FEE Memory Management – Basic Mechanisms

Product Information

Version 2.02.01

Authors	Peter Lang/Marcus Kratzsch
Status	Released

Document Information

History

Author	Date	Version	Remarks
Peter Lang	2011-07-01	1.00.00	Creation of document
Dr. Günther Heling	2011-12-22	1.00.01	Review and minor modifications
Marcus Kratzsch	2012-01-05	2.00.00	Complete Rework
Marcus Kratzsch	2012-02-01	2.01.00	Rework after reviews
Marcus Kratzsch	2012-02-08	2.01.01	3.2.1.1 Add note; 4. Add hints
Dr. Günther Heling	2012-02-09	2.01.02	Review and minor modifications and Release
Marcus Kratzsch	2012-03-09	2.02.00	Add chapter 5
Christian Kaiser	2012-10-05	2.02.01	Referenced Version in Chapter 5

Contents

1 Introduction

1.1 Abstract

This document gives a brief description of basic FEE mechanism and therefore is dedicated to all potential FEE users.

First the relevant difference between EEPROM and flash memory concerning data management will be explained. Then common FEE requirements and concepts will be discussed, before focusing functionalities of the MICROSAR FEE. Finally some essential configuration and integration hints are given.

1.2 Abbreviations

Term	Description
FEE	Flash-Eeprom-Emulation.
Flash memory	FLASH memory is a kind of memory to store permanent data
Page	Smallest addressable memory unit
Physical Sector	Smallest erasable memory unit (contains multiple pages)
Logical Sector	Alignment of multiple physical sectors.
Chunk	Reserved address space for a data block
BSS	Background (logical) Sector Switch
FSS	Foreground (logical) Sector Switch
Parallel sector usage	Both logical sectors containing valid data
Single sector usage	Only one of the two logical sectors contains valid data
Partition	Alignment of at least two logical sectors

2 FEE Basics

2.1 Difference between EEPROM and Flash memory

While flash memory is technically a type of EEPROM, the term “EEPROM” is used in this document to refer specifically to non-flash EEPROM.

EEPROM is erasable and rewritable in small blocks of same size (typically 1 to 4 bytes). Data can be easily rewritten on the same memory unit, which leads to a predictable cycle time for rewriting data (typically up to 20 ms). Figure 2-1 shows a simple EEPROM rewrite operation with a block size of 4 bytes.

Note
There are also EEPROMs on the market where the erase step is not necessary.

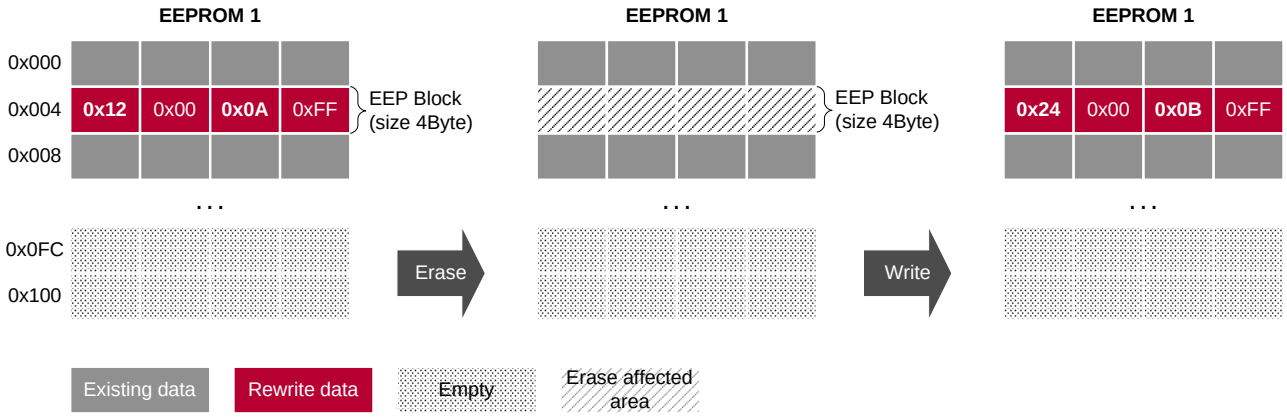

Figure 2-1 Simple EEPROM rewrite operation

Compared to EEPROM, Flash memory has larger memory elements. The smallest addressable memory unit (**page**) is not singularly erasable; hence multiple pages are aggregated to the smallest erasable memory unit (**physical sector**). To rewrite data on the same page, the related physical sector has to be erased, which affects all other stored data within the physical sector. Figure 2-2 illustrates such a rewrite operation. In the example the page size is 4 bytes and the physical sector size is 256 Bytes.

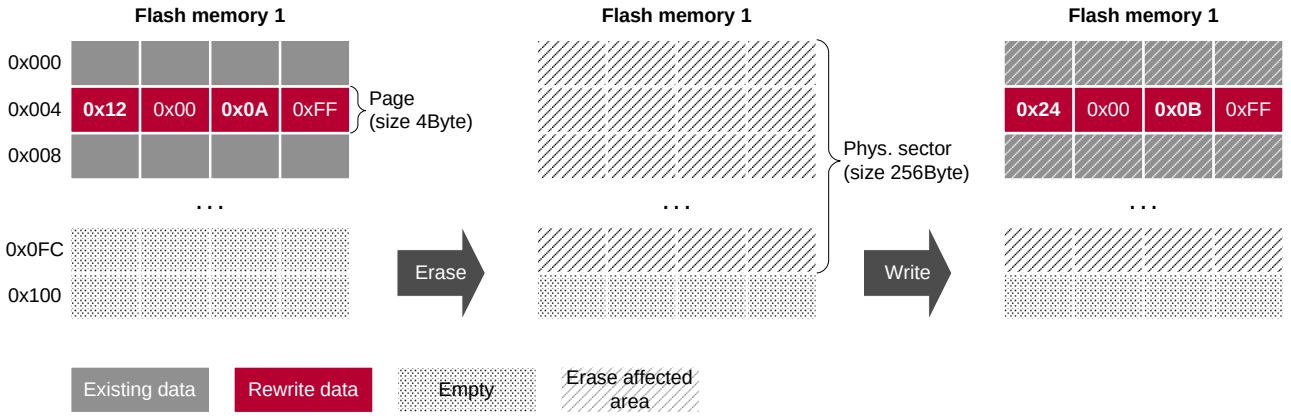

Figure 2-2 Simple Flash memory rewrite operation

The page size as well as the physical sector size is hardware dependent and usually larger than in our example. Table 2-1lists sizes of frequently used MCUs.

MCU Family	Physical sector size	Page size
S12XE	256 Byte – 1024 Byte	2-8 Byte
MPC56xx	16kByte - 128kByte	4-16 Byte
TriCore 17xx	64kByte - 128kByte	128-256 Byte

Table 2-1 Page and physical sector sizes of selected MCUs

Different physical sector size also implies different time periods for sector erase. At scales to hundreds of milliseconds per 16KB, an erase operation can last up to seconds which is a very long time period for embedded real time systems.

2.2 Common FEE mechanism

From application side, Flash-EEPROM-Emulation (FEE) is measured according the following criteria:

- **Flash usage:** Provide enough memory space to the application, i.e. minimize (total) memory consumption for a certain amount of data.
- **Flash lifetime:** Assure sufficient lifetime, i.e. minimize write and erase cycles to maximize flash endurance[1] and flash retention.[2]
- **FEE performance:** Ensure acceptable execution time, i.e. minimize runtime to read data and to (re)write data.
- **Data Safety:** Assure availability of (all) valid data in terms of data loss and data integrity. i.e. minimize the probability that data get lost or get corrupted.

[1] Flash endurance is the maximum possible number of program/erase cycles over flash lifetime and is limited by technology.
[2] Flash retention is the time period the flash is able to keep programmed data. Retention time decreases with every erase cycle.

These criteria are partially competitive and it depends on hardware configuration, FEE algorithms, and FEE configuration to which extend the criteria can be fulfilled. For example, it is not possible to get the highest performance by minimized flash usage (refer to chapter 2.4). Importance of each criterion depends on the specific application.

To achieve efficient memory usage, every FEE requires an algorithm to avoid rewriting data always on the same page. There are different implementation options, each with pros and cons, but all of them share two common principles:

- **Walking data**, i.e. for every data update a new instance (version) will be created, stored on another location and marked as actual value.
- **Garbage collection** to release memory from outdated data, i.e. valid data has to be reorganized if necessary and at least one physical sector will be erased.

2.3 Sample FEE implementation

One obvious FEE implementation is to split flash memory in half (two logical sectors). Starting at the first logical sector, every new data version will be stored in the next free page until the logical sector is filled (walking data).

For garbage collection a sector switch is performed, where the latest version of every data is copied from the first to the second logical sector, before the first one is erased. Now data layout reflects initial state and operation starts from scratch. Figure 2-3 illustrates this implementation with two data blocks.

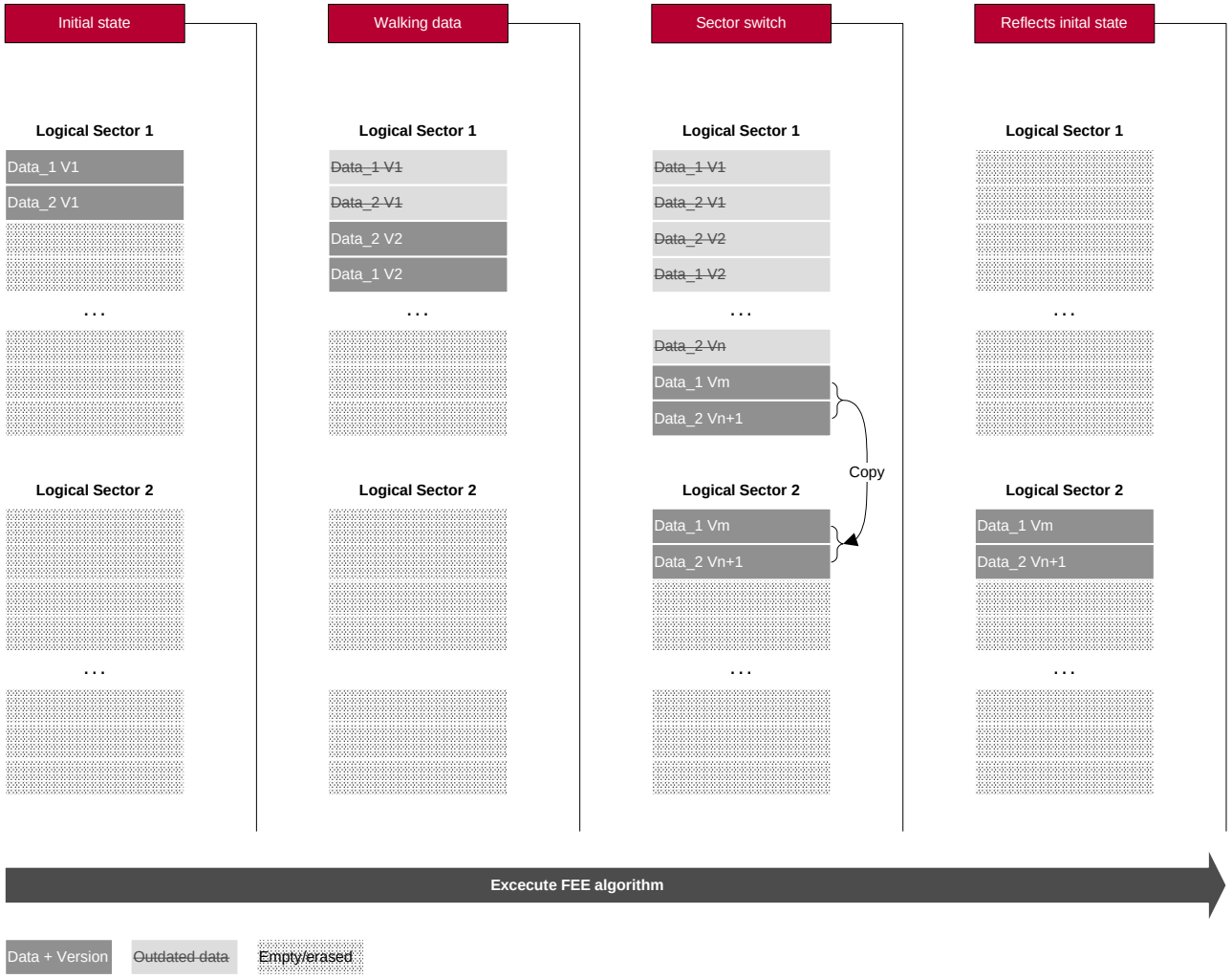

Figure 2-3 Sample implementation of FEE algorithm

Let's have a look, how this implementation fulfills the requirements in chapter 2.2:

Flash usage is well balanced. The alignment of data allows minimal gaps within flash memory. Also data safety can be assured all the time, because every data is copied before the logical sector is erased. But: flash lifetime as well as FEE performance is limited, since always all valid data have to be copied during sector switch.

2.4 Deeper look into criteria for FEE implementations

FEE sample algorithm in chapter 2.3 is described in a simplified way. To understand the challenge of FEE implementation, it is helpful to have a closer look at the criteria from chapter 2.2.

2.4.1 Flash usage

Every walking data concept needs data management information to find latest data version or next storage location. Therefore little FEE data management overhead is required to reduce total amount of stored data. Beyond that, (unused) gaps between stored data must be avoided for flash usage optimization.

2.4.2 Flash lifetime

As mentioned, flash lifetime depends on write/erase cycles. The best flash lifetime can be reached, when flash memory units are used strictly alternating and every data version is stored only once, i.e. no copy operations occur. Of course total cycle amount can be reduced by shrinking flash fill level, respectively using larger flash memory.

2.4.3 FEE performance

To achieve short runtime for data reading, a fast search algorithm to scan data information is required. Usually search time increases with every new data version stored, until garbage collection runs. This is because more potential locations have to be scanned to find latest data version. Also a possibility to read data during garbage collection is needed, due to its long runtime.

The runtime to (re)write data depends on search time to find next free storage location, the amount of data which has to be written and whether garbage collection has to be performed before. Since timing of garbage collection is unpredictable and erasing a physical sector is very time consuming (up to seconds), FEE performance varies very much within the same FEE implementation.

Shortest runtime of garbage collection is given, when no data has to be copied and only one physical sector has to be erased.

Summary: Best FEE performance is given, when locations of last data version and next free page are known, when FEE data management overhead is minimal (data amount)[3] and when garbage collection only runs when no FEE requests pending (FEE idle).

2.4.4 Data safety

To assure that no data gets lost, FEE has to keep the latest version of every data all the time, i.e. no interims copy on other medium (e.g. RAM) is allowed. Furthermore FEE algorithm has to provide mechanisms like check sums to detect corrupted data.

Note
In contrast to EEPROM, storing important data twice on flash memory is less reasonable. Due to walking data, previous data versions are available.

Data safety can be a weak point if frequent ECU resets occur during write operations or garbage collection. In this case, flash memory may contain invalid data fragments.

3 MICROSAR FEE 6.x Concept

Compared to sample FEE in chapter 2.3, MICROSAR FEE 6.x offers several enhanced mechanism to allow better balancing of the above mentioned criteria (chapter 2.4).

[3] Huge FEE data management overhead means more data to handle which influences FEE performance negatively.

3.1 Walking data with chunks

To improve FEE performance, every data is organized by **chunks**. A chunk reserves flash address space for its related data and has a configurable size. Every data will be updated within its related chunk, until a new chunk has to be allocated. Figure 3-1 demonstrates the principle by updating Data_1 twice:

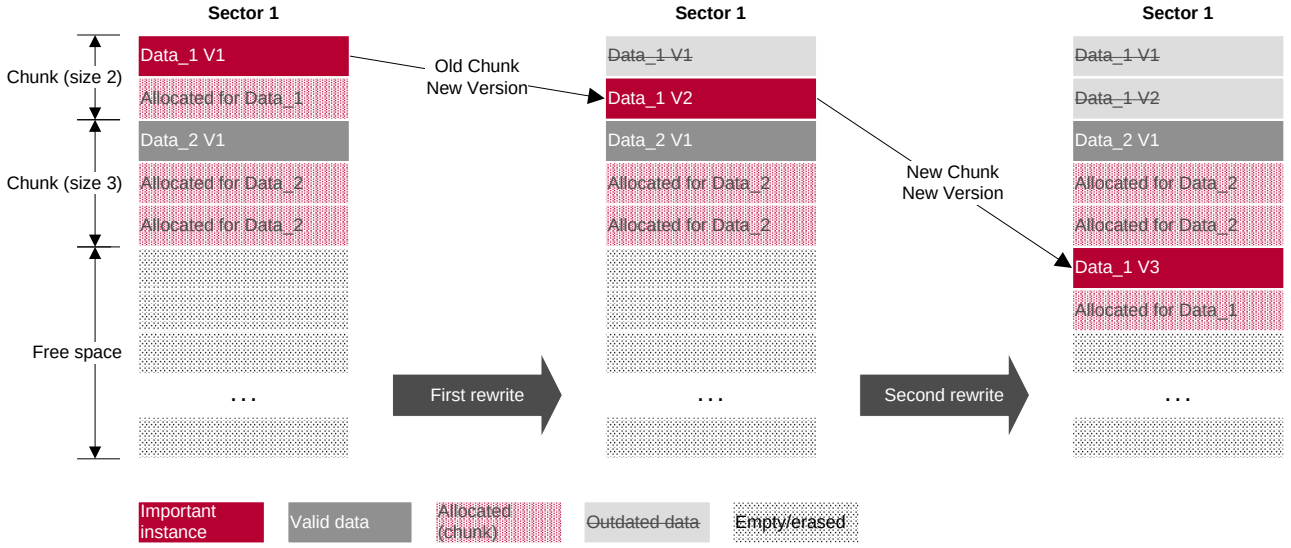

Figure 3-1 Walking data with chunks

Search algorithm now looks for the latest chunk, before scanning for the latest data version within this chunk. This reduces search time, because fewer locations have to be scanned.

A further advantage of chunks is the reduction of FEE data management overhead, because same management information can be used for more than on data instance. Flash usage is improved, compared to a simple walking data concept, where every data instance needs separate management information.

Optimal chunk size for each data depends on its number of write cycles. Data which only has to be written once or during service interval should be configured with chunk size one. For frequently written data, chunk size should be larger and the optimal size should be evaluated stepwise during configuration.

3.2 Enhanced garbage collection

To keep FEE hardware independent, implementation divides the complete available flash memory into two logical sectors4, like our sample FEE in chapter 2.3. This means when garbage collection finishes all valid data is located in one logical sector, whereas the other logical sector is completely erased and ready to store new data again.

[4] Some MCUs on the market only offer two physical sectors.

Garbage collection itself is split into background logical sector switch (BSS), that is interruptible and foreground logical sector switch (FSS) that is not interruptible. Both sector switches are activated via configurable threshold.

This implementation allows improvement of FEE performances as well as flash lifetime, because both logical sectors can contain valid data. We call this **“parallel logical sector usage”**.

3.2.1 Parallel (logical) sector usage

In default configuration of FEE, both logical sectors are used in a parallel way (i.e. both sectors contain valid data). As soon as the BSS threshold is reached, FEE begins to copy all valid data from the “older” logical sector to the “newer” logical sector. If any data should be rewritten during this copy operation, copying will be halted; updated data will be appended and finally the copy operation will be continued. Figure 3-2 shows FEE with parallel sector usage.

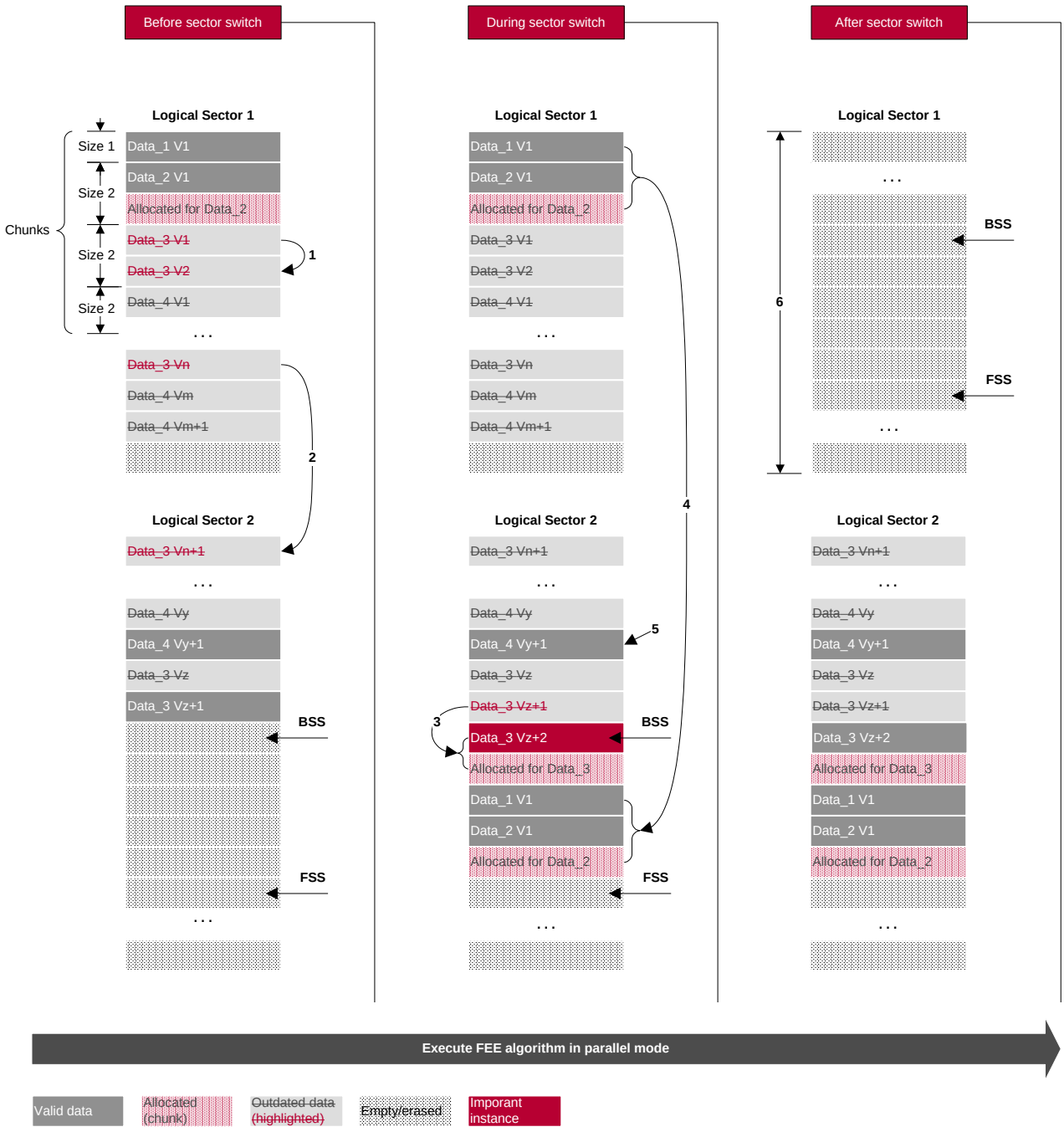

Figure 3-2 Logical sector switch in parallel mode

1. Walking data begins like described in chapter 3.1. Here “Data_3” changes from “Version_1” to “Version_2” by data update.
2. At the end of “Logical_Sector_1”, there is not enough space to allocate a new chunk for “Data_3”. Therefore the chunk will be allocated in “Logical_Sector_2” where the new version of “Data_3” is stored.
3. Once BSS threshold is reached (here by “Data_3”), the new instance of “Data_3” is stored and BSS starts with logical sector switch.

4. BSS copies all valid data (here “Data_1 V1” and “Data_2 V1”) from “Logical_Sector_1” to “Logical_Sector_2”. Because BSS is interruptible new write requests can be executed during BSS.
5. Latest version of “Data_4” is already in “Logical_Sector_2”, which speeds up sector switch and increases flash lifetime.
6. When all valid data is located in “Logical_Sector_2”, “Logical_Sector_1” will be erased and BSS has finished. The new thresholds are located now in “Logical_Sector_1”.

In parallel mode, FSS should only act as “last-chance” processing, i.e. when flash memory is almost full and sector switch needs high priority. FSS suspends all pending requests for data updates and finishes the logical sector switch.

3.2.1.1 Reset robustness

If FEE write operation is interrupted, by calling “Fee_Cancel” or MCU resets, invalid data fragments are left in flash memory. This has little influence as long as enough free flash memory space is available. As stated before, FSS is not intended to be interruptible. But in case of MCU resets, caused by e.g. instable voltage, FSS might be aborted. If FSS is aborted too often, it will be possible to run out of flash memory before copy operation is finished (logical sector overflow).

FEE can report this situation to the application, which can decide whether to loose potential valid data (by erasing one of the logical sectors), or to lock FEE for further write requests (**write deny**) which keeps all valid data available. Figure 3-3 illustrates such behavior.

Note
The report mechanism has to be activated by enabling Error Callback. If Error Callback is disabled and sector overflow occur, FEE continues by erasing the “newer” logical sector. This keeps FEE running, but might cause data loss.

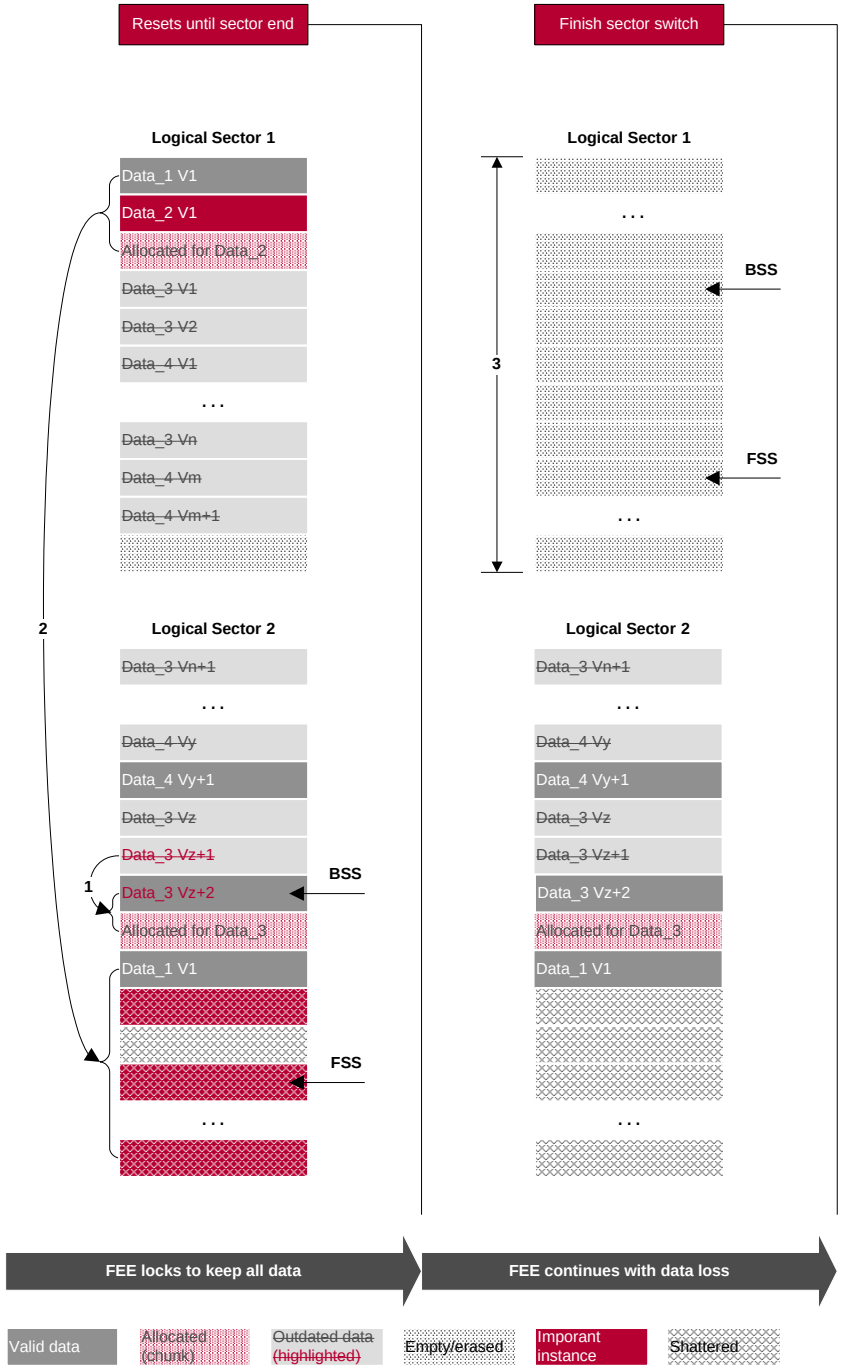

Figure 3-3 Logical sector switch in parallel mode under resets

1. New version of “Data_3” starts BSS
2. All valid data from “Logical_Sector_1” should be copied to “Logical_Sector_2”. This was successful for “Data_1”. While copy “Data_2”, frequent ECU resets stops copy operation, even by passing FSS threshold. By reaching the end of “Logical_Sector_2”, FEE can report this situation to the application.
3. Application now can decide to keep all data (FEE locks), or to erase one logical sector and to loose data, here “Data_2”.

Compared to our sample implementation (chapter 2.3) it is not possible to safely erase always one logical sector without potential loss of data. Thus using FEE in parallel mode implies less data safety.

As countermeasure FEE provides APIs to activate or deactivate FSS (Fee_EnableFss/ Fee_DisableFss) in certain situation. In combination with voltage monitoring, the application can avoid to perform FSS during potential voltage instability (e.g. engine start).

3.2.1.2 Critical data blocks

Flash memory typically stores a variety of data, with different implications of data loss. For example, severity of losing the Vehicle Identification Number (VIN) is much higher than losing the last setup of display luminance.

To ensure that no important (critical) data gets lost, even in case of frequent resets (potential write deny), data blocks can be configured as **“critical”**. FEE makes sure, that all critical data blocks are located in one logical sector, i.e. once a “critical” data moves to the other logical sector; FEE initiates a logical sector switch. This improves data safety for all “critical” data blocks.

Note
With every data block marked as “critical”, FEE performance and flash lifetime decreases. In case all data blocks are marked as “critical”, FEE acts in single logical sector usage, like our sample FEE (chapter 2.3).

3.2.1.3 Single (logical) sector usage

Single logical sector usage is a special case of parallel logical sector usage, where FEE only performs FSS. Figure 3-4 illustrates this configuration in detail.

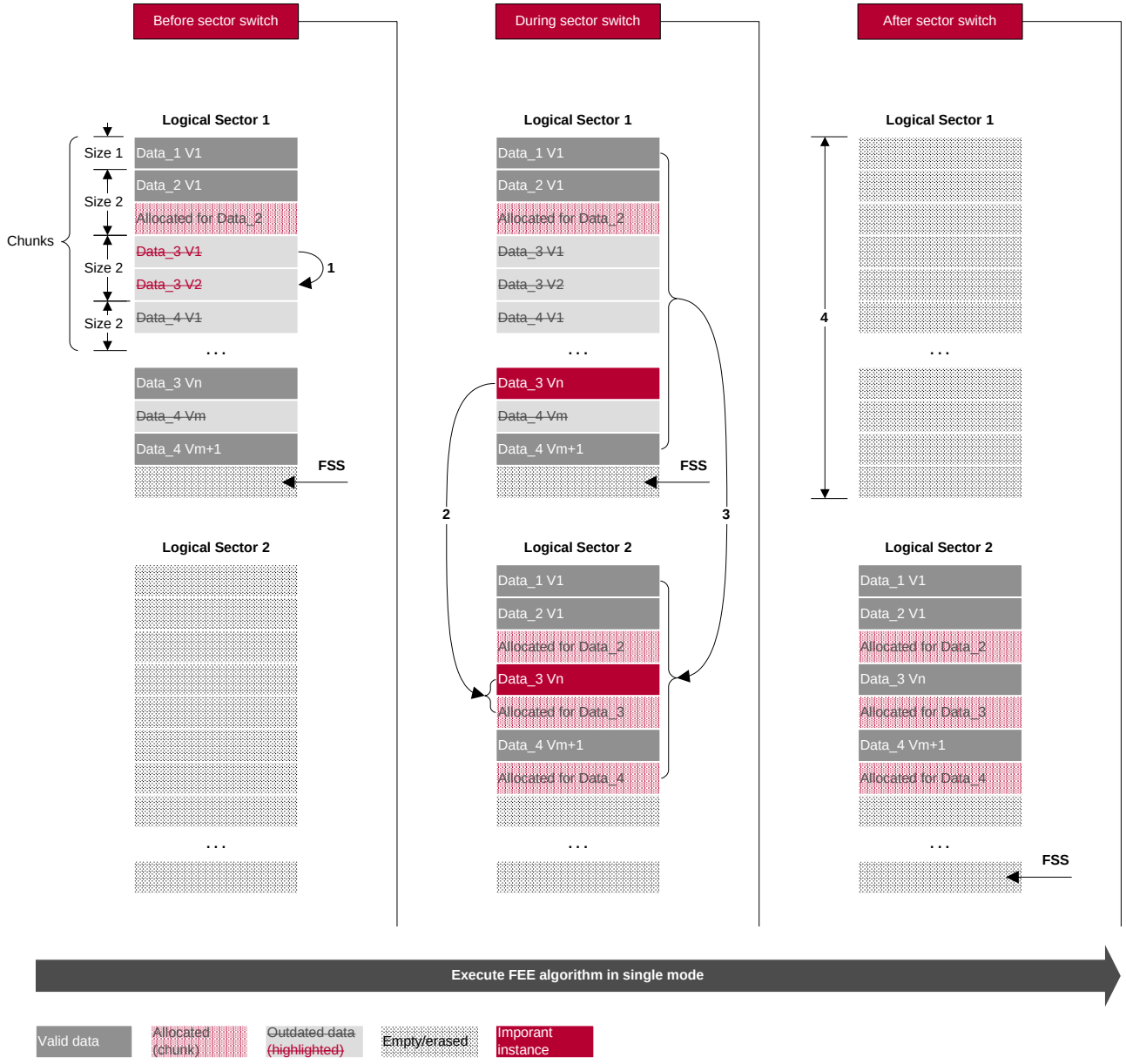

Figure 3-4 Logical sector switch in single mode

1. Walking data starts, until “Logical_Sector_1” is filled.
2. A new request to update “Data_3” starts logical sector switch; but the data update will be displaced after sector switch completes
3. All valid data from “Logical_Sector_1” is copied to “Logical_Sector_2”, including “Data_3”
4. Sector switch completes by erasing “Logical_Sector_1”. Now the update of “Data_3” can be performed (not shown in figure)

In single logical sector usage, FEE is completely reset robust, because one logical sector can always be erased without data loss.

Note
Be aware, that this configuration may cause FEE performance and flash lifetime problems.

4 FEE configuration and integration hints

When using FEE within a specific system, first of all the amount of data should be considered. As rule of thumb, the total amount of data should not exceed 50% of the smallest logical sector. Check if data can be stored apart from flash memory managed by the FEE. This is useful for:

- Static (constant) data which might be stored e.g. in program flash.
- Default data, where application can fall back in case of data loss can typically be stored in program flash.

If data which should be stored in FEE is defined, FEE has to be configured and integrated to find an optimal compromise between flash lifetime, FEE performance and data safety. Therefore the optimal solution has to be evaluated stepwise by estimations and simulations. Check

- Which data is really critical? Only those data blocks should be marked as “critical” where loss of latest version is not acceptable.
- Which chunk size is useful for each data block? As rule of thumb, the higher the write frequency the larger the chunk size. Data that is written rarely, e.g. once during lifetime or only during service interval, should be configured with chunk size one.

For proper FEE functionality, ECU voltage control is the key. The critical point is voltage instability during FSS, whose runtime is not predictable. Try to:

- Avoid write requests during voltage instability, e.g. engine start or during “controlled” resets, like fast power down or shut down for Terminal_15 ECUs.
- Use the provided APIs to suspend/resume FSS in situation where under voltage may occur.
- Mitigate frequent cyclic ECU resets as much as possible, using an appropriate voltage control.
- Provide enough reserves to finish write requests, e.g. by buffer capacitor. This is especially needed for Terminal_15 ECUs or to write crash data.
- Use Error Callback to decide from application side how to continue in case of logical sector overflow. The Error Callback can also notify the application when critical fill level exceeds, i.e. FSS has to start.

And last but not least system design has to care about startup behavior. When data which is quickly needed after startup is stored in FEE, system has to provide a cascaded startup mechanism to load FEE data stepwise.

5 MICROSAR FEE 8.x Extension

With MICROSAR FEE 8.x a new feature named **Partitions** is introduced. A partition is an alignment of at least two logical sectors with the objective to assign data blocks to different, independent areas. A maximum of two partitions can be configured. Figure 5-1 shows the principle of FEE partitions:

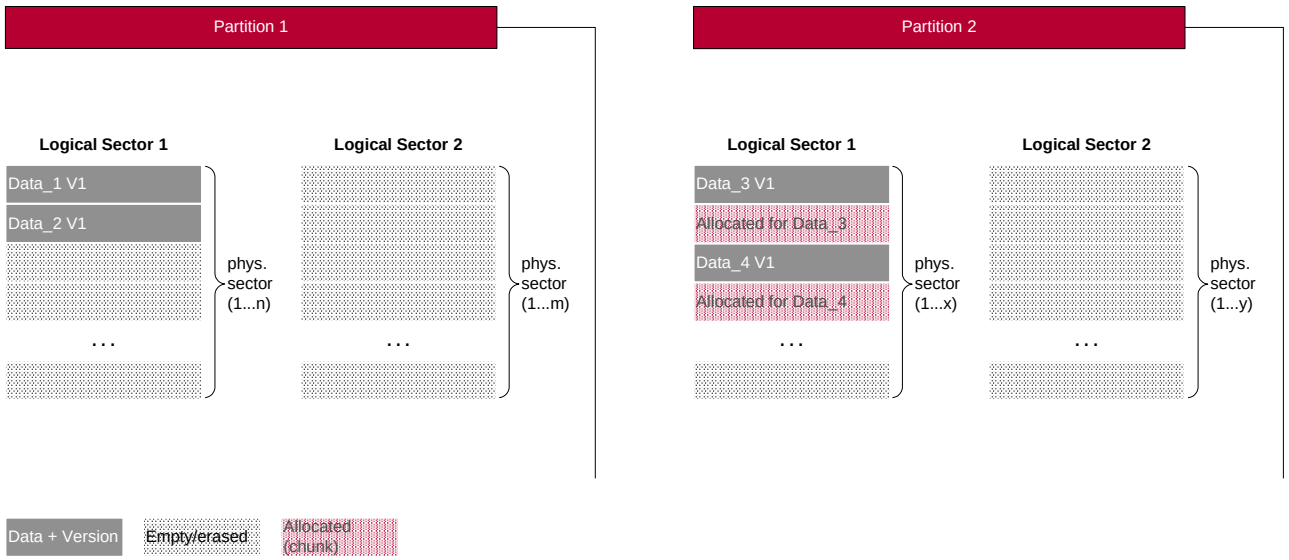

Figure 5-1 Principle of FEE partitions

A partition is like a “virtual” FEE instance, i.e. BSS or FSS runs in every partition separately. Therefore by assigning “constant” data to one partition and frequently written data to the other partition, copy overhead as well as search time for data blocks can be reduced, because:

- Less data blocks have to be copied during every sector switch, which also increases the probability that block copy can be skipped and “blocking time” of FEE reduces.
- No “constant” data has to be copied during normal operation, but during diagnostic session only.
- Smaller areas and fewer chunks have to be scanned to find next free location or latest data instance. This has significant impact on MCUs with large flashes.

Note
To use partitions, MCUs with four or more physical sectors are required. For MCUs with less than four physical sectors, only one partition can be configured, i.e. FEE has the same behavior like the latest version of FEE 6.x

Beside listed benefits of using partitions, some additional characteristics have to be considered:

- Due to smaller logical sectors, maximum data block size is limited.
- Write/erase cycles of flash memory may increase for the partition with frequently written data, because of the smaller logical sector size.
- Update capability is limited to the same partition, i.e. there is no possibility to resize partitions or to reassign data blocks between partitions.

Note
Regarding reset robustness, using partitions does not prevent from sector overflow. Nevertheless, a sector overflow only affects the related partition, whereas the other partition keeps operational. The APIs to suspend/resume FSS are global, i.e it affects both partitions when activated.

Compared to MICROSAR FEE 6.x, using partitions allows a smart way to manage constant and seldom written data within the FEE. For further configuration and integration hints, please also refer to chapter 4