SPI-NAND & Host-Side ECC in Linux – ECC engine Framework

This document firstly introduces the emergence of SPI-NAND devices, the motivation of offloading ECC to host side, and briefly describe the ECC engine framework for generic NAND devices in the Linux kernel v5.11~, and finally we demonstrate how to evaluate SPI-NAND with the generic ECC engine framework on the Xilinx ZYNQ platform.

Table of Contents

1. Introduction

The emergence of SPI-NAND devices, system designers can use the same hardware to switch from NOR to NAND instead of completely redesigning their systems for parallel NAND (i.e.: raw NAND). *Macronix* believes that the future direction of applications using SPI-NAND will eventually requires the host to manage ECC, not flash device, which will be the focus of this article.

Three factors are driving the design from NOR to NAND *[1][3]*, as well as the emerging SPI-NAND trend:

- Cost:

  Primarily due to architectural differences, NAND arrays are ~60% smaller than NOR arrays on the same process node. Compared with NOR, flash memory can also benefit from the ability to shrink to a lower geometric size. NAND can support higher density at a lower cost.

- Performance:

  Flash memory is typically used to store code that is downloaded to RAM for execution when the system is booted. This is called storing and downloading applications. Although the read performance of NAND is slower than NOR, it is sufficient for these store and download applications.

- Ease of use:

  Since SPI-NAND devices have the same pin layout as SPI-NOR, there is no need to change the PCB layout.

NAND flash device is cheap and inherently unstable, so it must have ECC (**E**rror-**C**orrecting **C**ode) function to ensure its data integrity. Later in this section, we also discuss the challenges of using SPI-NAND with an on-die ECC versus the flexibility of SPI-NAND with a host-based ECC.

2. Most Effective Way for ECC Handling

2.1 What is ECC?

ECC (**E**rror-**C**orrecting **C**ode) encodes the data so that the decoder can identify and correct errors in the data. Usually, the data is encoded by adding many redundant bits to the data. When the original data is reconstructed, the decoder checks the encoded message to check if there are any errors.

A NAND flash with ECC function has an error when accessing data, and the ECC will automatically detect and repair the error to keep the data in normal operation.

Each page of NAND corresponds to an area called the spare area. In Linux systems, it is generally referred to as the OOB (Out of band) area. This

area was originally based on the hardware characteristics of NAND flash: data reads and writes.

NAND is more prone to errors, so in order to ensure that the data is correct, there must be a corresponding detection and error correction mechanism. This mechanism is called ECC, so an additional area was designed to hold the checksum value of the data. read and write operations of the OOB are generally performed together with page operations, that is, when the page is read and written, the OOB is read and written accordingly.

The following is a brief description of the NAND flash ECC algorithms *[7]* that are commonly used and are also available in Linux:

1. Hamming
    - Very popular with stronger SLC NAND chips.
    - Efficiently corrects 1-bit error, and detects up to 2 bit errors per chunk.
    - Most of the raw NAND controllers have embedded a H/W Hamming ECC engine.
2. BCH
    - Very robust and flexible: suitable for almost any kind of (NAND) requirements.
    - Adaptable to any chunk size at almost any strength.
    - Very good ratio overhead/correction capability.
    - Limited to available out-of-band areas only.
    - The read operation is almost 10 times more complex than the write operation.

2.2 The Operation of Write/Read with ECC

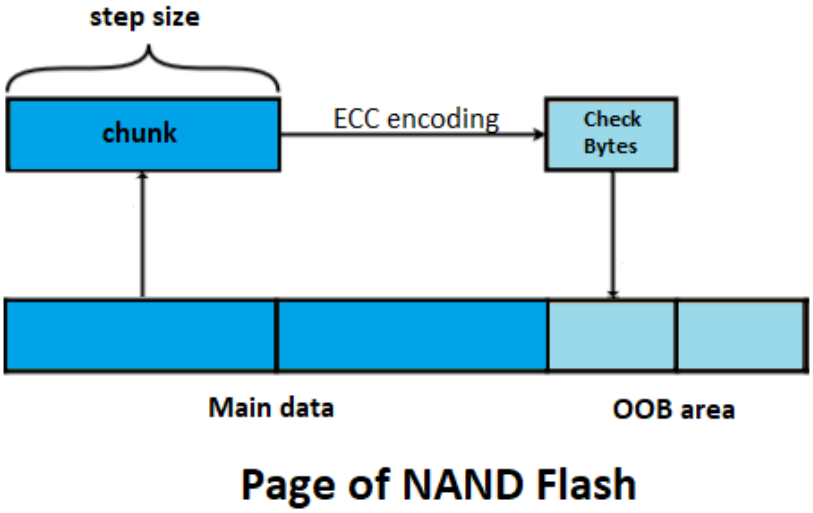
Let's look at the steps of Write/Read operation with ECC *[7]*:

Write operation:

1. The SPI host controller provides a chunk of data to ECC engine (there are multiple chunks contained in one page).
2. The ECC engine processes the chunk and generates ECC check bytes, and stores the check bytes in the OOB area.
3. Repeat this operation for all the data chunks contained in the page.
4. Write the entire page to the NAND.

Read operation:

1. Main data and ECC bytes are retrieved.
2. The ECC engine processes all the available data, chunk after chunk, to detect/correct the bit errors.
3. Return the original data and report ECC status



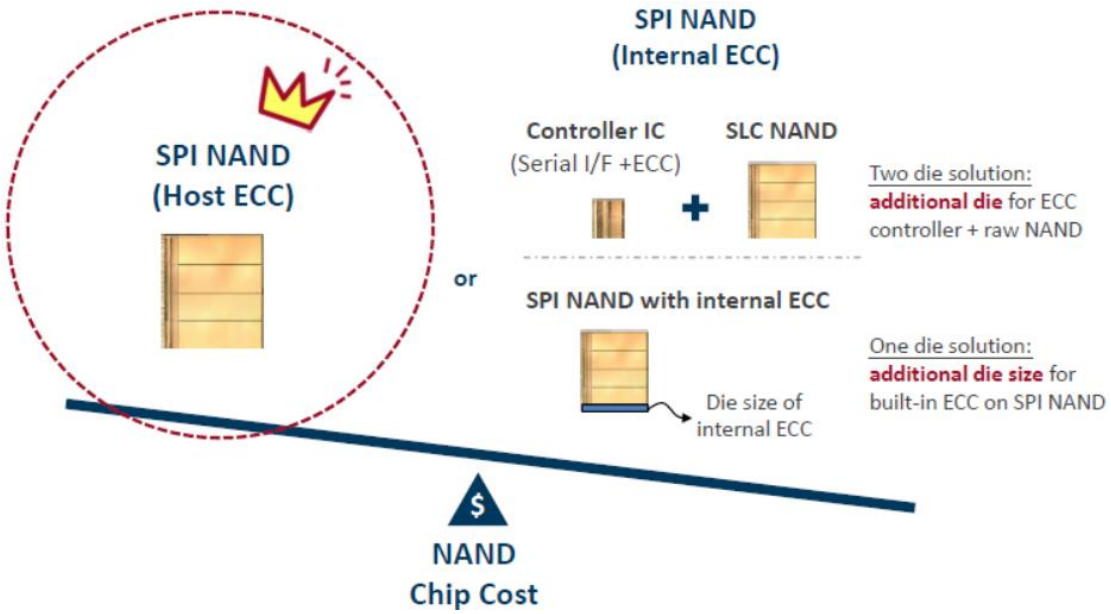
2.3 Why Offloading ECC to Host-side?

Let's look at the design of SPI-NAND. SPI-NAND devices are based on a standard raw NAND die with additional control logic to emulate the SPI and handle ECC.

This is typically achieved in one of two ways *[1]*:

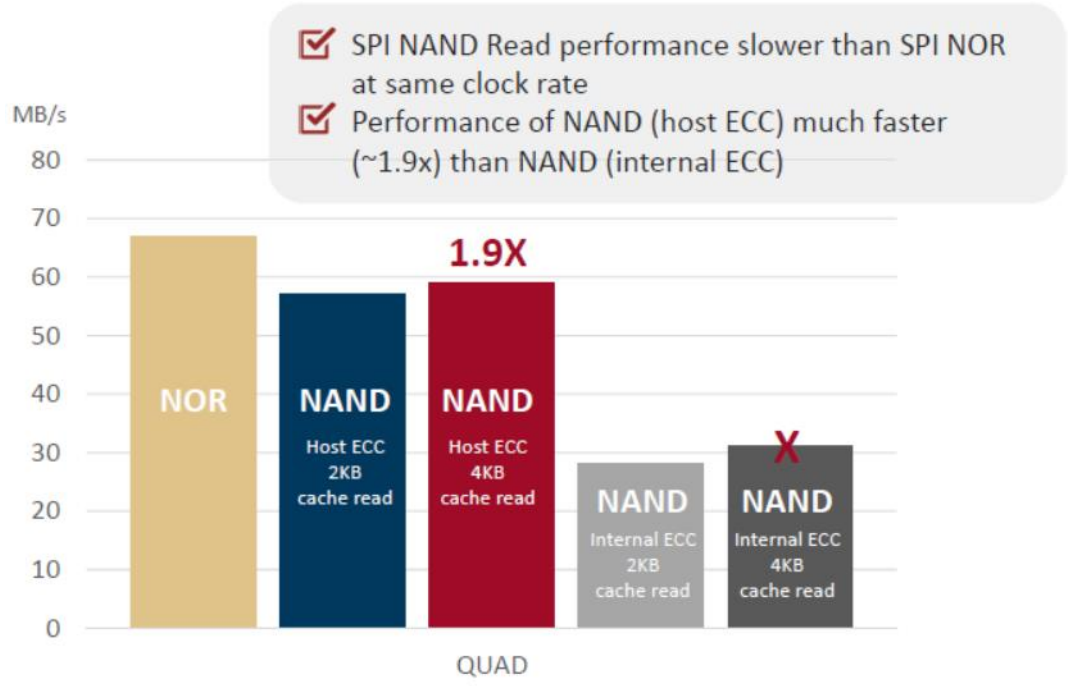
- By adding the control logic under NAND device itself, that is a monolithic die, which will increase the die size and the cost.
- By stacking a controller chip on the NAND die, making it a multi-chip solution. The multi-chip package has a cost adder, so it is generally more expensive than a monolithic solution.

However, in order to get the most cost-effective SPI-NAND solution, there is a third option: We can offload the H/W ECC engine from the NAND devices, just like a raw NAND, and shift it to the host-side(Includes H/W ECC engine embedded in host controller and SoC/CPU with S/W ECC engine). A typical 8-bit ECC engine can be implemented with roughly 50K additional gates, and for an entry-level host (MCU or SoC) this is a trivial amount of gates. By adding the ECC engine on the host, the host cost may go up slightly, but it will be much lower than the die size impact to a NAND device with on-die ECC.



Many designs are based on a store and download architecture (shadowing code form NAND flash to RAM) *[2]*. This means that the system cannot execute code directly from NAND flash (XIP), so it reads code from NAND to DRAM and executes it from DRAM.

Assuming all devices are running at similar clock speeds, NOR flash has the fastest read throughput at about 66 MB/sec. The two bars on the right illustrate the read performance of NAND with on-die ECC, where the NAND device calculates the ECC before reading the data. this NAND architecture is the slowest and can reach about 27 MB/sec, which is 60% lower than NOR. However, if the ECC calculation is performed by the host, this read speed can be increased to 56MB/sec, which is very similar to the read performance of NOR flash.

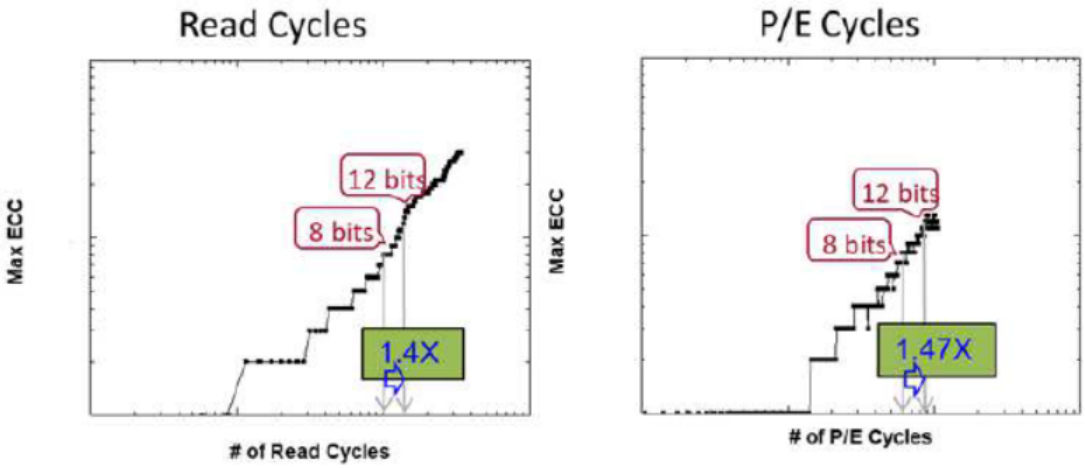


Bit flipping may occur during NAND read or programming operations. An additional benefit of host-based ECC is the ability to use stronger ECC and to extend the life of the NAND device. When using NAND with on-die ECC, the device operates with a fixed level of ECC, for example, 8-bit error correction. However, if error correction is performed by the host, the host may have the ability to support multiple levels of ECC.

By using stronger ECC, (e.g., 12-bit vs. 8-bit), the NAND device will be able to tolerate more bit scrambling before the data needs to be flushed or rewritten. This means that NAND will perform fewer program erase cycles and ultimately experience a longer lifetime.

Host SoC can implement a strong ECC to increase the life of a NAND device, e.g. 12-bit ECC will extend the NAND reliability lifetime:

- **1.4X** Read Cycles Lifetime compared to an 8-bit ECC in SPI NAND
- **1.47X** P/E Cycles Lifetime compared to an 8-bit ECC in SPI NAND



By offloading the ECC function to the host side, the MCU will absorb some additional cost. A typical MCU may have about 3 million gates. A BCH 8-bit ECC engine requires about 50,000 gates to implement, which is about a 1.7% increase in gate count *[1][3]*. If we add ECC engines to NAND, this impact on the chip size of the MCU will be much smaller than the impact on the NAND IC. As mentioned earlier, there are also two types of raw NAND, with and without on-die ECC. however, host-based ECC is the primary error correction method for handling raw NAND.

Therefore, in the short term, we see a need for NAND with on-die ECC to support the migration from SPI-NOR to SPI-NAND, but as applications evolve and the serial NAND market matures, we see a growing need for lower cost solutions with better performance and longer life. The best way to support these requirements is through a host with a built-in ECC engine.

In the raw NAND market, most vendors support the ONFI standard. For SPI-NAND, there is no such standard, which can lead to differences in specifications between suppliers.

For example, each vendor may require 8-bit ECC error correction, but alternate regions may have different sections or regions protected by ECC. As a result, customers need to design their firmware to accommodate these differences between multiple vendors. This is an unnecessary overhead that limits the flexibility of choosing different NAND vendors. However, if the ECC

function is performed from the host side, the system does not have to consider these differences for each vendor. The host can support any vendor's SPI-NAND device, as long as it provides the required minimum error correction. With host-based ECC, this problem is eliminated.

3. SPI-NAND & ECC Engine Framework in Linux

3.1 Introduce the SPI-NAND & ECC Framework

The focus of this section is on the relationship between the generic ECC framework and the SPI-NAND subsystem, and therefore the origin of the SPI-NAND subsystem will be mentioned first.

Supporting for the SPI-NAND devices in Linux has been available since v4.19, but only the on-die ECC engine was supported, others were not. Raw NAND controllers usually integrated an ECC engine and controller's driver embedded some code to enable/disable the ECC function. However, SPI-NAND devices may not have an on-die ECC engine and must use an external ECC engine in host side.

Although we have been seeing new high density NAND devices without on-die ECC coming out. It needs more powerful computing power to carry out longer bit error correction, it is best to offload to a dedicated hardware, but the SPI-NAND subsystem is not yet ready for this.

To solve these situations, *Macronix* commissioned Miquèl Raynal (Embedded Linux Engineer at Bootlin, also Linux MTD subsystem maintainer) to proposal a series patch (Introduce the generic ECC engine abstraction) to make things work, and the patch set has been accepted/merged in Linux v5.11.

The generic ECC framework for NAND is almost completed, the remaining is to proposal of a driver for *Macronix*'s external hardware ECC engine.

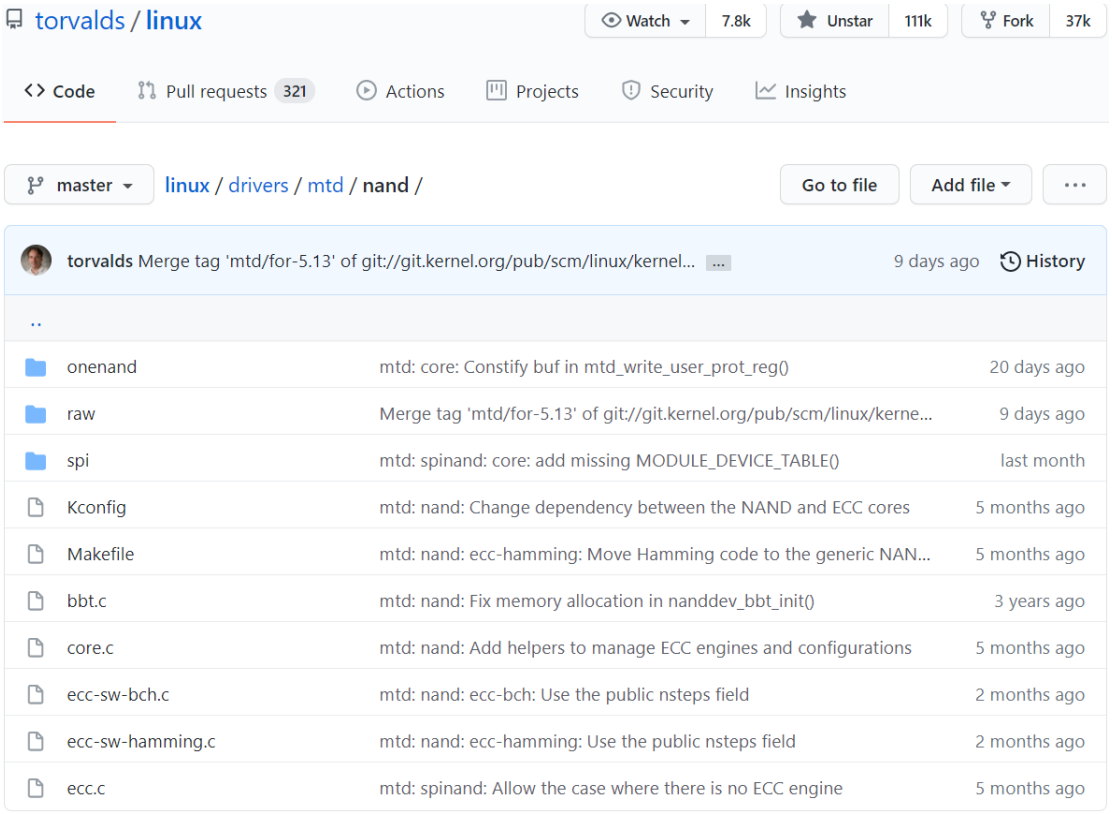
The design logic in the generic NAND ECC engine framework *[5]* (i.e.: differences from before v5.11) is:

1. Use the common NAND core for all NAND devices (raw and SPI)
2. Create the ECC engine interface in drivers/mtd/nand/ecc.c
3. Make the two software engines (Hamming and BCH) generic by

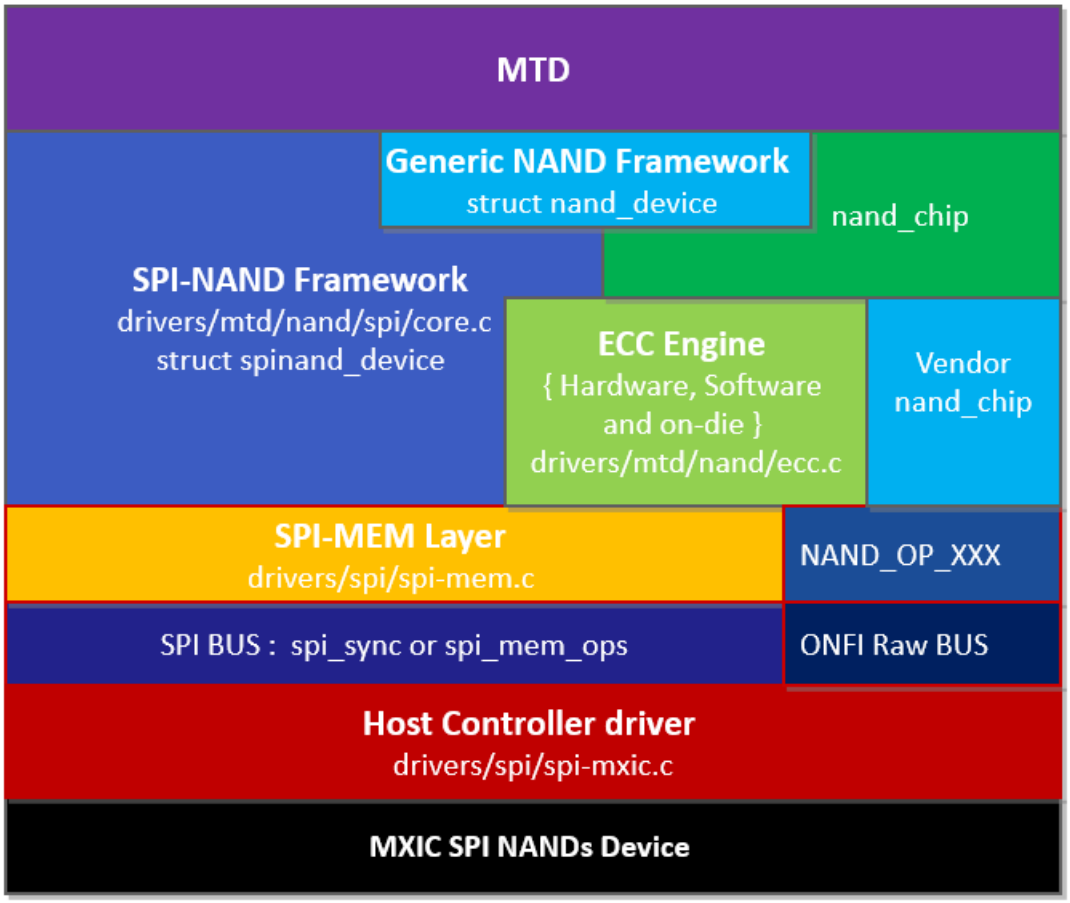
moving them to drivers/mtd/nand/ecc-sw-*.c, and write the raw NAND helper to use these two new engines
4. Using all the above ECC engines from the SPI-NAND layer (user can now use soft BCH if no ECC engine is available).
5. Isolating the SPI-NAND on-die ECC engine in its own driver
6. Migrating the raw NAND core to make a proper use of these S/W ECC engines
7. Deprecating in the raw NAND subsystem the interfaces used until now

3.2 Source Code Structure

This figure is the SPI-NAND & ECC framework code structure in Linux kernel source folder (drivers/mtd/nand) *[8]*, spi/ folder is containing SPI-NAND drivers, raw/ folder is containing raw NAND (ONFI NAND). ECC framework related files are ecc.c (generic NAND ECC engine abstraction layer, for SPI and raw), ecc-sw-bch.c and ecc-sw-hamming.c (both are S/W ECC engine implements). If there is a driver for the SPI host ECC engine, it should be placed here, too.

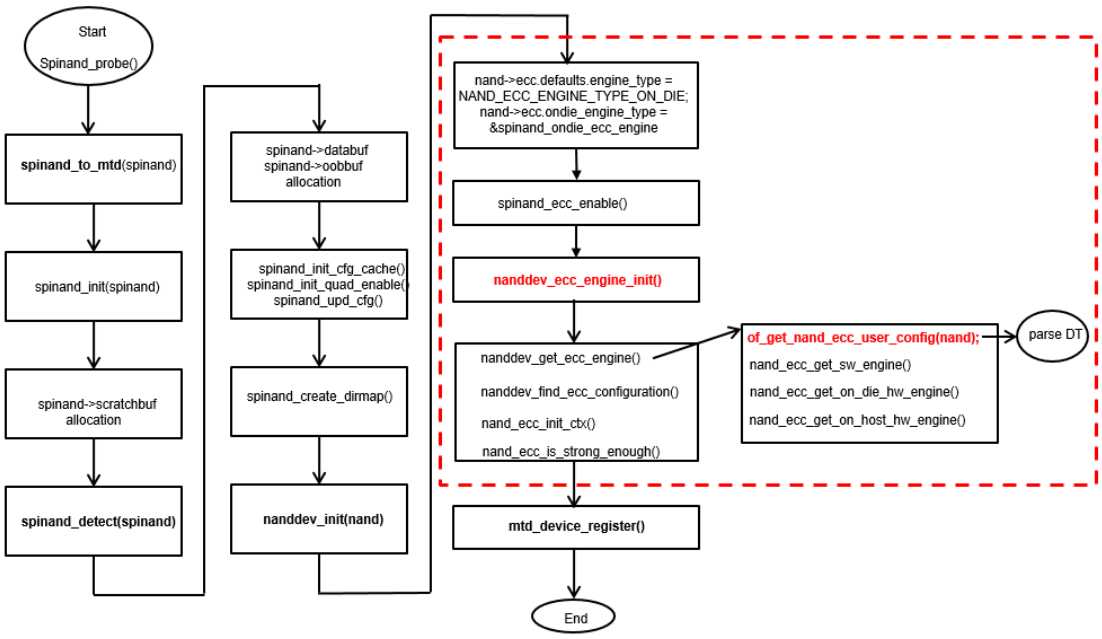


In summary, adding the generic ECC framework to the NAND subsystem is a generalization of the software ECC algorithm and related data structures that were originally only available for the raw NAND controller driver. This allows both SPI-NAND and raw NAND to share the NAND core, ECC related initialization procedures, parameters of device tree and S/W ECC algorithm implements.



3.3 Initialization flow of SPI-NAND + ECC framework

The process of probe, after adding the generic ECC framework, is roughly the same as the general SPI-NAND one. Then it will initialize the ECC engine related data structure and read the ECC related configuration settings in the device tree (e.g. engine type, S/W ECC algorithm type, ECC step size, ECC strength...), if the user does not specify the ECC engine type in the device tree, it will be defaulted to on-die type.

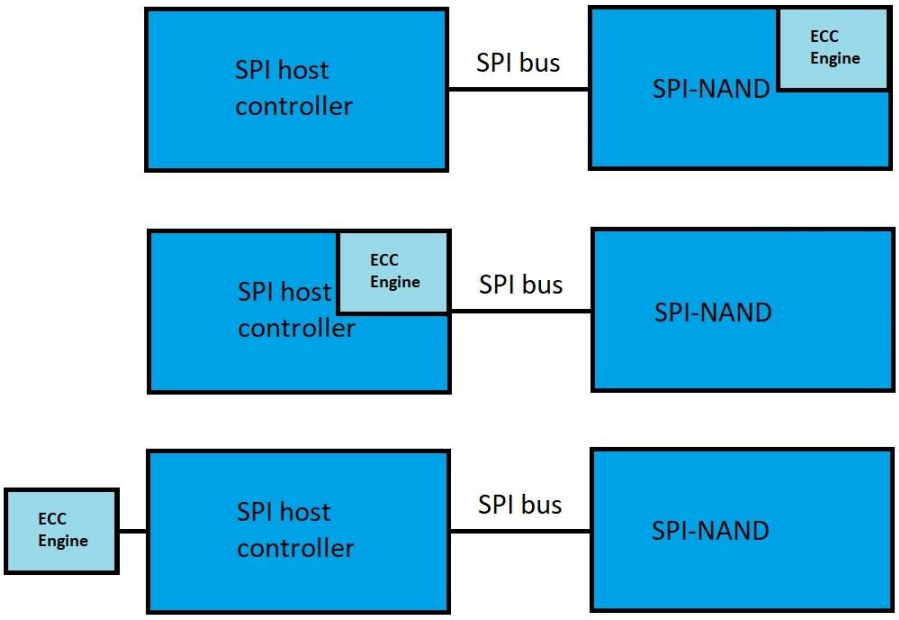


The following is driver/mtd/nand/core.c: nanddev_get_ecc_engine(), it will find and get a suitable ECC engine according device tree's related configurations. It is also the entry point for various ECC engines, such as S/W ECC, ON-DIE ECC or H/W ECC. But, currently there is no support for any of them and *Macronix* is continuing to develop and verify our host-based H/W ECC engine for SPI-NAND, and the driver will be upstream in Linux soon.

```
static int nanddev_get_ecc_engine(struct nand_device *nand)
{
        int engine_type;

        /* Read the user desires in terms of ECC engine/configuration */
        of_get_nand_ecc_user_config(nand);

        engine_type = nand->ecc.user_conf.engine_type;
        if (engine_type == NAND_ECC_ENGINE_TYPE_INVALID)
                engine_type = nand->ecc.defaults.engine_type;

        switch (engine_type) {
        case NAND_ECC_ENGINE_TYPE_NONE:
                return 0;
        case NAND_ECC_ENGINE_TYPE_SOFT:
                nand->ecc.engine = nand_ecc_get_sw_engine(nand);
                break;
        case NAND_ECC_ENGINE_TYPE_ON_DIE:
                nand->ecc.engine = nand_ecc_get_on_die_hw_engine(nand);
                break;
        case NAND_ECC_ENGINE_TYPE_ON_HOST:
                pr_err("On-host hardware ECC engines not supported yet\n");
                break;
        default:
                pr_err("Missing ECC engine type\n");
        }
```

There are three configurations of ECC engines for SPI-NAND *[5][7]*, which are supported in the ECC framework (*Macronix* also provides those solutions):

1. ECC engine is on-die.
    - The ECC engine is inside the NAND's data access pipeline, on the chip's side.
2. ECC engine is part of the SPI host controller.
    - The ECC engine is inside the NAND's data access pipeline, on the host controller's side.
3. ECC engine may be external.
    - The ECC engine is outside the NAND's data access pipeline, it may be a S/W ECC or SPI host controller embedded with a H/W ECC engine.



We can according to the actual ECC software/hardware configuration, set the parameters in device tree, the following are samples. (ECC framework's ECC engine type default is the on-die, for which we don't need to set anything.)

S/W BCH

```
spi_controller: spi@43c30000 {
        compatible = "mxicy,mx25f0a-spi";
        reg = <0x43c30000 0x10000>, <0xa0000000 0x4000000>;
        reg-names = "regs", "dirmap";
        clocks = <&clkwizard 0>, <&clkwizard 1>, <&clkc 15>;
        clock-names = "send_clk", "send_dly_clk", "ps_clk";
        #address-cells = <1>;
        #size-cells = <0>;

        flash@0 {
                compatible = "spi-nand";
                reg = <0>;
                nand-use-soft-ecc-engine;
                nand-ecc-algo = "bch";
                nand-ecc-strength = <8>;
                nand-ecc-step-size = <512>;
                spi-max-frequency = <25000000>;
                spi-tx-bus-width = <1>;
                spi-rx-bus-width = <1>;
        };
};
```

External ECC

```
spi_controller: spi@43c30000 {
        compatible = "mxicy,mx25f0a-spi";
        reg = <0x43c30000 0x10000>, <0xa0000000 0x4000000>;
        reg-names = "regs", "dirmap";
        clocks = <&clkwizard 0>, <&clkwizard 1>, <&clkc 15>;
        clock-names = "send_clk", "send_dly_clk", "ps_clk";
        #address-cells = <1>;
        #size-cells = <0>;

        flash@0 {
                compatible = "spi-nand";
                reg = <0>;
                nand-ecc-engine = <&ecc_engine>;
                spi-max-frequency = <25000000>;
                spi-tx-bus-width = <1>;
                spi-rx-bus-width = <1>;
        };
};

ecc_engine: ecc@43c40000 {
        compatible = "mxic,nand-ecc-engine";
        reg = <0x43c40000 0x10000>;
};
```

SPI-Host ECC

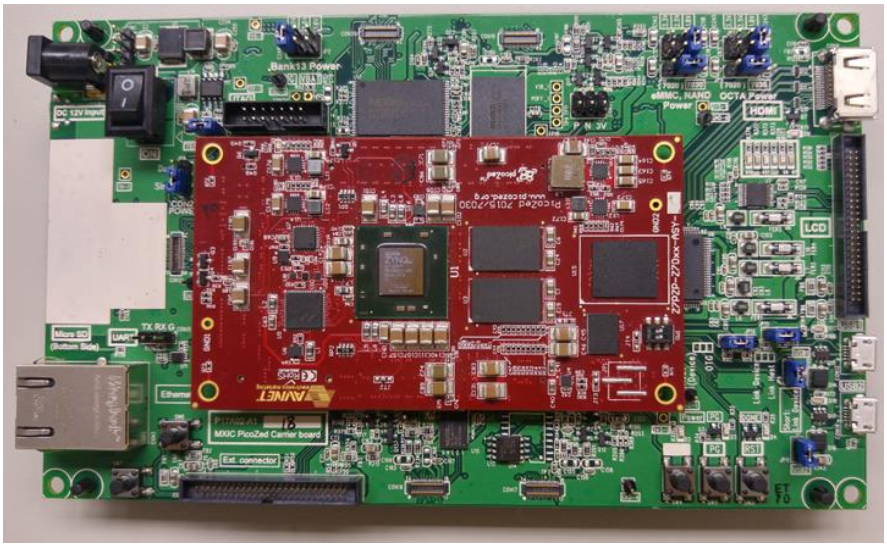
```
spi_controller: spi@43c30000 {
        compatible = "mxicy,mx25f0a-spi";
        reg = <0x43c30000 0x10000>, <0xa0000000 0x4000000>;
        reg-names = "regs", "dirmap";
        clocks = <&clkwizard 0>, <&clkwizard 1>, <&clkc 15>;
        clock-names = "send_clk", "send_dly_clk", "ps_clk";
        #address-cells = <1>;
        #size-cells = <0>;
        nand-ecc-engine = <&ecc_engine>;

        flash@0 {
                compatible = "spi-nand";
                reg = <0>;
                nand-ecc-engine = <&spi_controller>;
                spi-max-frequency = <25000000>;
                spi-tx-bus-width = <1>;
                spi-rx-bus-width = <1>;
        };
};

ecc_engine: ecc@43c40000 {
        compatible = "mxic,nand-ecc-engine";
        reg = <0x43c40000 0x10000>;
};
```

4. Evaluation

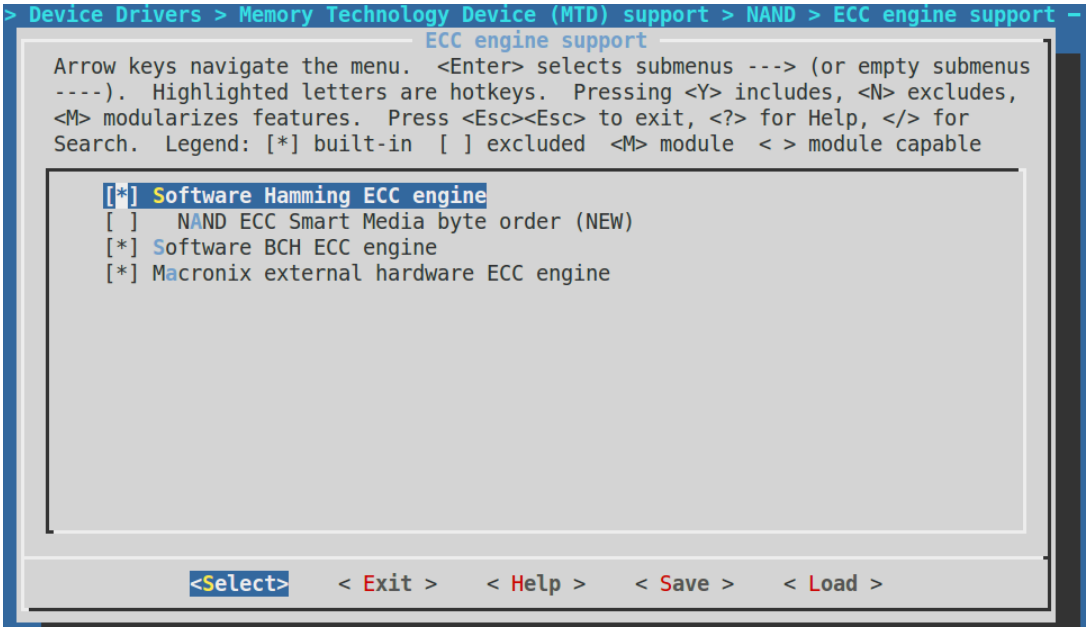
We will now verify that the SPI-NAND with generic ECC framework (we use the S/W BCH ECC, step size is 512 bytes and strength is 8 bits) can be used successfully.

4.1 Evaluation Environment

Our evaluation platform is *Macronix* PicoZed Carrier board & Zynq-7000 family SoC board with *Macronix*'s SPI Host controller IP, and *Macronix* SPI-NAND Flash chip (MX35LF2G14AC, without on-die ECC, 4-bit ECC/ 528B is required), and Linux kernel v5.11.



Ensure that device tree's configuration is setting for S/W BCH (according to the sample in the previous section), and the S/W ECC algorithm options (Software Hamming ECC engine & Software BCH ECC engine) are enabled in the kernel configuration.



4.2 Evaluation Method

We use *mtd-utilts* tools verify SPI-NAND with S/W (or H/W) ECC engine, for example:

- *nanddump*
  - It can show the main data and OOB area data of flash (with or without ECC correction)
- *nandtest*
  - NAND test tool, used to verify Erase/ Random Write/ Read-back and compare data, and show the ECC correction message & statistics results.
- *nandbiterrs*
  - This tool is useful to test multi-bit unexpected bit-flips on a NAND flash page, though the main purpose is to test ECC engine controller/driver robustness.
  - It has two ways to test: 1) artificially inserting bit errors until the

ECC fails, 2) re-writing the same pattern repeatedly until the ECC fails.

4.3 Evaluation Results

1. nanddump

```
$> ./nanddump -n -p -l 2048 /dev/mtd00
```

This is nanddump's output of read operation after erase/write (random pattern). This figure only shows a part of the whole contents (one page, 0x0-0x7FF), upper is main data of page, lower is ECC bytes (total 52 bytes, address: 0x12-0x63 in OOB area).

```
zynq> ./nanddump -o -p -l 2048 /dev/mtd0
ECC failed: 0
ECC corrected: 0
Number of bad blocks: 0
Number of bbt blocks: 0
Block size 131072, page size 2048, OOB size 64
Dumping data starting at 0x00000000 and ending at 0x00000800
```

```
0x00000700: 65 d7 28 8d db d5 0a 25 1f e6 2c 08 1d 90 21 49
0x00000710: 73 94 b4 aa bf 10 73 02 e2 be a8 06 8a 92 e2 0f
0x00000720: 52 2b ef ee c4 5a 47 0b 64 9d cc 56 b6 be 61 d9
0x00000730: 55 77 9b 71 07 01 47 65 97 0c fe 3b 70 37 c8 73
0x00000740: 88 fb 59 7d 7d 41 e1 c0 32 de e6 07 00 74 a6 c8
0x00000750: 7d a0 c7 10 5e e4 21 19 1d c7 db 7a c6 9e 34 70
0x00000760: 48 c5 6e 5a b2 4f a0 ab 0b 42 07 ce 35 df d9 a1
0x00000770: 35 72 cf d8 c1 2c da 98 de a3 17 43 56 b2 aa 45
0x00000780: 13 a0 38 3c df 17 67 e0 0c f2 75 19 e6 28 c9 6c
0x00000790: f1 a2 21 bc 01 4f e5 3d 0d eb 70 e9 a5 76 68 bd
0x000007a0: a9 4a ff fb d9 65 01 c2 af f7 b9 bf d4 3c 92 a0
0x000007b0: 98 93 ae f2 30 d5 7e 00 22 c2 2e 51 f9 7c 4b ce
0x000007c0: de 7b a4 fd 7a e2 8b 41 11 2a de 88 9b 4f e7 ea
0x000007d0: 18 c8 53 3d d3 e4 d0 e1 ff dc 4e 40 8f 30 61 4e
0x000007e0: 04 18 b3 a5 db 5b ec 24 a8 59 cf f7 dd 33 4b a2
0x000007f0: 51 24 14 a1 0a bc c3 96 35 bf c6 40 4d 27 8f ed
  OOB Data: ff ff ff ff ff ff ff ff ff ff ff ff ff 6e c3 55 9a
  OOB Data: 5a c3 88 86 f1 a0 a9 ef 88 08 a4 58 be a5 7f 8b
  OOB Data: d0 55 a7 c5 de 99 e2 ce e6 6c 25 32 4e c0 5c 97
  OOB Data: da 64 33 be da 19 f7 69 6c 31 fb 46 0b 7c b8 06
```

2. nandtest

```
$> ./nandtest /dev/mtd00
```

The nandtest used to verify erase/ random write/ read-back and compare data, and show the ECC correction message & statistics results.

The logs show that 2-bit errors were detected/corrected in the E/W/R testing of *nandtest*.

```
zynq> ./nandtest /dev/mtd0
ECC corrections: 0
ECC failures   : 0
Bad blocks     : 0
BBT blocks     : 0
06e40000: checking...random: crng init done
09a20000: reading...
 2 bit(s) ECC corrected at 09a20000
0ffe0000: checking...
Finished pass 1 successfully
```

3. nandbiterrs

```
$> ./nandbiterrs -i /dev/mtd00
```

It artificially inserts 0~n bit errors until the ECC fails.

The logs show that *nandbiterrs* generates biterrors from 0 to 9 bits, but it can only correct up to 8 bits of errors. (Because we are using 8-bit S/W BCH)

PS: There is a bug in the logic of ECC correction counting in S/W ECC engines (It already be fixed in v5.13), for the sake of good looks, first replace the result with Macronix's H/W ECC engine. (Only the number of bit errors is different)

```
zynq> ./mtd-utils/nandbiterrs -i /dev/mtd0

incremental biterrors test
Successfully corrected 0 bit errors per subpage
Inserted biterror @ 0/5
Read reported 1 corrected bit errors
Successfully corrected 1 bit errors per subpage
Inserted biterror @ 0/2
Read reported 2 corrected bit errors
Successfully corrected 2 bit errors per subpage
Inserted biterror @ 0/0
Read reported 3 corrected bit errors
Successfully corrected 3 bit errors per subpage
Inserted biterror @ 1/7
Read reported 4 corrected bit errors
Successfully corrected 4 bit errors per subpage
Inserted biterror @ 1/5
Read reported 5 corrected bit errors
Successfully corrected 5 bit errors per subpage
Inserted biterror @ 1/2
Read reported 6 corrected bit errors
Successfully corrected 6 bit errors per subpage
Inserted biterror @ 1/0
Read reported 7 corrected bit errors
Successfully corrected 7 bit errors per subpage
Inserted biterror @ 2/6
Read reported 8 corrected bit errors
Successfully corrected 8 bit errors per subpage
Inserted biterror @ 2/5
Failed to recover 1 bitflips
Read error after 9 bit errors per page
```

5. Conclusion

There are two ways to implement ECC function for SPI-NAND, *Macronix* believes that host-based ECC offers greater flexibility and advantages to our customers than on-die ECC. Some of these advantages *[1][3]* are as follows:

- Lower system cost:

  There's a huge cost savings moving from SPI-NOR to SPI-NAND. However, in this article, we discuss the added cost savings from removing the on-die ECC from the NAND die. The cost savings will more than offset the minor cost increase in the host-based IC.

- Increased performance:

  About 1.9X throughput improvement over on-die ECC engine

- Extended product life time:

Host-side ECC engines can use greater ECC strength to extend NAND lifetime

So, *Macronix* thinks the host to handle the ECC function is the right long-term way.

6. Reference

1. Psyche's presentation: https://youtu.be/9jpOZMYXYR0, PDF
2. Macronix's doc: Booting from NAND Flash Memory
3. Jim's article: Improving Performance, Reducing Cost Through Host-Based ECC
4. Miquel's NAND slides: https://elinux.org/images/3/3d/Raynal-understand-and-drive-your-nand.pdf
5. Miquel's ECC engine series patch message: Introduce the generic ECC engine abstraction
6. Miquel's ECC engine presentation: https://www.youtube.com/watch?v=kLzDbNUHPWg
7. Miquel's ECC engine ppt: https://bootlin.com/pub/conferences/2020/elce/raynal-ecc-engines/raynal-ecc-engines.pdf
8. Linux source code: https://github.com/torvalds/linux/tree/master/drivers/mtd/nand

Revision History

Revision No.	Descriptio	Page	D
Rev. 1.0	Initial Release	ALL	July 19, 2021

MACRONIX INTERNATIONAL CO., LTD.