker_oob(c)

JFFS2: fs/jffs2/fs.c

Use the original jffs2_cleanmarker_oob macro, which is(c->mtd->type == MTD_NAND-FLASH).

static int jffs2_flash_setup(struct jffs2_sb_info *c) {

int ret = 0;

- if (jffs2_cleanmarker_oob(c)) {

+ if (c->mtd->type == MTD_NANDFLASH) { //use mtd->type instead of jffs2_clean-marker_oob(c)

/ * NAND flash... do setup accordingly * /

ret = jffs2_nand_flash_setup(c);

void jffs2_flash_cleanup(struct jffs2_sb_info *c) {

- if (jffs2_cleanmarker_oob(c)) {

+ if (c->mtd->type == MTD_NANDFLASH) { //use mtd->type instead of jffs2_clean-marker_oob(c)

…

JFFS2: fs/jffs2/scan.c

```
Handle the (c->mtd->type == MTD_NANDFLASH) case in a different way.
static int jffs2_scan_eraseblock(…) {
…
#if CONFIG_JFFS2_FS_WRITEBUFFER
if (jffs2_cleanmarker_oob(c)) {
…
}
+ else {
+ if (mtd_block_isbad(c->mtd, jef->offset))
+ Return BLK_STATE_BADBLOCK;
+ }
#endif
```

MTD: nand_base.c

```
No ECC needed for on-die ECC NAND.
chip->ecc.mode = NAND_ECC_NONE;
```

MTD and NAND Driver Configuration

Nandsim Module Changes

Code changes are not required for nandsim. The following arguments are passed to the kernel to simulate 2KB+64byte per page, error free NAND, with bad blocks 1, 23 and 45:

mtdoops.mtddev=0

nandsim.first_id_byte=0x2c nandsim.second_id_byte=0xA2

nandsim.third_id_byte=0x00 nandsim.fourth_id_byte=0x05

nandsim.bitflips=0 nandsim.badblocks=1,23,45

The most important requirement is to verify that the mtd_write_oob() function is never called. This is done by adding the following to the beginning of the function: while(1) printk("writing OOB!");

NAND Driver Implementation

The configuration should use a NAND driver that works with regular NAND flash. Applicable details can be found in *TN-29-56: Enabling On-Die ECC for OMAP3 on Linux/Android OS Introduction*.

The critical part on the NAND driver side is to implement and verify mtd_block_isbad() and mtd_block_markbad(). Based on chip->ecc.mode = NAND_ECC_NONE and nand_base.c code, the default calling path will be:

Jffs2_scan_eraseblock() à mtd_block_isbad() à nand_block_isbad() à nand_block_checkbad() à chip->block_bad() à nand_block_bad() à send commands to read BBM…

jffs2_write_nand_bad_block() à mtd_block_markbad() à nand_block_markbad() à nand_write_oob_std() à send commands to write BBM…

The above calling path uses standard functions in nand_base.c, not turning off ECC. Therefore chip->block_bad() and chip->block_markbad() need to be implemented, which involves enabling/disabling on-die ECC using the SET FEATURES command.

The chip->ecc.read_oob_raw() and chip->ecc.write_oob_raw() are not needed since all operations except the bad block management are with ECC on.

Implementation details of enabling/disabling on-die ECC can be found in *TN-29-56: Enabling On-Die ECC for OMAP3 on Linux/Android OS Introduction*.

Conclusion

By slightly altering JFFS2, it is possible to make it work with Micron on-die ECC NAND devices. The NAND driver for a hardware platform also needs to be modified based on existing working code.

Revision History

Rev. A – 8/13

- Initial release

8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-3900
www.micron.com/productsupport Customer Comment Line: 800-932-4992

This data sheet contains minimum and maximum limits specified over the power supply and temperature range set forth herein.
Although considered final, these specifications are subject to change, as further product development and data characterization sometimes occur.