ECC engines

Miquèl Raynal

miquel@bootlin.com

© Copyright 2004-2020, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
Miquèl Raynal

- Embedded Linux engineer at Bootlin
  - Embedded Linux expertise
  - Development, consulting and training
  - Strong open-source focus
  - https://bootlin.com

- Contributions
  - Maintainer of the NAND subsystem
  - Co-maintainer of the MTD subsystem
  - Kernel support for various ARM SoCs

- Living in **Toulouse**, south west of France
A bit of context

Miquel Raynal
miquel@bootlin.com
You want to share an information

The communication medium is subject to disturbances

What do you do?

In a crowd you would either...

- Speak louder?
  - Uses more power in the case of telecommunications
  - Would need to decrease storage media density

- Repeat yourself?
  - Adds redundancy
  - Introduces more latency
Redundancy

- Provide the original data to an algorithm
- Retrieve the (transformed) data, including check/redundancy information
  - We usually prefer to transfer readable data, so the original data prepends within the code
  - Longer than the data you actually want to transmit
- The point of this code being, the receiver must be able to
  - Detect one or more errors
  - Eventually correct one or more errors
- Not only reserved to communications
- Already widely used with storage media as well!
ECCs in communications

- Case of radio audio communications
  - NATO phonetic alphabet
    - Lima
    - India
    - November
    - Uniform
    - X-ray

- Probably the most widely known ECC
- All words are very different to the ears
- Mathematiciens call that the distance
Let’s take the number 0xA, b1010
A single disturbance could produce b0010
How do you know that 0x2 is not the right number?
Any change leads to a (in appearance) valid number
Simplest algorithms

- Repeating may be a solution
  - Send twice the same bit
    - \texttt{b1010} becomes \texttt{b11001100}
    - Detection of a single bit error
    - No correction
Simplest algorithms

- Repeatating may be a solution
  - Send twice the same bit
    - b1010 becomes b11001100
    - Detection of a single bit error
    - No correction
  - Send three times the same bit
    - b1010 becomes b11100011100
    - Detection of a single bit error
    - Automatic correction by majority vote
    - Very costly!
ECCs are everywhere

- Parity bits in UART communications
  - A byte may be composed of
    - 7 bits of data
    - 1 parity bit
  - The parity bit is selected to match either an even or an odd parity
    - Example: 0x4A (b1001010) has 3 binary 1
    - If we look for an even parity, we will then append a 1
    - If the message has an odd parity it is assumed to be corrupted

- 1-bit error detection is achieved with a 15% overhead
- Much less than the 100% overhead of the “repeating” algorithm!
ECC for storage

- RAM chips embed simple hardware ECC algorithms
  - Old technologies used parity bits
  - Then 1-bit correction algorithms
  - Silicon vendors tend to move towards more complex on-the-fly corrections

- Compact Disks
  - Are intrinsically less prone to bit errors
  - Errors come from external scratches or dust
  - Unlike RAMs, errors happen in batch
  - Philips norm covers the loss of up to 4096 consecutive bits (this is a 1 millimeter thick scratch!)

Elixir 512MB DDR RAM M2U51264DS8HC3G-5T for desktop computers

Flat view of a CD-R
NAND and bitflips, a love story

Miquèl Raynal

miquel@bootlin.com
NAND technology 101

It’s cheap
It’s cheap

It’s intrinsically unstable
NAND technology 101

It’s cheap

It’s intrinsically unstable

ECC is mandatory
NAND technology 102

- NAND devices are made of a huge amount of tiny NAND cells
- A cell is like a bucket with a small hole
- An empty bucket is seen as a binary 1
- A filled bucket is seen as a binary 0
- Multiple reasons can cause a NAND cell to not return the right data:
  - Time
    - remember, there is a hole in the bucket!
  - Intensive use (too many erase cycles) damages the cell
    - the hole gets bigger!
  - Read disturbances
    - looking into a bucket shakes the other one
  - Level sensing
    - when do we consider the bucket full/empty?
- A more scientific explanation of the NAND technology internals available here: conference / slides
Particularly true with newer chips

- NAND cells are smaller
- Density rises
- The probability of bit error as well (due to the inherent disturbances)

We need reliable corrections that suit the chip requirements

Stronger corrections involve:
- More processing power
- Additional delays
- A bigger overhead
The host controller provides to the ECC engine a chunk of data

The ECC engine processes the chunk and produces ECC bytes
  - Usually the processed data is kept identical
  - The ECC bytes are stored in the out-of-band area

Repeat this operation for all the data chunks contained in the page

Write the entire page to the storage medium
ECC engine mission: read path

- Raw data and ECC bytes (possibly corrupted) are retrieved

- The ECC engine processes all the available data, chunk after chunk, to:
  - Detects bit errors
  - Eventually corrects them

- Return the original data to the caller and report a status
Hamming algorithm

- Very popular with older/stronger Single Level Cell (SLC) chips
- Efficiently corrects up to 1 bit error per chunk
- Detects up to 2 bit errors per chunk
- Invented in 1950 to cover defects from punched card readers!
- Most of the existing raw NAND controllers embed an hardware Hamming ECC engine
- Linux provides a software Hamming ECC engine

Historic portrait of Richard W. Hamming
BCH algorithm

▶ Invented independently in 1959 by Alexis Hocquenghem and 1960 by Raj Bose and D. K. Ray-Chaudhuri
▶ Very powerful and flexible: fits almost any kind of (NAND) requirement
   ▶ Adapts to almost any strength over any chunk size
   ▶ Carries the data unaltered
   ▶ Very good ratio overhead/correction capabilities
   ▶ Only limited by the available out-of-band area
▶ Read path almost 10 times more complex than the write path
   ⇒ Better if offloaded to hardware
▶ But still, BCH decoding is considered as rather inexpensive compared to its correcting capabilities
▶ Leverages polynomial algebra over binary data
▶ Reverse engineering session of a hardware BCH ECC engine: https://bootlin.com/blog/supporting-a-misbehaving-nand-ecc-engine/
▶ Linux also provides a customizable software BCH ECC engine
Reed-Solomon algorithm

- Introduced in 1960 by Irving S. Reed and Gustave Solomon
- Considers symbols instead of bits
  - Many bit-errors in a single symbol appear as a single failure
  - Makes RS codes well suited to fight against burst errors
- Treats “lack of data” and “bit failures” differently
  - Given $t$ the number of check symbols, it can correct:
    - up to $t$ missing symbols (provided that the algorithm know their position) or
    - up to $t/2$ unlocated errors otherwise
- A bit less common than BCH codes in the NAND world
- Base of the CIRC ECC algorithm used for CD’s (Cross Interleaved Reed-Solomon Code)
ECC engines support

Miquèl Raynal
miquel@bootlin.com

© Copyright 2004-2020, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
The ECC engine in the raw NAND world

How people usually see their hardware:

- Kernel, drivers and embedded Linux
- Development, consulting, training and support

NAND host controller

NAND device

NAND bus
The real situation:

- Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
Current situation for parallel NANDs

- Historically, raw NAND device, NAND bus, NAND controller and ECC engine were treated by Linux as a single entity.
- Recently, we separated the NAND device and the NAND controller representations.
- The raw NAND controller and its embedded hardware ECC engine are still mixed in practice.
- We recently pushed in favor of the distinction between:
  - `struct nand_chip *chip`
  - `struct nand_controller *controller`
  - `struct nand_ecc_ctrl *ecc`
Current situation for serial NANDs

- Only “on-die” ECC engines:

![Diagram showing the connection between SPI host controller, SPI bus, hardware ECC engine, and SPI-NAND device]

[40x254]Current situation for serial NANDs

- Only “on-die” ECC engines:

![Diagram showing the connection between SPI host controller, SPI bus, hardware ECC engine, and SPI-NAND device]
- Support added much more recently (v4.19)
- At this time, only on-die ECC engines were supported
- Software engines not available (yet)!
- We see new devices coming out without embedded engines
  - Cheaper to manufacture?
  - More powerful (for larger corrections) to offload to dedicated hardware
  - Even more when mutualizing between several chips
- SPI-NAND subsystem not ready for that
External/pipelined ECC engine
What describes ECC engines?

- What should discriminate two engines?
- Common properties may be used to pick the most appropriate one, like:
  - The type of engine
  - The possible strengths
  - The supported chunk sizes (also called step size, or ECC size) on which the correction applies

```c
/**
 * struct nand_ecc_props - NAND ECC properties
 * @engine_type: ECC engine type
 * @placement: OOB placement (if relevant)
 * @algo: ECC algorithm (if relevant)
 * @strength: ECC strength
 * @step_size: Number of bytes per step
 * @flags: Misc properties
 */
struct nand_ecc_props {
    enum nand_ecc_engine_type engine_type;
    enum nand_ecc_placement placement;
    enum nand_ecc_algo algo;
    unsigned int strength;
    unsigned int step_size;
    unsigned int flags;
};
```
How is the engine’s configuration chosen?

The core must tune the engine’s configuration to best fit the engine’s capabilities, the NAND part requirements, the subsystem defaults, the user desires,...

/**
 * struct nand_ecc - Information relative to the ECC
 * @defaults: Default values, depend on the underlying subsystem
 * @requirements: ECC requirements from the NAND chip perspective
 * @user_conf: User desires in terms of ECC parameters
 * @ctx: ECC context for the ECC engine, derived from the device @requirements
 * @ondie_engine: On-die ECC engine reference, if any
 * @engine: ECC engine actually bound
 */

struct nand_ecc {
    struct nand_ecc_props defaults;
    struct nand_ecc_props requirements;
    struct nand_ecc_props user_conf;
    struct nand_ecc_context ctx;
    struct nand_ecc_engine *ondie_engine;
    struct nand_ecc_engine *engine;
};
For each NAND device, the core must find the engine to be used and tune it appropriately.

```c
nanddev_ecc_engine_init(struct nand_device *nand)
{
    /* Look for the ECC engine to use */
    nanddev_get_ecc_engine(nand);

    /*
     * Configure the engine:
     * balance user input and chip requirements
     */
    nanddev_find_ecc_configuration(nand)
    {
        nand_ecc_init_ctx(nand);

        if (!nand_ecc_is_strong_enough(nand))
            pr_warn("weak ECC...\n");
    }
}
```
SPI-NAND Bindings

On die ECC engine

```c
&spi_host {
  flash@0 {
    compatible = "spi-nand";
    reg = <0>;
    nand-ecc-engine = <&flash>;
  };
};
```

External ECC engine

```c
&spi_host {
  flash@0 {
    compatible = "spi-nand";
    reg = <0>;
    nand-ecc-engine = <&flash>;
  };
};
```

Software ECC engine

```c
&spi_host {
  flash@0 {
    compatible = "spi-nand";
    reg = <0>;
    nand-use-soft-ecc-engine;
    nand-ecc-algo = "bch";
  };
};
```

On host ECC engine

```c
&spi_host {
  nand-ecc-engine = <&ecc_engine>;
  flash@0 {
    compatible = "spi-nand";
    reg = <0>;
    nand-ecc-engine = <&spi_host>;
  };
};
```

```c
ecc_engine: ecc@xxxxxxxx {
  compatible = "mxic,nand-ecc-engine";
  reg = <xxxxxxxx yyyyyyyy>;
};
```
Hooks to provide

- `init/cleanup_ctx()` one time configuration/allocations
- `prepare_io_req()` gets called for any page I/O requesting ECC correction, enables the engine, save information on the request, etc
- `finish_io_req()` gets called for any page I/O requesting ECC correction, ends the transfer, disables the engine, reports read errors if relevant, etc

```c
/**
 * struct nand_ecc_engine_ops - ECC engine operations
 * @init_ctx: given a desired user configuration for the pointed NAND device, requests
 *  the ECC engine driver to setup a configuration with values it supports.
 * @cleanup_ctx: clean the context initialized by @init_ctx.
 * @prepare_io_req: is called before reading/writing a page to prepare the I/O request
 *  to be performed with ECC correction.
 * @finish_io_req: is called after reading/writing a page to terminate the I/O request
 *  and ensure proper ECC correction.
 */

struct nand_ecc_engine_ops {
    int (*init_ctx)(struct nand_device *nand);
    void (*cleanup_ctx)(struct nand_device *nand);
    int (*prepare_io_req)(struct nand_device *nand, struct nand_page_io_req *req);
    int (*finish_io_req)(struct nand_device *nand, struct nand_page_io_req *req);
};
```
The final NAND ECC engine structure

This structure will be registered at probe time and saved into a system-wide list of available ECC engines.

```c
/**
 * struct nand_ecc_engine - ECC engine abstraction for NAND
 * @dev: Host device
 * @node: Private field for registration time
 * @ops: ECC engine operations
 * @priv: Private data
 */

struct nand_ecc_engine {
    struct device *dev;
    struct list_head node;
    struct nand_ecc_engine_ops *ops;
    void *priv;
};
```
What’s next?

- Bootloaders don’t have support for these external engines yet
- The raw NAND core carries so much history that is very difficult to make it fit the ECC abstraction without breaking numerous drivers
- New ECC engine drivers to come?
- NOR flashes carrying embedded Hamming ECC engines due to automotive safety constraints (ASIL-B/ASIL-D).
  → Will it soon be offloaded? (please don’t)
Questions? Suggestions? Comments?

Miquèl Raynal

miquel@bootlin.com

Slides under CC-BY-SA 3.0

References

▶ RAM picture slide ??:
Unknown author. Possibly Cyberdex (given the authors right revendication). “Personnal work” supposed (given the revendication). Public domain,

https://commons.wikimedia.org/w/index.php?curid=647267

▶ CD-R picture slide ??:
Author: Ubern00b, Personnal work, CC BY-SA 3.0,

https://commons.wikimedia.org/w/index.php?curid=226419

▶ R. Hamming picture slide ??:
Source (WP: NFCC 4), Fair use,