MICROSAR FEE

Technical Reference

Version 8.02.03

Authors	Christian Kaiser
Status	Released

Document Information

History

Author	Date	Version	Remarks
Christian Kaiser	2012-06-26	8.00.00	- Removed revision history entries, due to major changes in component. - Removed all references to Fee30Inst2 - Added Ch. 2.4.2 “Partitions”, - Added Ch. 4.3.2 “Fee_InitEx”, updated Ch. 4.3.1 - Reworked Ch. 2.3, Ch. 2.6, Ch. 2.10, Ch. 3.1.1, 4.4.1, Ch. 5 - Changes throughout the document: introduction of partitions - Added Ch. 6.3.4 - Added Ch. 2.4.3.1
Christian Kaiser	2012-09-20	8.00.01	- Minor editorial changes in Ch. 5 - Added Ch. 5.2.1
Christian Kaiser	2013-03-05	8.00.02	- Ch. 1: AUTOSAR version(s) - Ch. 6.3.1: maximum number of Partitions
Christian Kaiser	2014-03-11	8.00.03	- Editorial changes (rework of review findings) - Ch. 5.1.5.5: corrected description of “Suspend Long” - Ch. 3.5: Critical Section description - Ch. 1.1 – updated figure, added notes.
Claudia Mausz	2015-02-13	8.01.00	- Add new chapter: 2.12 Fee_MainFunction Triggering
Christian Kaiser	2016-01-13	8.02.00	- Ch. 5.1.5.3: Corrected requirements on “Internal Buffer Size” configuration - Ch. 2.6.1.1: Updated “parameter checks” table - Ch.: 3.1.1: Updated deliverables table - Ch.: 4.3.2, 4.3.3, 4.3.11: Updated descriptions. - Ch. 5.1.5.1: Updated list of “Error detection switches”
Christian Kaiser	2016-07-27	8.02.01	- Ch. 6.3.16: Behavior of Fee_ForceSectorSwitch
Christian Kaiser	2016-08-31	8.02.02	- Ch. 4.3.6: internal behavior of Fee_Cancel
Christian Kaiser	2016-10-19	8.02.03	- Ch. 4.2.2: `FEE_SECTOR_CRITICAL_FILL_LEVEL` is obsolete - Ch 6.3.6: added restriction regarding allowed Flash page sizes - Ch. 6.3.7: added section regarding empty partitions and `Fee_ForceSectorSwitch`

Reference Documents

No.	Title	Version
[1]	AUTOSAR_SWS_Flash_EEPROM_Emulation.pdf	--
[2]	AUTOSAR_SWS_DET.pdf	V2.2.1
[3]	AUTOSAR_BasicSoftwareModules.pdf	V1.3.0

Please note
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

1 Introduction ... 10
1.1 Architecture Overview ... 11

2 Functional Description ... 13
2.1 Features ... 13
2.2 Initialization ... 14
2.3 States ... 15
2.3.1 Module States ... 15
2.3.2 Job States/Results ... 17
2.4 Flash organization ... 18
2.4.1 Block Handling ... 19
2.4.1.1 Block Chunks ... 19
2.4.1.2 Block Search ... 19
2.4.2 Partitions ... 19
2.4.3 Logical Sectors ... 20
2.5 Processing ... 21
2.5.1.1 Initial processing ... 21
2.5.1.2 Processing of Read Job ... 22
2.5.1.3 Processing of Write Job ... 22
2.5.1.4 Processing of InvalidateBlock Job ... 22
2.5.1.5 Processing of EraseImmediateBlock Job ... 22
2.5.1.6 Processing of GetEraseCycle Job ... 22
2.5.1.7 Processing of GetWriteCycle Job ... 23
2.6 Error Handling ... 23
2.6.1 Development Error Reporting ... 23
2.6.1.1 Parameter Checking ... 25
2.6.2 Production Code Error Reporting ... 26
2.6.3 Error notification ... 26
2.7 Sector Switch ... 27
2.7.1 Background Sector Switch (BSS) ... 27
2.7.2 Foreground Sector Switch (FSS) ... 27
2.7.3 Sector Overflow ... 28
2.7.4 Sector switch reserves and thresholds ... 28
2.7.5 Background Sector Switch Reserve/Threshold ... 29
2.7.6 Foreground Sector Switch/Threshold ... 29
2.8 Data Conversion ... 30
2.9 Flash Page Size impacts ... 31
2.10 Services for handling under-voltage situations ... 31

2.11 Critical Data Blocks ... 32
2.12 Fee_MainFunction Triggering ... 34

3 Integration ... 35
3.1 Scope of Delivery ... 35
3.1.1 Static Files ... 35
3.1.2 Dynamic Files ... 36
3.2 Compiler Abstraction and Memory Mapping ... 36
3.3 Dependencies on SW Modules ... 38
3.3.1 OSEK/AUTOSAR OS ... 38
3.3.2 Module SchM ... 38
3.3.3 Module Det ... 38
3.3.4 Module Fls ... 38
3.3.5 Callback Functions ... 39
3.3.5.1 Lower layer interaction ... 39
3.3.5.2 Upper layer interaction ... 39
3.3.5.3 User Error Callback ... 40
3.4 Dependencies on HW modules ... 41
3.5 Critical Sections ... 41

4 API Description ... 42
4.1 Interfaces Overview ... 42
4.2 Type Definitions ... 42
4.2.1 Fee_SectorSwitchStatusType ... 42
4.2.2 Fee_SectorErrorType ... 42
4.3 Services provided by FEE ... 44
4.3.1 Fee_Init ... 44
4.3.2 Fee_InitEx ... 45
4.3.3 Fee_SetMode ... 45
4.3.4 Fee_Read ... 47
4.3.5 Fee_Write ... 48
4.3.6 Fee_Cancel ... 49
4.3.7 Fee_GetStatus ... 50
4.3.8 Fee_GetJobResult ... 50
4.3.9 Fee_InvalidateBlock ... 51
4.3.10 Fee_GetVersionInfo ... 52
4.3.11 Fee_EraseImmediateBlock ... 52
4.3.12 Fee_MainFunction ... 53
4.3.13 Fee_GetEraseCycle ... 54
4.3.14 Fee_GetWriteCycle ... 55
4.3.15 Fee_GetSectorSwitchStatus ... 56

4.3.16 Fee_ForceSectorSwitch ... 58
4.3.17 Fee_ConvertBlockConfig ... 59
4.3.18 Fee_SuspendWrites ... 61
4.3.19 Fee_ResumeWrites ... 61
4.3.20 Fee_DisableFss ... 62
4.3.21 Fee_EnableFss ... 63
4.4 Services used by FEE ... 63
4.4.1 Data Conversion Callback ... 64
4.5 Callback Functions ... 65
4.5.1 Fee_JobEndNotification ... 65
4.5.2 Fee_JobErrorNotification ... 65
4.6 Configurable Interfaces ... 66

5 Configuration ... 67
5.1 Configuration with DaVinci Configurator ... 67
5.1.1 Start configuration of the FEE ... 67
5.1.2 Useful Chunk-Sizes (instance counts) ... 67
5.1.3 Update of block configuration ... 69
5.1.4 FEE Configuration tab ... 69
5.1.5 General Settings tab ... 73
5.1.5.1 Error Detection – Development Mode ... 73
5.1.5.2 Area “Error Callback” ... 73
5.1.5.3 Area Buffer ... 74
5.1.5.4 Area “Upper Layer” ... 74
5.1.5.5 Area “Critical Section Handling” ... 74
5.1.6 Partitions ... 75
5.1.6.1 Area “Management” ... 77
5.1.6.2 Area “Lower Layer” ... 77
5.1.7 Module API tab ... 78
5.1.7.1 API Configuration ... 78
5.1.7.2 Provided API ... 79
5.2 Configuration Parameters only visible in GCE ... 79
5.2.1 Fls API deviating from AUTOSAR naming convention ... 79

6 AUTOSAR Standard Compliance ... 81
6.1 Deviations ... 81
6.1.1 Maximum Blocking Time ... 81
6.2 Additions/ Extensions ... 81
6.2.1 Parameter Checking ... 81
6.2.2 Fee_InitEx ... 81
6.2.3 GetEraseCycle ... 81

6.2.4 GetWriteCycle ... 81
6.2.5 GetSectorSwitchStatus ... 81
6.2.6 ForceSectorSwitch ... 81
6.2.7 Fee_ConvertBlockConfig ... 81
6.2.8 Fee_SuspendWrites / Fee_ResumeWrites ... 81
6.2.9 Fee_EnableFss / Fee_DisableFss ... 81
6.3 Limitations ... 81
6.3.1 Partitions ... 81
6.3.2 Flash Usage ... 82
6.3.3 Performance ... 82
6.3.4 Aborts/Resets ... 82
6.3.5 Write Cycle and Erase Cycle Counters ... 83
6.3.6 Fls Page sizes ... 83
6.3.7 Fee_ForceSectorSwitch and unformatted Partitions ... 83

7 Glossary and Abbreviations ... 84
7.1 Glossary ... 84
7.2 Abbreviations ... 84

8 Contact ... 86

Illustrations

1-1 AUTOSAR architecture ... 11
Figure 1-2 FEE in a typical (AUTOSAR) SW architecture ... 12
Figure 2-1 FEE Module States ... 16
Figure 2-2 From User Blocks to Flash Sectors ... 18

Tables

Table 2-1 Supported SWS features ... 13
Table 2-2 Not supported SWS features ... 13
Table 2-3 Module states ... 15
Table 2-4 Job results ... 17
Table 2-5 Mapping of service IDs to services ... 24
Table 2-6 Errors reported to DET ... 24
Table 2-7 Development Error detection: Assignment of checks to services ... 26
Table 2-8 Det_ReportError ... 27
Table 3-1 Static files ... 35
Table 3-2 Generated files ... 36
Table 3-3 Compiler abstraction and memory mapping ... 36
Table 3-4 Error Codes and FEE's default behavior ... 41
Table 4-1 Fee_SectorSwitchStatusType ... 42
Table 4-2 Fee_SectorError Type ... 43
Table 4-3 Fee_Init ... 44
Table 4-4 Fee_InitEx ... 45
Table 4-5 Fee_SetMode ... 46
Table 4-6 Fee_Read ... 47
Table 4-7 Fee_Write ... 48
Table 4-8 Fee_Cancel ... 49
Table 4-9 Fee_GetStatus ... 50
Table 4-10 Fee_GetJobResult ... 51
Table 4-11 Fee_InvalidateBlock ... 51
Table 4-12 Fee_GetVersionInfo ... 52
Table 4-13 Fee_EraseImmediateBlock ... 53
Table 4-14 Fee_MainFunction ... 54
Table 4-15 Fee_GetEraseCycle ... 55
Table 4-16 Fee_GetWriteCycle ... 56
Table 4-17 Fee_GetSectorSwitchStatus ... 57
Table 4-18 Fee_ForceSectorSwitch ... 59
Table 4-19 Fee_ConvertBlockConfig ... 61
Table 4-20 Fee_SuspendWrites ... 61
Table 4-21 Fee_ResumeWrites ... 62
Table 4-22 Fee_DisableFss ... 62
Table 4-23 Fee_EnableFss ... 63
Table 4-24 Services used by the FEE ... 64
Table 4-25 User defined conversion callback ... 65
Table 4-26 Fee_JobEndNotification ... 65
Table 4-27 Fee_JobErrorNotification ... 66
Table 4-28 Configurable interfaces ... 66
Table 5-1 Fee configuration ... 72
Table 5-2 Error Detection – Development Mode ... 73
Table 5-3 Error Callback ... 74
Table 5-4 Upper Layer ... 74

Table 5-5 Critical Section Services 75
Table 5-6 Lower Layer 76
Table 5-7 sector switch reserve 77
Table 5-8 Lower Layer 77
Table 5-9 API Configuration 78
Table 5-10 Provided API 79
Table 5-11 Parameters only visible in GCE view. 79
Table 7-1 Glossary 84
Table 7-2 Abbreviations 85

1 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module FEE as specified in [1].

Supported AUTOSAR Release*:	3 and 4	
Supported Configuration Variants:	link-time with AUTOSAR 3 pre-compile with AUTOSAR 4	
Vendor ID:	FEE_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	FEE_MODULE_ID	21 decimal (according to ref. [3])

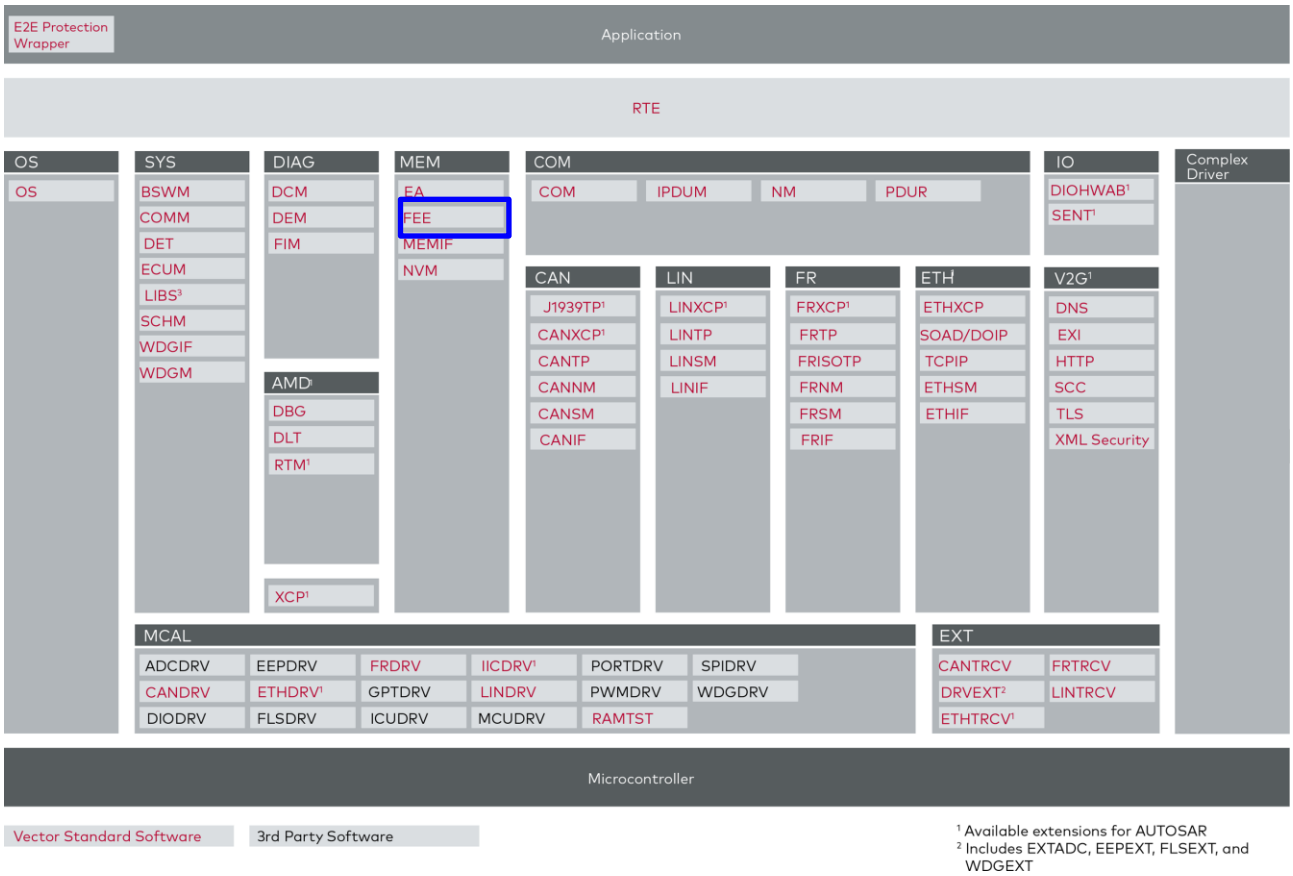
* For the precise AUTOSAR Release please see the release specific documentation.

The FEE enables you to access a dedicated flash area for storing data persistently. It is intended to be used exclusively either by the NVRAM Manager or on SW instance within a Flash-Boot-Loader in the Boot-Loader mode.

Further on, the module depends on some other modules, like DET for error handling, the underlying Flash driver for hardware access and the MEMIF for consistent types.

1.1 Architecture Overview

The following figure shows where the FEE is located in the AUTOSAR architecture.



1-1 AUTOSAR architecture

The following figure shows the FEE and its relationship to other modules.

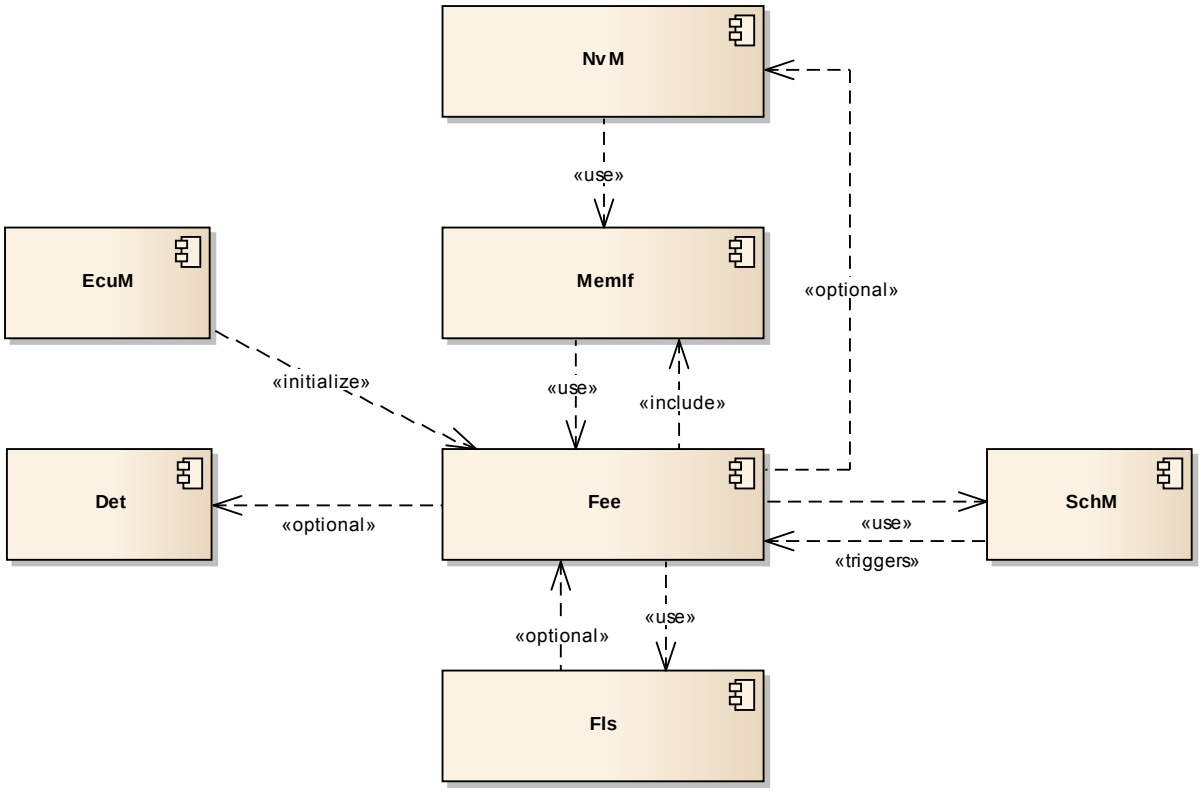


Figure 1-2 FEE in a typical (AUTOSAR) SW architecture

Note
Figure 1-2 shows the normal case. In general FEE does not depend on who actually uses it, and who provides required services. E.g. it does not matter who provides mechanisms to synchronize access to critical sections (chapter 3.5), or who actually calls `Fee_MainFunction`. It also doesn't matter whether callbacks (Fls to FEE, FEE to NvM) are directly called as depicted in the figure, or whether they are "intercepted" by someone.

Caution
FEE assumes exclusive usage of Fls. Allowing other components to use Fls's services results in synchronization issues, which are hard to solve. Hence this shall be avoided.

2 Functional Description

2.1 Features

The features listed in this chapter cover the complete functionality specified in [1].

The "supported" and "not supported" features are presented in the following two tables. For further information of not supported features also see chapter 6.

The following features described in [1] are supported:

Feature
The module operates on blocks provided by the NVRAM Manager. Read accesses to blocks are handled byte-wise.
Hardware restrictions like erase cycles or sector sizes are abstracted and not visible for the upper layer.
Incomplete writes (e.g. due to reset) are detected.
The virtual sectors use a wrap-around concept with backup of the most recent data blocks.
Possibility to retrieve the number of erase cycles of a logical sector (done via the API service `Fee_GetEraseCycle()`). This feature is an add-on to the AUTOSAR standard.
Possibility to retrieve the number of write cycles of a block (done via the API service `Fee_GetWriteCycle()`).This feature is an add-on to the AUTOSAR standard.
Possibility to force sector switches (done via the API service `Fee_ForceSectorSwitch()`).This feature is an add-on to the AUTOSAR standard.
API to perform data conversion after configuration update i.e. blocks whose payload has changed may be converted according to new configuration without data-loss. This feature is an add-on to the AUTOSAR standard. **Info** This feature is optional, i.e. it has to be ordered explicitly.
Fee supports Flash Address Spaces (provided by Fls) of up to 2GBytes, i.e. Sectors to be used by FEE may resist in range `0x00000000` to `0x7FFFFFFF`
`Fee_MainFunction` Triggering: Possibility to call the `Fee_MainFunction` in a cyclic task or in a background task.

Table 2-1 Supported SWS features

The following features described in [1] are not supported:

Feature
The `MAXIMUM_BLOCK_TIME` is not supported by the FEE, because no time reference is provided to the FEE.

Table 2-2 Not supported SWS features

2.2 Initialization

The FEE is initialized and operational after `Fee_Init()` or `Fee_InitEx()` has been called.

Caution
The FEE is driven asynchronously, i.e. jobs are requested via dedicated API, and they are processed by calling `Fee_MainFunction()`.

Initialization just prepares FEE to accept and process requests. Initialization of sectors (checking their headers and determining new/old one) is deferred until first request is being started on a partition.

Additionally after initialization, any sector switch processing is disabled:

- Sector Switch Processing in Background (see 2.7.1) can be enabled by `Fee_SetMode(MEMIF_MODE_SLOW)`.
- Depending on “FSS Control API” (see Ch. 5.1.7.1) they can only be enabled by:
  - `Fee_EnableFss()`, if “FSS Control API” was enabled. In that case write requests beyond Foreground Sector Switch Threshold are disabled; refer to chapter 2.10.
  - `Fee_Write()` / `Fee_InvalidateBlock()` / `Fee_EraseImmediateBlock()`, otherwise.

These implicit defaults help to ensure that FEE does not perform any write, or even erase operations during ECU start-up, unless explicitly requested by user.

Caution
It is not recommended to use any of the mentioned services during ECU’s start-up phase. Special care needs to be taken about write requests: Proper initialization (order) of application software is important to prevent from too early write accesses to NV memory.

2.3 States

2.3.1 Module States

Point in Time	Module State
After Reset, before Fee_Init was called	`MEMIF_UNINIT`
When Fee_Init() returns	`MEMIF_ BUSY_INTERNAL`
When accepting a job request.	`MEMIF_BUSY`
If rejecting a job request.	No change.
Upon completing a job (including returning from `Fee_SetMode()`)	`MEMIF_BUSY_INTERNAL`
Pending internal operations (while no user job is pending)	`MEMIF_BUSY_INTERNAL`
Not even an internal job is pending	`MEMIF_IDLE`

Table 2-3 Module states

Note
State MEMIF_UNINIT can only be delivered after reset, if start-up code was executed, and section `FEE_START_SEC_VAR_INIT_UNSPECIFIED` was mapped to an appropriate section, being initialized by startup code. See also: chapter 3.2

Note
Normally (unless Fee_Cancel was called) FEE_IDLE state can only be entered via FEE_BUSY_INTERNAL.

Whenever FEE leaves MEMIF_BUSY_STATE due to job completion it asynchronously checks all partitions for their fill levels, and the necessity of sector switch. Therefore, even if no sector switch is necessary or allowed, it takes few `Fee_MainFunction` cycles to finally become (and report) IDLE

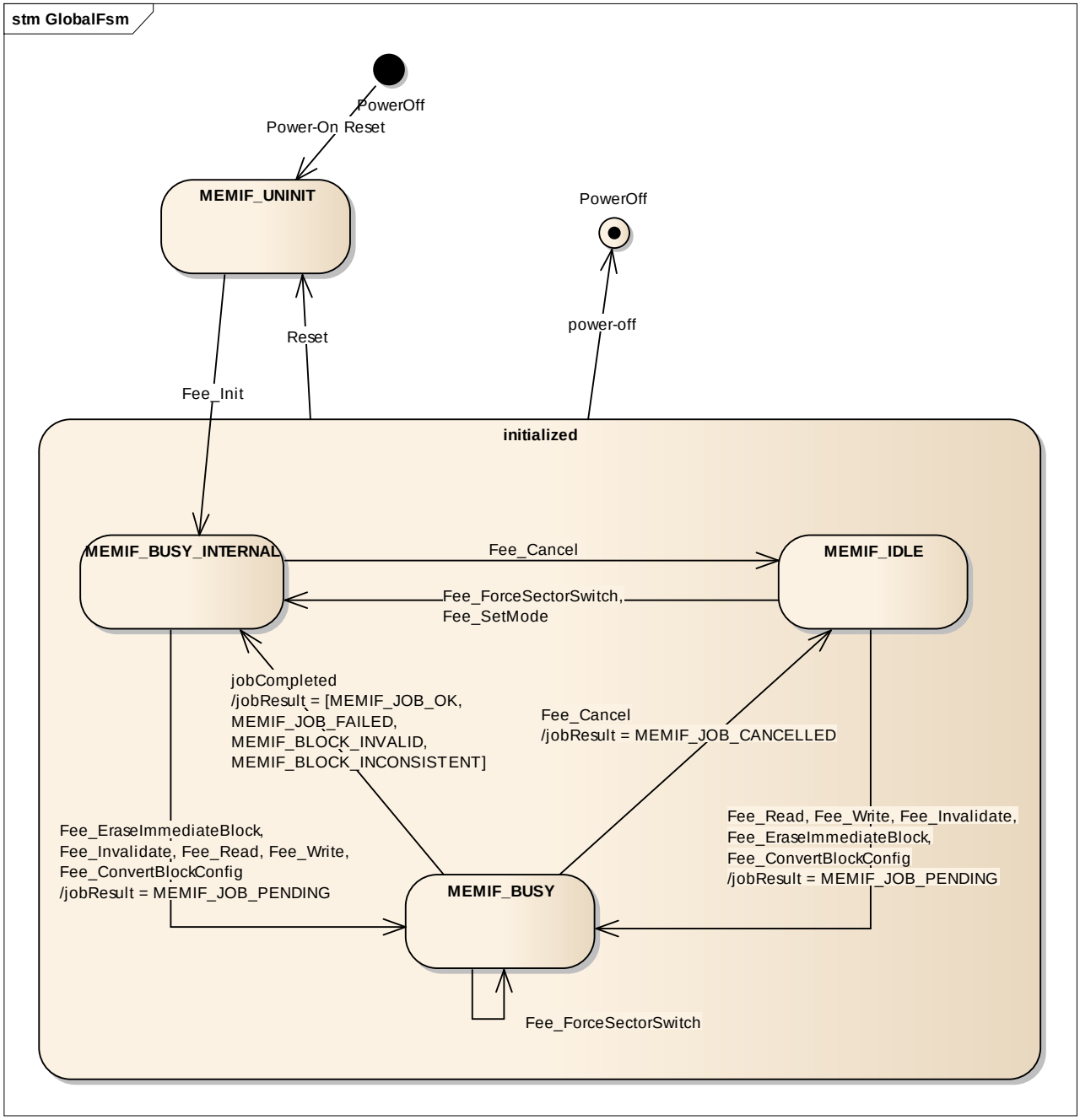


Figure 2-1 FEE Module States

2.3.2 Job States/Results

Point in Time	Job State
After successfully finished job	`MEMIF_JOB_OK`
After a job has been accepted	`MEMIF_JOB_PENDING`
After `Fee_Cancel()`	`MEMIF_JOB_CANCELLED`
After a read job has been finished and an invalidated block was found, or the requested block was never successfully written before.	`MEMIF_BLOCK_INVALID`
After a block is read which has not been written successfully before.	`MEMIF_BLOCK_INCONSISTENT`
After a job has been finished and retrieving data (independent from management information of payload data) from the Flash failed	`MEMIF_JOB_FAILED`

Table 2-4 Job results

2.4 Flash organization

Figure 2-2 gives an overview of how FEE organizes flash memory, and how FEE's three main concepts – (User) Blocks, Partitions and Logical Sectors – are related.

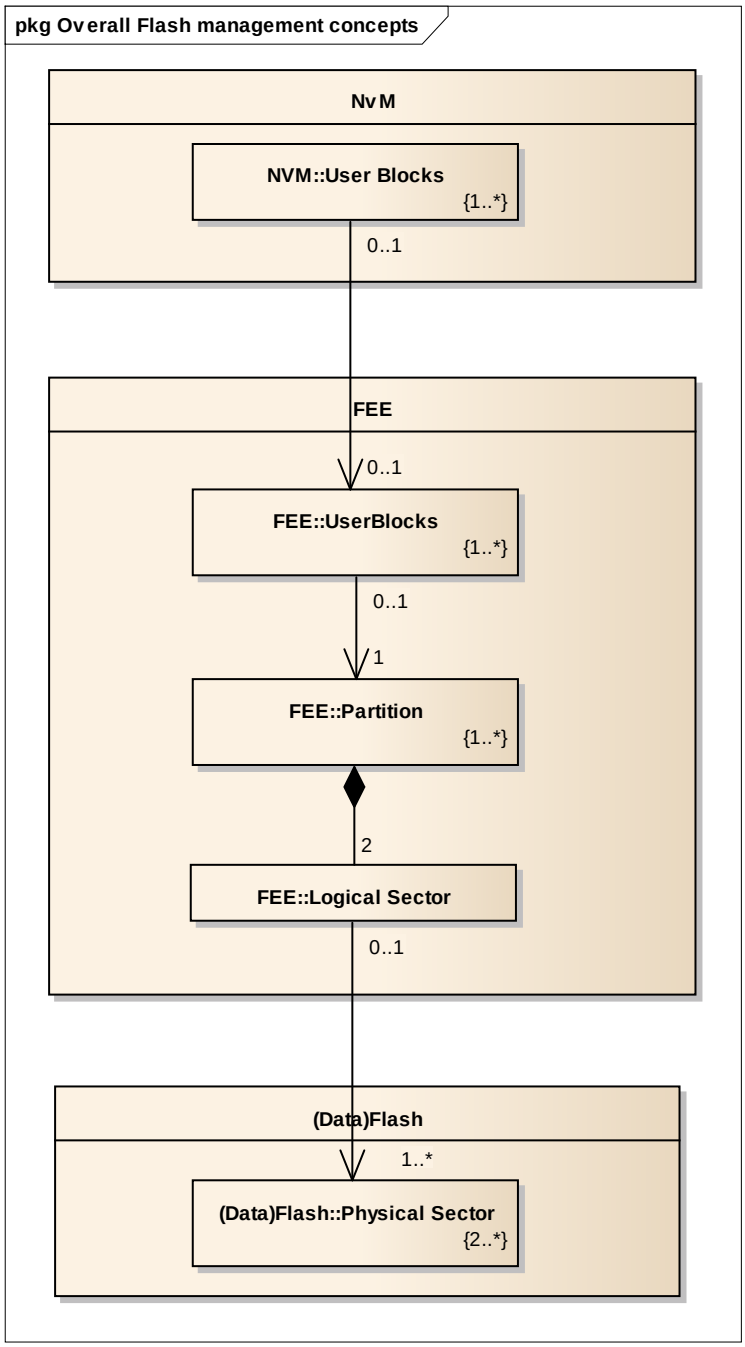


Figure 2-2 From User Blocks to Flash Sectors

2.4.1 Block Handling

2.4.1.1 Block Chunks

Block chunks are the smallest entities the FEE can allocate dynamically in Flash. A chunk allocation reserves space for a configurable number (refer to chapter 5.1.4) of versions of the associated block. FEE continues writing new versions (aka instances) of this block into its most recent chunk, until it's full. Writing the next instance requires allocation of a new chunk. Each block's chunks build up a linked list in flash: each chunk points to its successor.

2.4.1.2 Block Search

A requested block is searched by following the links provided by the block chunks. The start point is the default area of the block. If the chunk link is erased the search is continued within the current chunk using a binary search algorithm.

The search concept provides fast access to every block as only the needed block chunks are accessed.

2.4.2 Partitions

FEE employs a concept of partitions. A partition can be thought of an emulation space that is managed separately from other ones:

> Errors in one partition do not affect data in other ones

> Error states (e.g. Read-Only Mode) are local to a single partition.

> Job processing in one partition does (almost) not depend on sector switch processing (chapter 2.7) in other partitions.

However, compared to two FEEs operating on two different flash devices, partitions still require synchronization:

> FEE is still restricted to use one single flash driver.

> According to AUTOSAR, FEE is still limited to one pending user request.

> Only one partition can perform a sector switch at a time

> Sector switch processing can only be interrupted when processing a block completed.

Each partition consists of two logical sectors, which in turn are mapped to physical flash sectors.

During configuration, each logical block must be assigned to a partition.

Note
FEE configurations for FBL and Application do not need to share all partitions. E.g. a partition containing only application data may remain unknown to the FBL. However, shared partitions must refer to identical Fls configurations (FlsConfigSet container), and they must match in address and size, as well as in alignment settings.

Example
Most typical as well as most recommended use case for partitions is the physical separation of frequently and infrequently written data. Such a separation reduces sector switch efforts, because a whole partition would be infrequently updated. Therefore, separation also increases availability data. However, it may increase number of flash's erase cycles over life-time, because frequently written data will usually (it depends on actual amounts) be spread over less flash memory.

2.4.3 Logical Sectors

The FEE divides the physical flash assigned to a partition in two parts, called logical sectors. Each logical sector must be mapped to one or more continuous physical Flash sectors. Logical sectors do not need to be aligned on physical sector boundaries, i.e. they may start and/or end within physical sectors. However, start and end addresses must adhere to the configured address alignment.

Note
The mapping of logical to physical sectors is mutual exclusive, i.e. physical sectors may not be shared between logical sectors.

It results in a minimum number of physical sectors that must be available to define (additional) partitions: Per partition, two physical sectors must be provided by HW (and Fls).

Caution
Unused flash space within a physical sector must not be used (neither read nor write) by other software. It should be assumed to be erased, because FEE does never write it, but it will be erased, too, with each sector erase triggered by FEE.

Besides the lower logical sector must be located at a smaller address than the upper one, both sectors may be located anywhere within the flash space provided by the underlying driver (considering restrictions imposed by hardware or the driver, of course), i.e. there may be a gap between them.

Furthermore, the two logical sectors must be erasable independently from each other, i.e. they must be mapped to distinctive physical sectors.

The size of the smaller logical sector defines the maximal number of blocks that can be handled by the FEE.

Info
Within AUTOSAR specification logical sectors are called virtual sectors.

2.5 Processing

All jobs (Read, Write, InvalidateBlock, EraseImmediateBlock, GetWriteCycle, GetEraseCycle, ForceSectorSwitch) will be executed asynchronously with the help of a job state machine.

Caution
The FEE must be initialized before the services are called.

Caution
Only one job can be accepted at a time. Hence, it is not allowed to request an asynchronous job to the FEE as long as the currently pending job has not been completed.

While internal operations are performed (current status is equal to `MEMIF_BUSY_INTERNAL`) a job can be accepted, but actual processing may be deferred due to sector switch handling (see chapter 2.7 for more details).

2.5.1.1 Initial processing

In order to become able to process a request, FEE has to determine both logical sectors' states, i.e. whether they are usable at all, and which one contains most recent data. Once FEE determined these states, it maintains them in RAM.

Since a request is always associated with a partition, FEE performs this initialization step at beginning of normal job processing.

Expert Knowledge
FEE attempts to read both sector headers, i.e. the first 8 bytes in each logical sector. This processing takes at least 3 cycles (from initial job request to start of actual job processing), assuming FEE does not have to wait for Fls, i.e. Fls_Read processing completes between two Fee_MainFunction calls.

Info
If one of both Fee sector headers is erased or corrupted, and the other one is ok, FEE tries to use the valid sector; erasing the other one will be done either when necessary, or when no user jobs are pending (see also chapter 2.7)

If both sector headers are erased, FEE behaves as if flash is nearly full (see also 2.7.2): It will erase one logical sector and write its header, as part of first write class job's processing.

If both sector headers are corrupted or one header is corrupted and the other is erased, FEE calls the user error callback function, if configured. Refer to chapter 3.3.5.3

2.5.1.2 Processing of Read Job

The FEE provides the service `Fee_Read()` for reading a block. This service reads the data of the block which has been most recently written.

This asynchronous job is initiated with the API function `Fee_Read()` and is processed by subsequent calls of `Fee_MainFunction()` (see also chapter 2.3).

2.5.1.3 Processing of Write Job

To write the current block content to flash memory the API function `Fee_Write()` is used. FEE searches the next free position in the most recent block chunk to write block's new instance to.

This asynchronous job is initiated with the API function `Fee_Write()` and is processed by subsequent calls of `Fee_MainFunction()` (see also chapter 2.3).

2.5.1.4 Processing of InvalidateBlock Job

To invalidate the block content in flash memory the API function `Fee_InvalidateBlock()` is used. The FEE component marks the block as invalid; upon success, subsequent read attempts report `MEMIF_BLOCK_INVALID`.

Expert Knowledge
Block Invalidation is very similar to Block Write, as it also creates a new instance, i.e. it consumes flash memory.

This asynchronous job is initiated with the API function `Fee_InvalidateBlock()` and is processed by subsequent calls of `Fee_MainFunction()` (see also chapter 2.3).

2.5.1.5 Processing of EraseImmediateBlock Job

Immediate data is data of a block which should be written with a higher priority than the other blocks.

To mark an immediate block as erased the API function `Fee_EraseImmediateBlock()` is used. As the FEE component can't erase the corresponding block it writes invalid information for the block to the flash memory.

Expert Knowledge
FEE processes EraseImmediateBlock jobs identically to InvalidateBlock jobs.

This asynchronous job is initiated with the API function `Fee_EraseImmediateBlock()` and is processed by subsequent calls of `Fee_MainFunction()` (see also chapter 2.3).

2.5.1.6 Processing of GetEraseCycle Job

The erase cycle counter is increased during every erase of a certain logical sector and consequently counts the erase cycles of the flash.

To get the erase cycle counter of a specified logical sector, the API function `Fee_GetEraseCycle()` is used. The availability of this service is configurable at pre-compile time which can be done via the configuration tool.

This asynchronous job is initiated with the API function `Fee_GetEraseCycle()` and is processed by subsequent calls of `Fee_MainFunction()` (see also chapter 2.3).

2.5.1.7 Processing of GetWriteCycle Job

The write cycle counter counts the write cycles of each block.

To get the write cycle counter of a specified block, the API function `Fee_GetWriteCycle()` is used. The availability of this service is configurable at pre-compile time which can be done via the configuration tool.

This asynchronous job is initiated with the API function `Fee_GetWriteCycle()` and is processed by subsequent calls of `Fee_MainFunction()` (see also chapter 2.3).

2.6 Error Handling

The module offers detection of errors.

Errors are classified in development and production errors.

Development errors should be detected and fixed during development/integration phase. Those errors are caused by faulty configuration or incorrect usage of the module's API.

Production errors are hardware errors and software exceptions that cannot be avoided and are also expected to occur in production code. FEE has no case to report a production error.

2.6.1 Development Error Reporting

Development errors are reported to DET using the service `Det_ReportError()` as specified in [2], if development error detection and reporting, are enabled (see chapter 5.1.5.1).

If another module than DET is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()` (see chapter 2.6.3).

The reported FEE ID can be found in chapter 1.

The reported service IDs identify the services which are described in 4.3. The following table presents the service IDs and the related services:

Service ID	Service
0x00u	Fee_Init() / Fee_InitEx()
0x01u	Fee_SetMode()
0x02u	Fee_Read()
0x03u	Fee_Write()
0x04u	Fee_Cancel()
0x05u	Fee_GetStatus()
0x06u	Fee_GetJobResult()
0x07u	Fee_InvalidateBlock()

Service ID	Service
0x08u	Fee_GetVersionInfo()
0x09u	Fee_EraseImmediateBlock()
0x10u	Fee_JobEndNotification()
0x11u	Fee_JobErrorNotification()
0x12u	Fee_MainFunction()
0x20u	Fee_GetEraseCycle()
0x21u	Fee_GetWriteCycle()
0x22u	Fee_GetSectorSwitchStatus()
0x23u	Fee_ForceSectorSwitch()
0x24u	Fee_ConvertBlockConfig()

Table 2-5 Mapping of service IDs to services

The errors reported to DET are described in the following table:

Error Code		Description
0x02	FEE_E_INVALID_BLOCK_NO	This error code is reported if an API service is called with invalid block number.
0x10	FEE_E_PARAM_DATABUFFERPTR	It is reported if a pointer parameter of an API service is called with the NULL_PTR value.
0x11	FEE_E_PARAM_SECTOR_NUMBER	It is reported if the Fee_GetEraseCycle() API service is called with an invalid logical sector number.
0x12	FEE_E_PARAM_LENGTH_OFFSET	It is reported if the Fee_Read() API service is called with invalid BlockOffset and Length values.
0x13	FEE_E_BUSY	It is reported if one of the asynchronous API services has been called in parallel.
0x14	FEE_E_NO_INIT	It is reported if one of the API services has been called without an initialized FEE.

Table 2-6 Errors reported to DET

Expert Knowledge
All error codes starting from 0x10 are defined in addition to [1].

2.6.1.1 Parameter Checking

AUTOSAR requires that API functions check the validity of their parameters. The checks listed in Table 2-7 are internal parameter checks of the API functions. These checks are intended for development error detection and can be en-/disabled separately; it is described in chapter 5.1.5.

Capability of controlling execution of single checks is an addition to AUTOSAR which just requires to en-/disable the complete parameter checking via the parameter `FEE_DEV_ERROR_DETECT`.

The following table shows which parameter checks are performed on which services:

Service \ Check	FEE_E_INVALID_BLOCK_NO	FEE_E_PARAM_DATABUFFERPTR	FEE_E_PARAM_SECTOR_NUMBER	FEE_E_PARAM_LENGTH_OFFSET	FEE_E_BUSY	FEE_E_NO_INIT
Fee_Init()						
Fee_InitEx()		■				
Fee_SetMode()						■
Fee_Read()	■	■		■	■	■
Fee_Write()	■	■			■	■
Fee_Cancel()						■
Fee_GetStatus()						
Fee_GetJobResult()						■
Fee_InvalidateBlock()	■				■	■
Fee_GetVersionInfo()		■				
Fee_EraseImmediateBlock()	■				■	■
Fee_JobEndNotification()						
Fee_JobErrorNotification()						
Fee_MainFunction()						■[1]
Fee_GetEraseCycle()		■	■		■	■
Fee_GetWriteCycle()	■	■			■	■
Fee_GetSectorSwitchStatus()						■
Fee_ForceSectorSwitch()						■

[1] Check only; no error reporting to DET

Service \ Check	FEE_E_INVALID_BLOCK_NO	FEE_E_PARAM_DATABUFFERPTR	FEE_E_PARAM_SECTOR_NUMBER	FEE_E_PARAM_LENGTH_OFFSET	FEE_E_BUSY	FEE_E_NO_INIT
`Fee_ConvertBlockConfig()`		■			■	■
`Fee_SuspendWrites()`						■
`Fee_ResumeWrites()`						■
`Fee_EnableFss()`						■
`Fee_DisableFss()`						■

Table 2-7 Development Error detection: Assignment of checks to services

2.6.2 Production Code Error Reporting

FEE does not report any production related error to an error/event manager like DEM.

2.6.3 Error notification

All detected errors in development mode are by default reported to the Development Error Tracer (DET), but can be configured regarding called function and include file.

The error declaration must have following syntax:

Prototype	
Prototype syntax described in chapter 8.2.2 Det_ReportError of [2].	
Parameter	
ModuleId	Specifies the identifier of the module causing the error. The module Id of FEE can be found in chapter 1.
InstanceId	The identifier of the index based instance of a module, starting from 0. Thus the FEE is a single instance module it will pass 0 as InstanceId.
ApiId	The identifier of the API function, which caused the error.
ErrorCode	The number of the specific error.
Return code	
void	--
Functional Description	
User function for the `Fee_Errorhook`, which specifies development errors.	

Particularities and Limitations
> This service is synchronous. > This service is non re-entrant. > This service is always available.
Expected Caller Context
> called in application context

Table 2-8 Det_ReportError

2.7 Sector Switch

The sector switch is responsible to gather most recent instances data of all blocks/datasets from one logical sector (= source sector) and write them to the other one (= target sector). After this procedure has been finished, the source sector can be erased to prepare it for storing data again.

Expert Knowledge
Under certain conditions, no copy is necessary at all. However, finally FEE has to erase a sector to reclaim free flash space.

There are different reasons to perform a sector switch

- Threshold exceeded (see below)
- User Request (Fee_ForceSectorSwitch)
- Critical Block Handling (refer to ch. 2.11)

2.7.1 Background Sector Switch (BSS)

Sector Switches will be processed in background, unless FEE was set to “Fast Mode” (refer to ch. 4.3.3). This means that processing occurs only if no user job is pending.

Sector Switch processing is interruptible by user jobs after a block has been processed (copied, or decision to skip copying was taken).

2.7.2 Foreground Sector Switch (FSS)

A Sector Switch being processed in Foreground defers a write job. That means a job cannot be processed before a sector switch has been completed.

Such a sector switch is not interruptible by any other job.

Expert Knowledge
The actual reason is the pending job that was deferred. At most one job may be pending at any time.

FSS processing is always initiated to a pending write job for a particular block. This block is associated with a specific partition; thus a FSS will be performed on that partition only.

2.7.3 Sector Overflow

A sector overflow occurs, if FEE failed to complete data copy, because both logical sectors ran out of space. Since in that case both sectors contain most recent data instances (of different blocks), FEE is unable to continue with erase operation without losing most recent data.

Such an event is caused by several consecutive aborted write attempts.

Caution
Such aborts should be avoided wherever possible, i.e. they shall be considerable being exceptional (thus rare) events. Especially a SW reset shall be performed only if FEE is IDLE.

A sector overflow is local to one particular partition, i.e. other partitions are not affected. Probability of an overflow depends on different parameters:

- Probability of resets, of course
- Partition's write load, which can be defined as number of bytes over time.
- (Foreground) Sector Switch in relation to blocks' sizes.

To minimize effects caused by under-voltage situations FEE provides additional services; see chapter 2.10.

Accepting loss of most recent data instances, FEE may be instructed to perform a sector erase in order to restore write ability; see chapter 3.3.5.3. Additionally the risk of losing data can be restricted to certain "uncritical" blocks at configuration time. For details refer to chapter 2.11.

2.7.4 Sector switch reserves and thresholds

The FEE determines the necessity of a sector switch based upon exceeding a related threshold value. This value is always an offset relative to the start address of the currently "newer" sector.

Basic Knowledge
A threshold is considered to have been exceeded, if one sector (the older one) **is** full and the other one (newer sector) is filled up to (or exceeding) this certain level.

Note
To assure that a sector switch in FBL mode can copy FBL's data and an unknown amount of application data, sector switch has to start as early as possible. Therefore in FBL mode a sector switch starts once the newer logical sector is used. This might degrade efficiency of flash usage, but since it would happen in FBL mode only, effects are negligible.

In configuration "reserve values", rather than the thresholds, need to be specified by user. During generation the thresholds to be used by FEE are calculated based on these values and block configuration. "Reserve" is meant as follows: How much space the FEE shall reserve additionally to the space complete block configuration would consume in flash (that is the amount of flash space that would be consumed, when every block was written exactly once).

Thus a reserve value of 0 means: For every configured block, one chunk is reserved; FEE shall start sector switch once less than one chunk per block fits into flash.

A reserve greater than 0 adds additional space resulting in more possible aborts until the flash runs full (sector overflow).

The worst case number of additional resets the FEE can deal with is the FSS reserve divided by the size of the largest chunk.

There is no standard recommendation how to configure this threshold, because this value depends on the individual environment of the ECU and the configuration of the memory stack, like:

- total available flash memory
- number of blocks/datasets
- size of the blocks/datasets
- page size of the flash hardware
- general handling of blocks, i.e. when blocks/datasets are written down to the non-volatile memory

2.7.5 Background Sector Switch Reserve/Threshold

A sector switch in background mode (BSS) is the type of sector switch that should normally occur. In flash, both sectors are currently in use, but there is quite much space available.

Exceeding BSS threshold causes FEE to start BSS processing, as described above.

2.7.6 Foreground Sector Switch/Threshold

If flash/partition fill level exceeds FSS threshold, it is quite full. It can be seen as "last chance processing". Therefore, a write job directed to that partition will be deferred in order to complete FSS first.

FSS threshold also indicates most critical fill-level: once it is exceeded, a call to `Fee_DisableFss()` becomes effective, i.e. in that case FEE terminates write requests delivering error result.

Since BSS is defined to start earlier than FSS, its threshold is smaller. This also means that an FSS implies BSS, i.e. processing is done in Background, as described above.

Expert Knowledge
FEE keeps track of logical sector's state. Once copying most recent data completed, it internally marks the "older" sector as outdated. This results in reducing criticality: Flash is still almost full, but we just need to erase a sector. An FSS becomes rather a BSS; after completing the copy process, a pending write job gets the chance to be processed before sector format is being started. If Flash is too full, the job will be deferred again; it has to wait for completion of sector format.

2.8 Data Conversion

Info
This feature is optional, i.e. it has to be ordered explicitly.

Standard FEE implementation allows to update the block configuration, and to modify block lengths with one restriction: Existing data of blocks, whose payload (length) was changed, will be lost. Trying to read such a block will lead to job result `MEMIF_BLOCK_INVALID`, until the block has been re-written according to new configuration.

Data Conversion provides a "framework" enabling the SW to keep even those blocks: The FEE scans Dataflash content. For each block (in detail: each block's most recent instance) it finds, a callback function will be invoked. This function gets following arguments:

> Unique block identifier (each dataset is treated separately)

> (pointer to) most recent data

> old length (as found in flash)

> new length (according to current configuration), if block is still configured.

Within the callback the user may decide what the FEE shall do with this block:

> skip it (it will be lost after finalization of Conversion)

> write it according to old length

> write it according to new length

The data passed to the callback may be modified; the FEE will write them, if desired. This is the actual conversion. Since the FEE is not able to detect internal configuration changes, the callback will also be invoked for blocks whose length did not change.

Info
The callback might need to implement some code to make correct decisions, e.g. if different blocks need to be treated differently, additionally depending on old configuration (from which version the update to current version has been performed)

If a block does not contain any data, i.e. it was never (successfully) written, it had been invalidated, or it became corrupted by other means, the callback will not be invoked; there is no data to be converted.

Caution
Conversion must be triggered immediately after start-up, using `Fee_ConvertBlockConfig` (refer to 4.3.17), before upper layers get running, and before a sector switch (with complete configuration) has been performed. Therefore it must happen before the first write request (including invalidate and erase) was issued to the FEE, and before an explicit `Fee_ForceSectorSwitch` has been requested. Otherwise the sector switch would remove all outdated (removed or payload changed) data blocks

2.9 Flash Page Size impacts

The smallest writeable entity, called page size, differs for example from 2 byte on the S12XDP512 up to 128 byte on the TC1796. The size of every kind of data to be written must be rounded up to a multiple flash device's page size. Therefore padding is added, if the data doesn't fit a multiple of the page size.

For example, if you want to write 13 byte (incl. management information) into the flash of a TC1796, the padding which is needed is 115 byte to reach a multiple of the page size 128 byte. But if you want to write 13 byte (incl. management information) into the flash of a S12XDP512, the padding which is needed is only 1 byte to reach a multiple of the page size 2 byte.

This example shows, that the flash could be used much more efficient, if data which should be written matches the page size or is only a little smaller. That is to say, that it is possible to write more instances of a block/dataset of the same type into a smaller flash with small page size than into a larger flash where plenty of padding has to be added.

2.10 Services for handling under-voltage situations

The FEE provides two sets of functions enabling you to handle under-voltage situations properly.

Both sets focus on different use cases:

`Fee_SuspendWrites()` is intended to react on actual under-voltage situation detected via dedicated monitoring circuitry. Usually there is some amount of time (few milliseconds) to react on such conditions until a low voltage reset occurs. Its counterpart,

`Fee_ResumeWrites()`, was introduced in order to prevent from stalling, if voltage reaches normal range, without any reset.

The other set, namely `Fee_EnableFss()` and `Fee_DisableFss()`, respectively, is intended to signal the increased or decreased risk of resets to the FEE. For instance, usually engine start (cranking) might be a situation of higher risk, while a running engine might be a much safer situation. Since this function set deals with risks, they will only have a noticeable effect if the flash becomes too full, i.e. if Foreground Sector Switch Threshold exceeded. As long as flash is not at a critical fill level (denoted by Foreground Sector Switch Threshold), the write operations and (background) sector switch are permitted. On the other hand, execution of FSS may be disabled. In this case, write requests are forbidden, once FSS became necessary; they will fail with error result.

Caution
If this set of functions is enabled in configuration, FEE does not automatically enable FSS, i.e. execution of Foreground Sector Switches is disabled per default (after `Fee_Init()`). In order to enable execution of FSS, you will have to call `Fee_EnableFss()` once ECU start-up completed, and operating conditions are stable (esp. normal voltage).

Basic Knowledge
Execution of Sector switch (“garbage collection”) is essential to keep FEE writable. In case of exceeded Foreground Sector Switch Thresholds the flash is at a critical fill level, i.e. cleaning up has become urgently necessary. Writing new data in this situation might consume additional flash space; it would become insufficient to complete the sector switch afterwards.

Caution
It should be ensured `Fee_EnableFss()` will be called after start-up, at least under normal conditions. Otherwise FSS’s intention of “last-chance processing” would be foiled; exceeding FSS threshold would cause entering read only mode. This in turn should be exceptional behavior.

Expert Knowledge
As stated in chapter 2.7.6, criticality reduces once block copy has been completed. Thus, if `Fee_DisableFss()` was recently called, FEE remains writable, once block copy completed, and only sector erase is outstanding.

2.11 Critical Data Blocks

In order to keep FEE writable in case of sector overflows (chapter 2.7.3), it has to erase one logical sector. Though this event is an exception, occurring really rarely, the erase causes loss of recent data instances.

While such a loss would be uncritical to some data blocks, losing some other blocks' recent instances might cause the whole ECU to cease working. Such data blocks are essential to ECUs operability; losing them is critical (as would result in "defective ECU" requiring service). FEE configuration provides the possibility to mark these blocks.

Basic Knowledge
Typically data block's "criticality" rises with decreasing write frequency. The more infrequently a block will be written the less number of older instances exist, and the larger their semantic difference would be, disqualifying them for usage as fallback data.

As a result FEE ensures that any write access keeps all "critical" Data Blocks' most recent instances within one logical sector. Thus there is always one logical sector that might be erased (in case of error) without losing such "critical" data.

Keeping critical data together requires conditional execution of sector switch. If necessary, i.e. if a critical block cannot be written into the same logical sector containing all other critical blocks, FEE performs a Foreground Sector Switch.

Since each block configured to be critical may cause an additional sector switch before writing a new instance, flash usage over life-time increases due to additional block copies.

Note
Due to independency of partitions, the differentiation between critical and "not as critical" data blocks is also local to a partition. On one hand FEE only has to care about all critical blocks in a particular partition. On the other hand, resolving a sector overflow (by formatting a sector) may cause loss of recent uncritical data in only that partition.

Basic Knowledge
By marking all Blocks assigned to one partition as "critical", that partition can be configured to operate with so called "single sector usage", that is, FEE ensures that all data blocks' recent instances are located within a single logical sector. Before switching to the other one, all recent instances will be copied. This results in most robust operations (concerning aborts). However, there are penalties in performance, especially because no BSS can actually be performed, as well as in flash usage, because always all recent data instances will be copied.

Expert Knowledge
Even though the FEE in FBL does not know anything about Application's, critical blocks can be handled safely, unless added to FBL configuration and set to "uncritical" (thus configured inconsistently).

Caution
If Data Flash is being shared between Application and FBL, it is highly recommended to configure all FBL blocks being “critical”. Typically there are some blocks, containing information about Application’s validity and information whether to start Application or FBL. These blocks are critical to an ECU!

Note
It is not recommended to mark blocks as “critical” and “immediate”. Requirements on both types contradict: while “immediate” data are required to be written fast, “critical” data are required to be written very safely, implying additional operations, which slow down write performance.

2.12 Fee_MainFunction Triggering

In AUTOSAR release 4.x an additional option is introduced, to be able to call the `Fee_MainFunction` in a cyclic task or in a background task.

The cyclic task (default configuration) is used when the main function shall be triggered periodically. Typically the cycle time needs to be defined, for example 10ms.

If the `Fee_MainFunction` shall be accessed quicker, the function shall be called in a background task. The background task runs when the system has nothing to do further. The `Fee_MainFunction` is called as often as the available CPU load allow.

Caution
If the system is overloaded, it may happen that the background task is no longer called.

Note
The `Fee_MainFunction` should not be triggered faster than `Fls_MainFunction`, because the FEE must wait for the FLS.

3 Integration

This chapter gives necessary information for the integration of the MICROSAR FEE into an application environment of an ECU.

3.1 Scope of Delivery

The delivery of the FEE contains the files which are described in the chapters 3.1.1 and 3.1.2.

3.1.1 Static Files

File Name	Description
`Fee.h`	Defines the public interface of MICROSAR FEE module.
`Fee_InitEx.h`	Declares Fee_InitEx service. Shall be included by user if, and only if, extended initialization via `Fee_InitEx()` (see chapter 4.3.2) is used.
`Fee_Types.h`	Defines public types of FEE.
`Fee.c`	Implements MICROSAR FEE module's API.
`Fee_Processing.c`	Implements module's processing, including the state machine
`Fee_IntBase.h` `Fee_JopParams.h`	Defines basic internal type definitions
`Fee_Int.h`	Defines the internal interface for all module internal source files, as well as the container for all RAM variables which are used by the FEE.

`Fee_Partition.h`
`Fee_PartitionDefs.h`
`Fee_Partition.c`
	Provide internal services to manage partitions

`Fee_Sector.h`
`Fee_SectorDefs.h`
`Fee_Sector.c`
	Internal services abstracting access to logical sectors

`Fee_ChunkInfo.h`
`Fee_ChunkInfoDefs.h`
`Fee_ChunkInfo.c`
	Internal services providing access to Chunks and instances contained within.

`Fee_Cbk.h`	Defines the callback interface for the lower layer.
`Fee_bswmd.arxml`	Contains the formal notation of all information, which belongs to the FEE.
`Identifier.xml`	Defines all configuration parameters.[2]
`Fee.xml`	Defines the GUI which represents this module.2
`If_AsrIfFee.jar`	Generator plug-in for DaVinci Configurator 5

Table 3-1 Static files

Only Fee.h shall be included directly by other components.

[2] AUTOSAR 3.x deliveries only

3.1.2 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator.

File Name	Description
`Fee_Cfg.h`	Contains the static configuration part of this module.
`Fee_Lcfg.c`	Contains the link-time part of configuration.
`Fee_PrivateCfg.h`	Contains the static configuration part, which is only included of this module.

Table 3-2 Generated files

3.2 Compiler Abstraction and Memory Mapping

The objects (e.g. variables, functions, constants) are declared by compiler independent definitions – the compiler abstraction definitions. Each compiler abstraction definition is assigned to a memory section.

The following table contains the memory section names and the compiler abstraction definitions defined for the FEE and illustrate their assignment among each other.

Memory Mapping Sections \ Compiler Abstraction Definitions	FEE_API_CODE	FEE_APPL_CODE	FEE_APPL_CONFIG	FEE_APPL_DATA	FEE_CONST	FEE_PRIVATE_CODE	FEE_PRIVATE_CONST	FEE_PRIVATE_DATA	FEE_VAR	FEE_VAR_NOINIT
FEE_START_SEC_CODE	■					■				
FEE_START_SEC_CONST_UNSPECIFIED					■		■			
FEE_START_SEC_APPL_CONFIG_UNSPECIFIED			■							
FEE_START_SEC_VAR_NOINIT_UNSPECIFIED										■
FEE_START_SEC_VAR_INIT_UNSPECIFIED									■	

Table 3-3 Compiler abstraction and memory mapping

- `FEE_START_SEC_CODE / FEE_STOP_SEC_CODE`
  - Placement of all API functions
  - `FEE_API_CODE`
    Calling convention for all API functions
    Note that these functions are called from different modules, suitable convention depends on global section mapping.

  - `FEE_PRIVATE_CODE`
    Calling convention all internal functions
    It is recommended to locate all FEE code sections into one single output section and to make the internal function calls as “near” as possible.

- `FEE_START_SEC_APPL_CONFIG_UNSPECIFIED / FEE_STOP_SEC_APPL_CONFIG_UNSPECIFIED`
  - `FEE_APPL_CONFIG`
    Configurable constants allocated in Fee_Lcfg.c

- `FEE_START_SEC_CONST_UNSPECIFIED / FEE_STOP_SEC_CONST_UNSPECIFIED`
  - `FEE_PRIVATE_CONST`
    Internal constants, to be accessed only from within FEE.
  - `FEE_CONST`
    All module constants, to be accessed from outside the FEE

Expert Knowledge
At least on 16bit platforms it is recommended to locate code and constants into same memory area, in order to get most efficient access (“near”).

- `FEE_START_SEC_VAR_NOINIT_UNSPECIFIED / FEE_STOP_SEC_VAR_NOINIT_UNSPECIFIED`
  - `FEE_VAR_NOINIT`
    Module variables which do not need to be initialized (not even zeroed out)
    There are not intended to be accessed outside FEE.

- `FEE_START_SEC_VAR_INIT_UNSPECIFIED / FEE_STOP_SEC_VAR_INIT_UNSPECIFIED`
  - `FEE_VAR`
    Internal global variables of the module which must be initialized by start-up code and which are not fixed to one type

Note
Currently `Fee_ModuleStatus_t` is the only variable that needs to be initialized, in order to get the “(Un)Init Development Check” working.

If this check was disabled at pre-compile time, FEE does not have any initialized variables.

Caution
The distance of `FEE_APPL_DATA` must be the same or bigger than the distance of `FEE_VAR_NOINIT`.

`FEE_APPL_DATA` is used to reference to buffers provided by client software.

3.3 Dependencies on SW Modules

3.3.1 OSEK/AUTOSAR OS

This operating system is used for task scheduling, interrupt handling, global suspend and restore of interrupts and creating of the Interrupt Vector Table. Resources like shared variables can be protected by the usage of OS services.

Note
FEE does not directly depend on OS. Rather, dependency is actually created by integrator when configuring OS services and assigning Fee_MainFunction to an OS task

3.3.2 Module SchM

In an AUTOSAR environment, protection of “critical sections” is encapsulated by the Scheduling Manager, SchM.

Integrator has to ensure, SchM maps critical section functions to appropriate services. Therefore SchM just encapsulates dependency to OS.

Note
Dependency on SchM can be globally disabled in DaVinci Configurator. Then the (probable) more direct dependency to OS would be used.

3.3.3 Module Det

This module is the Development Error Tracer. It is optional and records all development errors for diagnostic purposes.

Its usage can be enabled and disabled by the switch `FEE_DEV_ERROR_DETECT`.

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()` (see chapter 2.6.3).

3.3.4 Module Fls

The Fls driver provides the access to the underlying hardware. The specific properties of the flash hardware influence the configuration of the FEE component.

Its services are called to request a special job to the driver.

3.3.5 Callback Functions

3.3.5.1 Lower layer interaction

The FEE offers the usage of callback notification functions for the underlying driver to inform the FEE that a job has finished successfully or not. The `Fee_JobEndNotification()` is called when a job is completed with a positive result and the `Fee_JobErrorNotification()` is called when a job is cancelled, aborted or failed.

Note
The interaction between FEE and the underlying driver does not need to be performed via a notification mechanism. Also polling mode can be chosen if desired.

However, using notifications, decreases CPU usage, because Fee_MainFunction will be left earlier as long as FEE waits for an Fls request to complete.

3.3.5.2 Upper layer interaction

The NVM offers the usage of callback notification functions for the FEE, to get informed that a job has finished successfully or not. The `Nvm_JobEndNotification()` is called when a job is completed with a positive result and the `Nvm_JobErrorNotification()` is called when a job is cancelled, aborted or failed.

Note
The interaction between NVM and FEE does not need to be performed via a notification mechanism. Also polling mode can be chosen if desired.

3.3.5.3 User Error Callback

Prototype	
`uint8 Appl_CriticalErrorCallback (uint8 partitionId, Fee_SectorError errCode)`	
Parameter	
partitionId	ID of partition the error occurred. Note that FEE publishes symbolic names (macros) as chosen in configuration.
errCode	See chapter 4.2.2
Return code	
`FEE_ERRCBK_REJECT_WRITE`	FEE's partition shall enter "read only mode" currently pending and all subsequent write requests will be completed with result `MEMIF_JOB_FAILED`
`FEE_ERRCBK_REJECT_ALL`	FEE' partition shall enter "reject all mode" all subsequent requests, including currently pending one, will be completed with result `MEMIF_JOB_FAILED`
`FEE_ERRCBK_RESOLVE_AUTOMATICALLY`	Try to resolve error automatically. This might result in loss of most recent data instances.
Functional Description	
This function may be implemented by environment software, if special handling for serious errors is necessary. For instance, FEE's default behavior might not be feasible, or additional handling, such as reporting an event to the DEM is required. **Info** FEE's default behavior is to keep itself running and writable. This means, per default the FEE tries to resolve the problem automatically, unless the error code is `FEE_SECTOR_FORMAT_FAILED`, where it would enter read-only mode. Additionally, if development mode is configured, checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).	
Particularities and Limitations	
■ This service is synchronous. ■ This service is non re-entrant. ■ This service is always available.	
Expected Caller Context	
■ This routine might be called on interrupt level, depending on the calling function.	

Each error results in a specific default behavior, which becomes effective if User Error Callback was disabled in configuration. This default behavior can also be requested from User Error Callback by using `FEE_ERRCBK_RESOLVE_AUTOMATICALLY`.

Error Code	Default behavior
FEE_SECTORS_CORRUPTED	Try to erase/re-init logical sectors.
FEE_SECTOR_OVERFLOW	Erase the newer logical sector.
FEE_SECTOR_FORMAT_FAILED	Enter read only mode. Return value FEE_ERRCBK_RESOLVE_AUTOMATICALLY may be used by user-implemented error callback. It results in retrying logical sector format. Since it might fail with same error, retries should be limited within error callback .
FEE_SECTOR_CRITICAL_FILL_LEVEL	Just continue, i.e. perform foreground sector switch, if it is enabled.

Table 3-4 Error Codes and FEE's default behavior

3.4 Dependencies on HW modules

The FEE is principally hardware independent. Nevertheless, the Fls driver has to provide some parameters (defined in the BSWMD file) the FEE relies on, e.g. the page size or sector sizes.

3.5 Critical Sections

In general Fee_MainFunction and FEE's job API may concurrently access variables; they need to be synchronized. FEE defines one critical section, FEE_EXCLUSIVE_AREA_0. Sections of code to be synchronized are very short; they contain only few instructions, and their run times do not depend on configuration. Therefore, a simple global interrupt lock may be used, though not necessary.

If Fee_MainFunction (the OS task(s) it is running in) cannot be preempted by callers of FEE API (especially NvM_MainFunction) and vice versa, no synchronization mechanism is necessary, at all.

Changes

A second type of critical section, FEE_EXCLUSIVE_AREA_1, became obsolete. It should not be used. Rather, it should just map to "nothing". As specified by AUTOSAR MainFunctions are not re-entrant.

If Fee_MainFunction may be called from different (task) contexts, which may pre-empt each other, a synchronization mechanism (e.g. an OS Resource) should be locked and released outside.

Usually other Main Function calls, in particular NvM_MainFunction and/or Fls_MainFunction would require synchronization as well. Thus, an externally defined mechanism could be better adapted to specific needs.

4 API Description

4.1 Interfaces Overview

For an interfaces overview please see Figure 1-2.

4.2 Type Definitions

4.2.1 Fee_SectorSwitchStatusType

Description	
This type specifies the possible status values of the sector switch.	
Range	
`FEE_SECTOR_SWITCH_IDLE`	The sector switch is currently not running.
`FEE_SECTOR_SWITCH_BLOCK_COPY`	The sector switch is currently busy with copying blocks/datasets in a partition from the logical source sector to the logical target sector.
`FEE_SECTOR_SWITCH_ERASE`	The sector switch is currently busy formatting the previous source sector in a partition.
`FEE_SECTOR_SWITCH_UNINIT`	The FEE is not initialized. This result may only be delivered if “Development Error Detection” was enabled.

Table 4-1 Fee_SectorSwitchStatusType

Info
This is an addition to AUTOSAR.

4.2.2 Fee_SectorErrorType

Description	
This type specifies the possible errors which shall describe erroneous situations the FEE can reach.	
Range	
`FEE_SECTORS_CORRUPTED`	The sector headers respectively the sector ID of both logical sectors could not be read. Hence, the most recent sector could not be determined.
`FEE_SECTOR_OVERFLOW`	Both logical sectors are completely filled and it is not possible to write any data. Nevertheless, it is still possible to read data from the Flash memory.
`FEE_SECTOR_FORMAT_FAILED`	One logical sector could not be allocated correctly, e.g. sector header is not valid.

`FEE_SECTOR_CRITICAL_FILL_LEVEL`	Foreground Sector Switch Threshold exceeded, and both logical sectors are currently in use. This means that the flash is nearly full. If FSS was disabled, this error code also denotes a temporary read only condition, since FEE won't write to flash unless Fee_EnableFss was called. **Note** This error code is obsolete; FEE does not report it any more. It is subject to be removed in future versions.

Table 4-2 Fee_SectorError Type

Info
This is an addition to AUTOSAR.

4.3 Services provided by FEE

The FEE API consists of services, which are realized by function calls.

Info
Most of the following API functions report development errors as listed in chapter 2.6.1. If an error is detected the concerning API function will be left without any further actions.

4.3.1 Fee_Init

Prototype	
`void Fee_Init(void)`	
Parameter	
--	--
Return code	
void	--
Functional Description	
This service initializes the FEE module and all needed internal variables. The FEE module doesn't support any runtime configuration. Hence, a pointer to the configuration structure is not needed by this service. The FEE does not initialize the underlying Flash driver, but this shall be done by the ECUM module. (This is no AUTOSAR deviation.) This function is specified in [1], and it should be used unless, a different configuration shall be used by FEE. **Info** This service calls Fee_InitEx (see clause 4.3.2), passing pointer to generated Fee_Config structure.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > This service is always available. > This service shall not be called during a pending job.	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-3 Fee_Init

4.3.2 Fee_InitEx

Prototype	
`void Fee_InitEx(Fee_ConfigRefType ConfigPtr)`	
Parameter	
ConfigPtr	Pointer to FEE's Block configuration, which is always named `Fee_Config`; it is defined in Fee_Lcfg.c, and its incomplete declaration is available via Fee.h. **Expert Knowledge** This parameter enables sharing FEE code between a Flash Bootloader and application, while both use different Block configurations.
Return code	
void	--
Functional Description	
This is an alternative/extended service to initialize the FEE module and all needed internal variables. The FEE does not initialize the underlying Flash driver, but this shall be done by the ECUM module. (this is no AUTOSAR deviation)	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > This service is always available. > This service shall not be called during a pending job.	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-4 Fee_InitEx

4.3.3 Fee_SetMode

Prototype	
`void Fee_SetMode(MemIf_ModeType Mode)`	
Parameter	
Mode	`MEMIF_MODE_SLOW`: Enable processing of Background Sector Switches `MEMIF_MODE_FAST`: Disable processing of Background Sector Switches
Return code	
void	--

Functional Description

Expert Knowledge
Deviating from [1], this function does not call underlying Flash driver's related function, `Fls_SetMode()`.

Info
This service is a synchronous call and does not need to be processed by the `Fee_MainFunction()`.

Note
Any value different from MEMIF_MODE_SLOW is treated as MEMIF_MODE_FAST. In particular, there is no parameter check.

This is important, especially if MemIf_ModeType (defined by MemIf implementation) happens to be not plain numeric type but not an `enum` type.

Caution
Calling `Fee_SetMode()` has significant impact on FEE's behavior. After startup of the ECU (ReadAll-process by the NVM has finished), the FEE is set to `MEMIF_MODE_SLOW`, which is initiated by the NVM, if configured accordingly. This is an indicator to the FEE, which enables processing relative time consuming sector switch in background, i.e. while no user jobs are pending. Refer to chapter 2.7 for details on sector switches.

On the other hand, during shut-down, performing sector switches in background may be inhibited by setting FEE to `MEMIF_MODE_FAST`.

Particularities and Limitations

> This service is synchronous.
> This service is non re-entrant.
> This service is always available.

Expected Caller Context

> Expected to be called in application context.

Table 4-5 Fee_SetMode

4.3.4 Fee_Read

Prototype

```
Std_ReturnType Fee_Read
(
  uint16 BlockNumber,
  uint16 BlockOffset,
  uint8 *DataBufferPtr,
  uint16 Length
)
```

Parameter

BlockNumber	Handle of a block (depending on block configuration)
BlockOffset	Read address offset inside the block
DataBufferPtr	Pointer to data buffer
Length	Number of bytes to read

Return code

E_OK	Read job has been accepted.
E_NOT_OK	Read job has not been accepted.

Functional Description

This function starts the read processing for the specified block.

Info
The job processing is asynchronous. The result of the finished job can be polled by calling `Fee_GetJobResult()`.

When the current block is found, the parameters BlockOffset and Length are used to call the Read function of the underlying Fls driver to read out the content of the flash memory to the provided DataBufferPtr.

As the processing is asynchronous the return value of the Fls Read function can only be returned indirectly via the job result.

Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).

Particularities and Limitations

- This service is asynchronous.
- This service is non re-entrant.
- This service is always available.

Expected Caller Context

- Expected to be called in application context.

Table 4-6 Fee_Read

4.3.5 Fee_Write

Prototype	
`Std_ReturnType Fee_Write ( uint16 BlockNumber, uint8 *DataBufferPtr )`	
Parameter	
BlockNumber	Handle of the block (depending on block configuration)
DataBufferPtr	Pointer to data buffer
Return code	
E_OK	Write job has been accepted
E_NOT_OK	Write job has not been accepted.
Functional Description	
This function starts the write processing for the specified block. **Info** The job processing is asynchronous. The result of the finished job can be polled by calling `Fee_GetJobResult()`. When the next free area for the block was found, the content of the provided DataBufferPtr is written to the flash memory. Additionally, management information is stored to identify the block. The real count of bytes which must be written depends on the hardware specific page alignment and the size of the management information. As the processing is asynchronous the return value of the Fls write function can only be returned indirectly via the job result. Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).	
Particularities and Limitations	
> This service is asynchronous. > This service is non re-entrant. > This service is always available.	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-7 Fee_Write

4.3.6 Fee_Cancel

Prototype	
`void Fee_Cancel(void)`	
Parameter	
--	--
Return code	
void	--
Functional Description	

This service cancels a currently pending job.

The state of the FEE will be set to `MEMIF_IDLE`.

If the FEE is currently `IDLE`, calling this service is without any effect.

Expert Knowledge
Additional actions, especially unwinding internal state machine stack and cancelling a pending Fls job, will be done asynchronously, within next Fee_MainFunction call

Changes
Previous versions delayed internal clean-up until next write request arrived in order to reduce probability of destructive cancellation. This behavior was removed – Fee_Cancel immediately cancels any processing.

Additionally, if development mode is configured, checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).

Particularities and Limitations

> This service is synchronous.
> This service is non re-entrant.
> This service is always available.

Expected Caller Context

> Expected to be called in application context.

Table 4-8 Fee_Cancel

4.3.7 Fee_GetStatus

Prototype	
`MemIf_StatusType` **`Fee_GetStatus`**`(void)`	
Parameter	
--	--
Return code	
MEMIF_UNINIT	The FEE is currently not initialized -> `Fee_Init()` must be called to use the functionality of the FEE.
MEMIF_IDLE	The FEE is currently idle -> no asynchronous job available
MEMIF_BUSY	The FEE is currently busy-> a asynchronous job is currently processed by the FEE
MEMIF_BUSY_INTERNAL	The FEE has to process internal operations to ensure further write and/or invalidate jobs.
Functional Description	
This service returns the FEE's current module state synchronously. Refer to chapter 2.3.1 for more details.	
Particularities and Limitations	
> This service is synchronous. > This service is re-entrant. > This service is always available.	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-9 Fee_GetStatus

4.3.8 Fee_GetJobResult

Prototype	
`MemIf_JobResultType` **`Fee_GetJobResult`**`(void)`	
Parameter	
--	--
Return code	
MEMIF_JOB_OK	The last job has been finished successfully.
MEMIF_JOB_PENDING	The last job is waiting for execution or currently being executed.
MEMIF_JOB_CANCELLED	The last job has been cancelled by the `Fee_Cancel()` service.
MEMIF_JOB_FAILED	The Flash driver reported an error or the FEE could not achieve the requested job due to hardware errors (e.g. memory cell defects).
MEMIF_BLOCK_INCONSISTENT	The data of requested block could not be read, because the data are corrupt.
MEMIF_BLOCK_INVALID	The requested block has been invalidated previously by the service `Fee_InvalidateBlock()` or reading from an erased block is achieved (independent from called `Fee_EraseImmediateBlock()` or a never written block).

Functional Description
This service returns the result of the last job executed. Refer to chapter 2.6.1 for more details.
Particularities and Limitations
> This service is synchronous. > This service is re-entrant. > This service is always available.
Expected Caller Context
> Expected to be called in application context.

Table 4-10 Fee_GetJobResult

4.3.9 Fee_InvalidateBlock

Prototype	
`Std_ReturnType` **`Fee_InvalidateBlock`**`(uint16 BlockNumber)`	
Parameter	
BlockNumber	Number of the block (depending on block configuration).
Return code	
E_OK	Invalidate job has been accepted.
E_NOT_OK	Invalidate job has not been accepted.
Functional Description	
This service invokes the invalidation procedure for the selected block. If the service succeeds the most recent data block is marked as INVALID. **Info** The job processing is asynchronous. The result of the finished job can be polled by calling `Fee_GetJobResult()`. When the last/current block was found the next free area will be written with special invalidate information. As this information is saved in flash memory it is resistant against resets. Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).	
Particularities and Limitations	
> This service is asynchronous. > This service is non re-entrant. > This service is always available.	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-11 Fee_InvalidateBlock

4.3.10 Fee_GetVersionInfo

Prototype	
`void` **`Fee_GetVersionInfo`**`(Std_VersionInfoType *VersionInfoPtr)`	
Parameter	
VersionInfoPtr	Pointer to where to store the version information of this module.
Return code	
void	--
Functional Description	
This service returns the version information of this module. The version information includes: > Module ID > Vendor ID > Instance ID > Vendor specific version numbers. Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 0). The function does not perform any action in case of a failure.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > This service is available, depending on pre-compile configuration checkbox 'Enable Fee_GetVersionInfo API' which is configured within the configuration tool.	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-12 Fee_GetVersionInfo

4.3.11 Fee_EraseImmediateBlock

Prototype	
`Std_ReturnType` **`Fee_EraseImmediateBlock`**`(uint16 BlockNumber)`	
Parameter	
BlockNumber	Number of the block (depending on block configuration).
Return code	
E_OK	Erase job has been accepted.
E_NOT_OK	Erase job has not been accepted.

Functional Description
This function doesn't erase flash memory. The addressed block is marked as invalid, thus a subsequent read request on the invalidated block completes with `MEMIF_BLOCK_INVALID`. **Info** The job processing is asynchronous. The result of the finished job can be polled by calling `Fee_GetJobResult()`. **Note** Fee does not care about immediate blocks; there is no special handling. Thus this service does exactly the same as Fee_InvalidateBlock
Particularities and Limitations
> This service is asynchronous. > This service is non re-entrant. > This service is always available. > This service shall only be called by e.g. diagnostic or similar system service to pre-erase the area for immediate data if necessary.
Expected Caller Context
> Expected to be called in application context.

Table 4-13 Fee_EraseImmediateBlock

4.3.12 Fee_MainFunction

Prototype	
`void Fee_MainFunction(void)`	
Parameter	
--	--
Return code	
void	--
Functional Description	
This service triggers the processing of the internal state machine and handles the asynchronous job and management operations. The complete handling of the job and the detection of invalidated or inconsistent blocks will be done in the internal job state machine. Additionally, if development mode is configured, checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).	

Particularities and Limitations
> This service is synchronous. > This service is non re-entrant. > This service is always available.
Expected Caller Context
> Expected to be called in application context.

Table 4-14 Fee_MainFunction

4.3.13 Fee_GetEraseCycle

Prototype

```
Std_ReturnType Fee_GetEraseCycle(
  uint8 SectorNumber,
  uint32 *DataPtr
)
```

Parameter	
SectorNumber	Identifies partition and logical sector whose erase cycle counter shall be retrieved: The least significant bit chooses the sector (0 or 1, meaning lower sector or upper sector, respectively); higher bits identify the partition. Partitions' symbolic names may be used, but values must be adapted (doubled).
DataPtr	Pointer to data buffer.

Return code	
E_OK	Job has been accepted.
E_NOT_OK	Job has not been accepted.

Functional Description
This service retrieves the erase cycle counter of the specified logical sector in the provided buffer. The user can determine how often the specific sector has been erased. **Info** This API service is not used by the NVM. It is just a feature to retrieve the number of erase cycle of a specific logical sector. **Info** The job processing is asynchronous. The result of the finished job can be polled by calling `Fee_GetJobResult()`. Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1). **Info** This is an addition to AUTOSAR.
Particularities and Limitations
> This service is asynchronous. > This service is non re-entrant. > This service is available, depending on pre-compile configuration checkbox 'Enable Fee_GetEraseCycle API' which is configured within the configuration tool.
Expected Caller Context
> Expected to be called in application context.

Table 4-15 Fee_GetEraseCycle

4.3.14 Fee_GetWriteCycle

Prototype

```
Std_ReturnType Fee_GetWriteCycle
(
  uint16 BlockNumber,
  uint32 *DataPtr
)
```

Parameter	
BlockNumber	Number of the block, provided by the FEE.
DataPtr	Pointer to data buffer.
Return code	
E_OK	Job has been accepted.
E_NOT_OK	Job has not been accepted.

Functional Description
This service retrieves the write cycle counter of a specified block and saves in the provided buffer. **Info** This API service is not used by the NVM. It is just a feature to retrieve the number of write cycles of a specific block. **Info** The job processing is asynchronous. The result of the finished job can be polled by calling `Fee_GetJobResult()`. Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1). **Info** This is an addition to AUTOSAR.
Particularities and Limitations
> This service is asynchronous. > This service is non re-entrant. > This service is available, depending on pre-compile configuration checkbox 'Enable Fee_GetWriteCycle API' which is configured within the configuration tool.
Expected Caller Context
> Expected to be called in application context.

Table 4-16 Fee_GetWriteCycle

4.3.15 Fee_GetSectorSwitchStatus

Prototype	
`Fee_SectorSwitchStatusType` **`Fee_GetSectorSwitchStatus`**`(void)`	
Parameter	
--	--
Return code	
`Fee_SectorSwitchStatusType`	see chapter 4.2.1

Functional Description
This function returns the current status of the sector switch. See chapter 4.2.1 for the specific return values. For more information about the sector switch see chapter 2.7. **Info** This service determines FEE's **current** sector switch processing state. Necessity of a sector switch does not imply this service to return a value different `FEE_SECTOR_SWITCH_IDLE`. For example, FEE will also report `FEE_SECTOR_SWITCH_IDLE` when copying a block completed, until it starts the next one. **Info** This is an addition to AUTOSAR.
Particularities and Limitations
> This service is synchronous. > This service is non re-entrant. > This service is always available.
Expected Caller Context
> Expected to be called in application context.

Table 4-17 Fee_GetSectorSwitchStatus

4.3.16 Fee_ForceSectorSwitch

Prototype	
`Std_ReturnType` **`Fee_ForceSectorSwitch`**`(void)`	
Parameter	
--	--
Return code	
E_OK	Job has been accepted.
E_NOT_OK	Job has not been accepted.

Functional Description

This service forces a sector switch to be performed on all configured partitions in “Foreground Mode”, i.e. the FEE will defer next incoming job until switch has been completed.

Purpose of this API is to compact partitions’ flash usages to one logical sector each.

FEE performs a sector switch (tries to copy data) even if it just completed one, i.e. it just cleaned up flash.

Subsequent calls to this service while FEE already is processing such a request are without any noticable effects; internally FEE restarts processing.

Normally, issuing one single request is sufficient. However, in some cases it might be desirable to have both sectors of every partition cleaned up. This requires forcing two sector switches:

```
Fee_ForceSectorSwitch();
<wait until IDLE>
Fee_ForceSectorSwitch();
```

Note
It **is not mandatory** to explicitly wait for FEE becoming IDLE, after issuing one single or for last `Fee_ForceSectorSwitch` request.

However, it **is mandatory** to wait between different requests, if several Sector Switches are required.

Note
In order to avoid disturbing FEE’s client layer (e.g. NVM), this service does not set FEE’s status to `MEMIF_BUSY`. Instead it makes sure FEE is `MEMIF_BUSY_INTERNAL` (unless it’s already `MEMIF_BUSY`).

Caution
Fee must be initialized before calling `Fee_ForceSectorSwitch()`.

Note
This API service is not used by the NVM.

Note
The processing state can be queried using Fee_GetStatus. It switches to `MEMIF_IDLE` once all partitions have been processed.

Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).

Note
This service is an addition to AUTOSAR.

Particularities and Limitations
> This service is asynchronous.
> This service is non re-entrant.
> This service is available, depending on pre-compile configuration checkbox 'Enable Fee_ForceSectorSwitch API' which is configured within the configuration tool.
Expected Caller Context
> Expected to be called in application context.

Table 4-18 Fee_ForceSectorSwitch

4.3.17 Fee_ConvertBlockConfig

Prototype

```
Std_ReturnType Fee_ConvertBlockConfig
(
  const Fee_ConversionOptionsType* options
)
```

Parameter		
Options	Pointer to structure of type `Fee_ConversionOptionsType`, containing the pointer to the user buffer and to the callback to be invoked for each block:	
	userBuffer	Pointer to user buffer to hold block data. Note that the area pointed to must be large enough to hold largest possible block's (across all configurations) data.
	notificationPtr	Pointer to callback function.

Return code	
E_OK	Job has been accepted.
E_NOT_OK	Job has not been accepted.

Functional Description

This service requests conversion of block configuration, as described in 2.7.5.

FEE uses the buffer passed in `options->userBuffer` to read block data from data flash and to pass them to the user callback `options->notificationPtr` for each block it finds in data flash (having a VALID instance).

Caution
Fee must have been initialized and it must be idle before calling **`Fee_ConvertBlockConfig`**`()`.

Additionally, the upper SW layers (NvM) shall be stalled, i.e. it must be prevented from issuing requests to the FEE.

Basic Knowledge
Block conversion affects all partitions, i.e. FEE will iterate over all partitions. Don't expect any specific order; decisions should be based on parameters passed to "Data Conversion Callback" (chapter 4.3.17).

Info
This API service is not used by the NVM.

Info
Job completion may be polled by calling `Fee_GetStatus()`; Job result may be determined using `Fee_GetJobResult()`.

Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to chapter 2.6.1).

Info
This service is an addition to AUTOSAR.

Particularities and Limitations

- This service is asynchronous.
- This service is non re-entrant.
- This service is available, depending on pre-compile configuration checkbox 'Enable Fee_ConvertDataBlocks API' which is configured within the configuration tool.

Expected Caller Context

> Expected to be called in application context.

Table 4-19 Fee_ConvertBlockConfig

4.3.18 Fee_SuspendWrites

Prototype	
`void Fee_SuspendWrites(void)`	
Parameter	
-	-
Return code	
void	--
Functional Description	
This service instructs Fee to block all write class jobs (writing, invalidating and erasing a block). Pending jobs will be blocked, i.e. they won't be finished. Next Fee_MainFunction calls cause FEE to enter a safe state (by means of Flash content). Once such state was reached, FEE does not issue new write requests to Fls. Multiple subsequent calls to this service don't have additional effects, i.e. to re-enable write accesses only one call to Fee_ResumeWrites is necessary. ***Info*** *As long as write class jobs are suspended no sector switch will be executed!* For more information, refer to chapter 2.10.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > This service is always available.	
Expected Caller Context	
> May be called at interrupt level.	

Table 4-20 Fee_SuspendWrites

4.3.19 Fee_ResumeWrites

Prototype	
`void Fee_ResumeWrites(void)`	
Parameter	
-	–
Return code	
void	--

Functional Description
This service instructs Fee to allow all write class jobs (writing, invalidating and erasing a block), including sector switch processing, again which have been suspended using `Fee_SuspendWrites()`. Multiple calls to this service enable write processing once, i.e. to disable it again there is still one call to Fee_SuspendWrites necessary. For more information, refer to chapter 2.10.
Particularities and Limitations
> This service is synchronous. > This service is non re-entrant. > This service is always available.
Expected Caller Context
> May be called at interrupt level.

Table 4-21 Fee_ResumeWrites

4.3.20 Fee_DisableFss

Prototype	
`void Fee_DisableFss(void)`	
Parameter	
-	-
Return code	
Void	-
Functional Description	
This function disables execution of foreground sector switch when threshold is reached. A typical situation using this function is start of engine. **Info** When foreground sector switch threshold is reached and a write class job (writing, erasing or invalidating a block) shall be processed, this job ends with result MEMIF_JOB_FAILED. For more information, refer to chapter 2.10.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > Availability depends on setting of “Enable API to allow/prohibit FSS” (see ch. 5.1.7.1)	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-22 Fee_DisableFss

4.3.21 Fee_EnableFss

Prototype	
`void Fee_EnableFss(void)`	
Parameter	
-	-
Return code	
void	-
Functional Description	
This function enables execution of foreground sector switch when threshold is reached. For more information, refer to chapter 2.10.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > Availability depends on setting of “Enable API to allow/prohibit FSS” (see ch. 5.1.7.1)	
Expected Caller Context	
> Expected to be called in application context.	

Table 4-23 Fee_EnableFss

4.4 Services used by FEE

In the following table services provided by other components, which are used by the FEE are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
DET	Det_ReportError (optionally)
FLS	Fls_Read Fls_Write Fls_Erase Fls_GetStatus (if polling mode deactivated) Fls_GetJobResult (if polling mode deactivated) Fls_SetMode Fls_Cancel
NVM	NvM_JobEndNotification (optionally) NvM_JobErrorNotification (optionally)
OS	Interrupt locking/unlocking functions (optionally)

Table 4-24 Services used by the FEE

4.4.1 Data Conversion Callback

This user-implemented callback is related to the Feature “Data Conversion” (refer to 2.7.5); a pointer to such a function shall be passed to `Fee_ConvertBlockConfig` (see 4.3.17).

Prototype	
Single Channel	

`uint8 <Function_Name>`
`(`
`  uint8* userBuffer,`
`  uint32 blockId,`
`  uint16 oldLength,`
`  uint16 newLength`
`)`	

Parameter	
userBuffer	Pointer to user data buffer, containing block’s most recent data read from flash. Content may be modified. Points to the userBuffer originally passed to `Fee_ConvertBlockConfig`
blockId	Unique 28bit block identifier, consisting of 4bit Partition Index (bits 27:24) 16bit “Block Tag” (bits 23..16) and 8bit data index (bits 7..0)[3]
oldLength	Old data length, as found in flash; the user buffer will contain exactly `oldLength` data bytes
newLength	New data length as given in current configuration, the FEE has been initialized with.
Return code	
`FEE_CONVERSION_WRITE_OLD_LENGTH`	Instruct the FEE to keep data (user buffer’s content) according to old length. Data to be written will not be readable using the `Fee_Read` (if the both lengths actually differ), but they will be kept in flash, after `Fee_ConvertBlockConfig` processing completes. It is useful if several subsequent runs of `Fee_ConvertBlockConfig` are necessary to perform update with multiple stages.
`FEE_CONVERSION_WRITE_NEW_LENGTH`	Instruct the FEE to write data (user buffer’s content) according to new length.
`FEE_CONVERSION_SKIP`	Instruct the FEE to skip this block. It will write nothing. Block data will actually get lost when `Fee_ConvertBlockConfig` processing completes.
Functional Description	
This callback has to be implemented by user. It should synchronously perform any action that is necessary to convert block data from old format to new format (e.g. lengths).	
Particularities and Limitations	
> Since the FEE is busy with Fee_ConvertBlockConfig processing, it is not allowed to issue any other request to it.	

[3] Bit 0 is defined to be the least significant bit, regardless of used platform.

Call context
> Called from context of `Fee_MainFunction`

Table 4-25 User defined conversion callback

4.5 Callback Functions

This chapter describes the callback functions that are implemented by the FEE and can be invoked by other modules. The prototypes of the callback functions are provided in the header file Fee_Cbk.h by the FEE.

4.5.1 Fee_JobEndNotification

Prototype	
`void Fee_JobEndNotification(void)`	
Parameter	
--	--
Return code	
void	--
Functional Description	
This routine shall be called by the underlying flash driver to report the successful end of an asynchronous operation. **Info** This function is configurable at pre-compile time using the parameter `FEE_POLLING_MODE`.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > This service is always available.	
Expected Caller Context	
> This routine might be called on interrupt level, depending on the calling function.	

Table 4-26 Fee_JobEndNotification

4.5.2 Fee_JobErrorNotification

Prototype	
`void Fee_JobErrorNotification(void)`	
Parameter	
--	--
Return code	
void	--

Functional Description
This routine shall be called by the underlying flash driver to report the failure of an asynchronous operation. **Info** This function is configurable at pre-compile time using the parameter `FEE_POLLING_MODE`.
Particularities and Limitations
> This service is synchronous. > This service is non re-entrant. > This service is always available.
Expected Caller Context
> This routine might be called on interrupt level, depending on the calling function.

Table 4-27 Fee_JobErrorNotification

4.6 Configurable Interfaces

API	Description
Function `Fee_GetVersionInfo()`	This function can be enabled/disabled by the configuration switch 'Enable Fee_GetVersionInfo API'. Refer to chapter 5.1.7.
Function `Fee_GetEraseCycle()`	This function can be enabled/disabled by the configuration switch 'Enable Fee_GetEraseCycle API'. Refer to chapter 5.1.7.
Function `Fee_GetWriteCycle()`	This function can be enabled/disabled by the configuration switch 'Enable Fee_GetWriteCycle API'. Refer to chapter 5.1.7.
Function `Fee_ForceSectorSwitch()`	This function can be enabled/disabled by the configuration switch 'Enable Fee_ForceSectorSwitch API'. Refer to chapter 5.1.7.

Functions
`Fee_JobEndNotification()`
`Fee_JobErrorNotification()`
	The functions can be enabled/disabled by the configuration switch 'Poll Flash driver'. Refer to chapter 5.1.6.2.

Functions
`Fee_EnableFss()`
`Fee_DisableFss()`
	The functions can be enabled/disabled by the configuration switch 'Enable API to allow/prohibit FSS'. Refer to chapter 5.1.7.

Function `Fee_ConvertBlockConfig()`	The functions can be enabled/disabled by the configuration switch 'Enable Data Conversion API'. Refer to chapter 5.1.7.

Table 4-28 Configurable interfaces

5 Configuration

FEE can be configured using following tools:

> DaVinci Configurator 5, domain “Memory” (AUTOSAR 4 packages only). Parameters are explained within the tool; parameters described in this chapter might not directly correspond to parameters visible in Configurator 5’s GUI.

> DaVinci Configurator 4 (AUTOSAR 3 packages only; for a detailed description see this chapter)

> Using a generic configuration editor (GCE)

5.1 Configuration with DaVinci Configurator

5.1.1 Start configuration of the FEE

The component name of the Flash-EEPROM-Emulation in DaVinci Configurator is “FEE”. In the “Architecture view” (initial page) of the DaVinci Configurator, the FEE can be opened by its context menu to start its configuration. Optionally, the FEE can be opened for configuration with the component list under the “Memory” tab located at the left side of the DaVinci Configurator.

5.1.2 Useful Chunk-Sizes (instance counts)

Carefully chosen chunk-sizes may significantly improve FEE’s performance as well as robustness.

Note
Chunk sizes’ effects are basically local to a single partition, i.e. a partition’s performance and robustness is affected. Another partition might be affected indirectly by sector switch processing (which is not immediately interruptible).

Therefore, when changing a block’s settings, other blocks in same partition must be considered.

Though it is not necessary (usually it is not even possible) to estimate optimum chunk-sizes, they should be configured to reasonable values. Therefore their effects are summarized first:
Basically a block’s instance count depends on the estimated number of write cycles, relative to other blocks’ write cycles, as well as on its size, in relation to logical sector size. A “(too) small” / “(too) large” chunk is meant to be related with “write cycle” and/or “block’s size”.

Small chunks:

> In general small chunks are suited for large and/or infrequently written blocks. Allocating a small chunk results in less flash-space being reserved for related block (instances).

- Smaller chunks result in less space wasted by aborts (resets), and more retries being possible in a sector in case of resets.
- They result in more overhead in flash. This becomes significant with small blocks (little payload), large page size, and number of write cycles.
- Decreased efficiency of flash usage, i.e. more erase cycles over life-time
- Result in increased average search efforts. Since FEE does not hold position information in RAM (neither Look-up tables, nor any caching mechanisms), searching must be done for each asynchronous block operation. Since one block's chunks (within one logical sector) build up a linked list; this list becomes longer. Each list node (chunk) requires one additional flash request, requiring additional Fee_MainFunction and Fls_MainFunction cycles.

Large chunks:

- Frequently written, small blocks usually benefit from larger chunks.
- Overhead in flash is reduced
- Decreased average search efforts, for two reasons:
  - Skipping an obsolete chunk means skipping more obsolete instances; the linked list becomes shorter.
  - Searching within a chunk is implemented to have logarithmic effort.
- Allocating a large chunk results in more flash space (for more instances) being reserved. If chunk is too large, it would never be used, because writing other blocks already caused a sector switch.
- Increased flash usage also may increase vulnerability to aborts, because space for other blocks shrinks. Usually each block requires at least one chunk per logical sector (to hold at least data copied during sector switch).

The larger a block's payload is, the smaller the instance count should be. It becomes less and less possible that such a chunk could be too small, because there is an absolute upper limit, how often a chunk fits into logical sector.

It is highly not recommended to try "optimizing" a block configuration in order to get only one chunk per block, resulting in nearly completely filled logical sector. Rather the chunk sizes should be set in a way that some reserve remains to be used dynamically. It is advisable to use approximately 50% of smaller logical sector. However it is more important to keep enough space to write even the largest blocks several times (regardless of actual write cycles ➔ due to aborts). On the other hand reserves might also be lower, if a partition will be written (very) infrequently, and if written to, the situation is known to be stable (very low risk of aborted writes).

Based on estimated number of write cycles, it is sufficient to care about chunk sizes of most frequently written blocks only (additionally considering their sizes). In typical

configurations the write cycles differ by magnitudes (1 … 1’000’000) – only the largest ones (e.g. more than 100’000 cycles, then more than 10’000 cycles) need to be considered. Blocks being written less than one hundredth of most frequently written one, may usually be considered to be constant; their instance count should be 1. If one or more large blocks are frequently written, check their maximum possible number writes per sector. This can be used as threshold to consider blocks being constant.

5.1.3 Update of block configuration

It is possible to update the block configuration (size, datasets,...) of each block. Every block which should be readable after the block configuration update must get the same BlockID as before (preconditioned that the size of this block has not changed). The BlockID could be set manually if the “BlockID fixed” is checked.

Note
It is not possible to “move” a block from one partition to another with an update, while keeping its data.

After flashing the new block configuration it is possible, but not actually necessary, to use `Fee_ForceSectorSwitch()` in order to clean up flash contents, causing flash layout to be prepared. In order to perform a complete update of flash layout – i.e. to clean up / update all logical sectors, across all partitions –, a second subsequent run of `Fee_ForceSectorSwitch()` may be used .

Writing updated blocks will be possible independent of usage of `Fee_ForceSectorSwitch()`.

5.1.4 FEE Configuration tab

Attribute Name	Value Type	Values The default value is written in bold	Description
Block Configuration			
FBL Configuration	--	ON **OFF**	This checkbox must be checked, if the current block configuration is the flash boot loader configuration. Otherwise it must not be checked.
Insert Block	--	**--**	This button inserts a FEE block into the table. This is only necessary if additional blocks are needed. All NVM blocks are inserted automatically.
Delete Block	--	**--**	This button deletes the selected block from the table. Only blocks which where inserted with the "Insert Block" button should be deleted with this button. NVM blocks should be deleted in the NVM configuration. They are deleted automatically in the FEE block table.
Move Up	--	**--**	This button moves a selected block one row above. The order of the block in the table influences the physical default position within the Flash.

Attribute Name	Value Type	Values The default value is written in bold	Description
Move Down	--	**--**	This button moves a selected block in the table one row below. The order of the block in the table influences the physical default position within the Flash.
Calculate	--	**--**	This button calculates the FEE block numbers, as well as BlockIDs, unless set to “fixed”. **Caution** This button must be pressed before generation process is started.
BlockID	uint16	**no default** 0...65535	The BlockID will be set automatically by clicking the “Calculate”-button. The BlockID could also be set manually (see BlockID fixed). The BlockID must be unique within a partition, i.e. amongst all blocks assigned to same partition. In order to save flash space it is recommended to start numbering with 0 for each partition. **Caution** If the block configuration shall be updated, each block which should be readable after the block configuration update must get the same BlockID as before.
NVM Blockname	--	**--**	This field shows the name of the corresponding NVM block entered within the NVM and is forwarded to the FEE. For user added blocks (within the FEE) this field is empty. **Info** The NVM Blockname can not be modified at all. It provides the information to the user to associate the FEE block configuration to the corresponding NVM block.

Attribute Name	Value Type	Values The default value is written in bold	Description
Fee Blockname	C-identifier	Valid C identifier, default: **Fee_User Block<n>**	This field shows the name of the configured FEE block. In the AUTOSAR ECUC file, it becomes the name of the Block Configuration container. **Info** As required by AUTOSAR, the linkage between block name and its numeric handle (“Block Num”) will be generated. Depending on global tool settings, the prefix “Fee_” will be added, or omitted.
Block Num	uint16	2…65534, no default	The even number of the FEE block. These not modifiable values will be calculated by pressing the button "Calculate". The dataset selection bits are used to calculate the block number. **Caution** The BlockNum should not be used directly. Instead use the symbolic block names, defined in Fee_Cfg.h (see chapter 3.1.2).
Datasets	--	**1** 1…255	This field determines the number of datasets configured for a dedicated block. **Info** The number of datasets can not be modified if the block is configured within and forwarded from the NVM. If the block is created within the FEE, the datasets can be adjusted for corresponding block.
Size	uint16	**1** 1…65535	The natural size (in bytes) of the block (payload). **Info** The size can not be modified if the block is configured within and forwarded from the NVM, because these blocks provide their size on their own.

Attribute Name	Value Type	Values The default value is written in bold	Description
Chunk block count	--	**1**, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, 32767	This field determines the number of blocks which can be stored within one block chunk. For blocks which are written often the number should be larger and vice versa. **Caution** This value must always be adjusted within the FEE and is never forwarded from the upper layer, because this value is responsible to abstract hardware constraints to the upper layer.
Partition	Ref.	--	Reference to a defined partition container.
Write Cycles	--	1… **10000** ...10000000	The estimated/desired number of write cycles which are required for the block. According to this value, the number of blocks within a chunk shall be influenced, but which is not done automatically and is left to the user. **Info** Currently, the adjustment of this value has no effect. Instead the number of blocks within a chunk (Chunk block count) influences the distribution of the data/blocks on the Flash memory.
High Prio	--	ON **OFF**	This field denotes that the block contains high priority data. If it is 'ON' the block/dataset can be erased and contains high priority data. If it is 'OFF' the block/dataset cannot be erased and contains "normal" data. For example, high priority data can be supposed as crash data. **Info** The marker for high priority data can not be modified if the block is configured within and forwarded from the NVM. If the block is created within the FEE, the datasets can be adjusted for corresponding block. **Caution** A block can only be erased by the API function `Fee_EraseImmediateBlock()` if the appropriate block is set to contain high priority data.
Critical Data	--	ON **OFF**	Marks a block to be critical, i.e. essential, for ECU operation. Refer to chapter 2.11.
BlockID fixed	--	ON **OFF**	This checkbox must be checked, to be able to set the BlockID manually.

Table 5-1 Fee configuration

5.1.5 General Settings tab

5.1.5.1 Error Detection – Development Mode

Attribute Name	Value Type	Values The default value is written in bold	Description
Enable Development Error Detection	--	ON **OFF**	Preprocessor switch to enable/disable development error detection and reporting. It is the main switch over following items: > Errorhook > Include File In production mode, this switch should be disabled to save ROM/RAM and to speed up the module.
Development Error Reporting	--	**ON** OFF	Preprocessor switch for enabling/disabling the Development Error Reporting.
Errorhook	C-function identifier	Valid C-function identifier, default: **Det_ Report Error**	Specifies the function that shall be called if a development error has occurred. (see chapter 2.6.3 for function signature)
Include File	header file	Valid header file, default: **Det.h**	Specifies the file that shall be included if development error reporting shall be used. The API used by the FEE must be specified within this file.

Table 5-2 Error Detection – Development Mode

5.1.5.2 Area “Error Callback”

Attribute Name	Value Type	Values The default value is written in bold	Description
Use Error Callback	--	ON, **OFF**	Defines whether the Error Callback notification mechanism shall be used to inform the application about a serious condition the Fee had detected.
Callback Function	C-function identifier	Valid C-function identifier, default: **Appl_CriticalError Callback**	Defines the function of the application that will be called if a critical condition has been entered by the Fee. It is expected, that the application is responsible to react on this situation according to the parameter value which has been passed. Refer to chapter 3.3.5.3
Include File	header file	Valid header file, default: **Appl_Include.h**	Defines the header file containing the Error Callback function.

Table 5-3 Error Callback

5.1.5.3 Area Buffer

Attribute Name	Value Type	Values The default value is written in bold	Description
Internal Buffer Size	uint16	**64,** 128, 256, 512, 1024	Size of internal buffer to be used for instance allocation during write job processing and for instance copy during sector switch. Must be an integral power of two. Must be equal to or larger than “Write Alignment” setting. **Info** The number of Fls_Read – Fls_Write request pairs necessary to copy a whole block depends on its size and Internal Buffer size.

5.1.5.4 Area “Upper Layer”

Following controls are visible only, if an NvM is enabled (part of configuration) and its parameter "Use Polling Mode" (NvmPollingMode) is unchecked (set to False). A hint about NVM’s current polling mode setting (including availability at all) will be additionally shown.

FEE should be used in polling mode, if NVM is disabled.

Attribute Name	Value Type	Values The default value is written in bold	Description
Job End Notification	C-function identifier	Valid C-function identifier, default: **NvM_JobEnd Notification**	The name of the job end callback function shall be inserted to this field. It is called to report to the upper layer that the job processing was finished successfully. Usually this function is provided by the NVM.
Job Error Notification	C-function identifier	Valid C-function identifier, default: **NvM_JobError Notification**	The name of the job error callback function shall be inserted to this field. It is called to report to the upper layer that the job processing was finished with failure. Usually this function is provided by the NVM.
Include File for Callbacks	header file	Valid header file, default: **NvM_Cbk.h**	In this field it can be inserted the name of the include file that holds the callback declarations of the upper layer.

Table 5-4 Upper Layer

5.1.5.5 Area “Critical Section Handling”

Changes
Fee 8.xx.xx only supports Critical Section services provided by an AUTOSAR Basic Software Scheduler (SchM). It ignores manual settings made, if “Use BSW Scheduler (SchM)” was disabled in “Board Setting / OS Sevices”

Attribute Name	Value Type	Values The default value is written in bold	Description
Short-Lasting Actions	--	**Use Suspend Functions** UseOS Functions UseEnable Functions	Defines the function set that will be called when short-lasting critical sections are entered or left. Refer to chapter 3.5.
Long-Lasting Actions	--	**Use Suspend Functions** UseOS Functions UseEnable Functions	Defines the function set that will be called when long-lasting critical sections are entered or left. Long-lasting actions are Fee_MainFunction as a whole. Refer to chapter 3.5.

Table 5-5 Critical Section Services

Note
Both dropdown lists are invisible, if the BSW Scheduler is enabled within the OS configuration tab of the ECU configuration.

5.1.6 Partitions

The left panel of partitions shows a tree-like overview of currently configured partitions. In this panel partitions can be added, deleted and renamed, as well.

Attribute Name	Value Type	Values The default value is written in bold	Description
Partition Device	Ref.	--	Choose a flash configuration. All partitions in one single flash device must refer to the same Flash Configuration (of type `/AUTOSAR/Fls/FlsConfigSet`) Currently, FEE is restricted to use one Flash Driver only, i.e. all partitions must point the same configuration.
Lower Sector Start Address	--	**--**	In this combo box the start address of the lower logical sector shall be selected. The values of the list are just typical values. A custom value may be entered; it must adhere to the “Fls address alignment” (see below).
Lower Sector Size	--	**1024**	Choose lower logical sector’s size in bytes. The combo box suggests some typical values; a custom size may be entered. The size must also adhere to “Fls address alignment” (see below).

Attribute Name	Value Type	Values The default value is written in bold	Description
Upper Sector Start Address	--	**--**	In this combo box the start address of the upper logical sector shall be selected. The values of the list are suggestions, derived from the chosen Fls driver (which depends on the used controller). A custom value may be entered; it must adhere to the “Fls address alignment” (see below). The upper sector’s start address must be located in a distinct physical sector, with higher address, than the last address of the lower sector.
Upper Sector Size	--	**1024**	Choose upper logical sector’s size in bytes. The combo box suggests some typical values; a custom size may be entered. The size must also adhere to “Fls address alignment” (see below).
Fls address alignment	--	**8,** 64, 128, 256, 512, 1024	In this drop down list the virtual page size in bytes will be set. This must be a power of two and be equal to or larger than the “Flash page size for write jobs”. This value will be used to check logical sectors’ alignment, to align of block instances as well as Fee management information.
Fls Page Size for write jobs	--	**8,** 64, 128, 256, 512, 1024	In this drop down list the Fls page size for write jobs in bytes will be set. This must be an integral multiple of the hardware specific flash page size. This value will be used for the write alignment, instead of the Fls page size of the hardware.

Table 5-6 Lower Layer

5.1.6.1 Area “Management”

Attribute Name	Value Type	Values The default value is written in bold	Description
Background Reserve	uint32	--	If there is less free space by writing data continuously to the Fee, the sector switch in background mode will be started. **Info** See chapter 2.7 for more details.
Foreground Reserve	uint32	--	If there is less free space by writing data continuously to the Fee, the sector switch in foreground mode will be started. **Info** See chapter 2.7 for more details.

Table 5-7 sector switch reserve

5.1.6.2 Area “Lower Layer”

This panel is independent from partitions; its setting has global meaning.

Attribute Name	Value Type	Values The default value is written in bold	Description
Poll Flash Driver	--	**ON** OFF	Enables/disables polling of the underlying Flash driver. If polling is disabled, make sure, that the callbacks `Fee_JobEndNotification()` and `Fee_JobErrorNotification()`, respectively, have been configured in the Flash driver. **Info** If polling is enabled the callback functions are not available, because there is no needed to. An additional hint will be visible in that case.

Table 5-8 Lower Layer

5.1.7 Module API tab

5.1.7.1 API Configuration

Attribute Name	Value Type	Values The default value is written in bold	Description
Enable Fee_GetVersionInfo API	--	**ON** OFF	Preprocessor switch to enable/disable the existence of the API service `Fee_GetVersionInfo()`. **Info** If this checkbox is switched off, the corresponding parameter check ("Check Parameter VersionInfo") within the "General Settings" tab is deselected and disabled.
Enable Fee_GetEraseCycle API	--	ON **OFF**	Preprocessor switch to enable/disable the existence of the API service `Fee_GetEraseCycle()`. **Info** If this checkbox is switched off, the corresponding parameter check ("Check Parameter Sector Number") within the "General Settings" tab is deselected and disabled.
Enable Fee_GetWriteCycle API	--	ON **OFF**	Preprocessor switch to enable/disable the existence of the API service `Fee_GetWriteCycle()`.
Enable Fee_ForceSectorSwitch API	--	ON **OFF**	Preprocessor switch to enable/disable the existence of the API service `Fee_ForceSectorSwitch()`.
Enable Data Conversion API	--	ON **OFF**	Preprocessor switch to enable/disable the existence of `Fee_ConvertDataBlocks`. **Info** If this feature was disabled at delivery time, this switch is still available, but it is disabled. It cannot be used to enable the feature “Data Conversion”.
Enable API to allow/prohibit FSS	--	ON **OFF**	Preprocessor switch to enable/disable function set `Fee_EnableFss()`/`Fee_DisableFss()` Refer to chapter 2.10.

Table 5-9 API Configuration

5.1.7.2 Provided API

Attribute Name	Value Type	Values The default value is written in bold	Description
Provided API	--	--	This group shows the API services which are currently provided by the FEE depending on its configuration.

Table 5-10 Provided API

5.2 Configuration Parameters only visible in GCE

There are parameters that cannot be configured using the comfort view of DaVinci Configurator. These parameters should only be modified, if really necessary, and if the user knows about their effects. These parameters' default values are usable in very most cases.

To meet platform specific requirements and/or restrictions, they may also be pre-configured during integration/delivery, i.e. they might not be changeable at all.

Attribute Name	Value Type	Values The default value is written in bold	Description
FeeMaxLinkTableSize (Maximum Number of LinkTable Entries)	Integer	0 … **4095**	Defines the maximum size of the Link Table. The link table itself improves performance of FEE, but it reduces space available for user data. Thus it might be desirable to limit its size. Especially on devices with very limited flash space and/or considerably large pages (e.g. TriCore TC179x) the performance penalty resulting from reducing the link table size (or disabling it completely, i.e. by setting size to 0), would be feasible.

Table 5-11 Parameters only visible in GCE view.

5.2.1 Fls API deviating from AUTOSAR naming convention

FEE is able to use any AUTOSAR compliant Fls "out of the box", i.e. associating a partition with a driver, its API is defined and FEE can be generated. This is the case for all internal device drivers whose API shall exactly match the naming given in Flash Driver SWS. It is also true for an external device driver, which shall include Vendor ID and a so-called "API infix"; both shall be published in its module description file.

However, sometimes an Fls doesn't fully comply, or there might be technical reasons to deviate from AUTOSAR.

Example
Due to address space limitiation mentioned in chapter 2.1, it is necessary to use some "proxy Fls", if FEE (a partition) shall address an Fls's sectors beyond 2GB limit (≥ 0x80000000). Using this Feature of FEE, its function names do not need to comply with AUTOSAR.

In such a case FEE would not compile (or not link) because used function names do not match Fls's provided ones. This can be solved as follows:

1. Add container `FeeFlsApi` to container `FeeGeneral`.
2. New Container may be renamed
3. Enter name of Fls's include file
4. Change function names according to your needs. Parameter names are self-explaining; they map from SWS function names (e.g. `Fls_Read`) to actually implemented names (e.g. `MyVeryOwn_ReadFunction`).
5. Associate this container with an existing Fls configuration (`FeeFlsDeviceIndex` is a reference to a container of type `Fls/FlsGeneral`).

Expert Knowledge

Container `FeeFlsApi` has a multiplicity of 0..*, i.e. you may create an arbitrary number of Flses. Generator matches references to Fls drivers with partitions' references to Fls (runtime) configurations to generate necessary information of drivers which are actually used. Of course, each Fls driver instance must be referenced at most once.

6 AUTOSAR Standard Compliance

6.1 Deviations

6.1.1 Maximum Blocking Time

The parameter `FEE_MAXIMUM_BLOCK_TIME` is not supported by the current version of the FEE, because a time reference is missing to support this requirement.

6.2 Additions/ Extensions

6.2.1 Parameter Checking

The internal parameter checks of the API functions can be en-/disabled separately. The AUTOSAR standard requires en-/disabling of the complete parameter checking only. For details see chapter 2.6.1.

6.2.2 Fee_InitEx

See chapter 4.3.2 for further information.

6.2.3 GetEraseCycle

See chapter 2.5.1.6 and 4.3.13 for further information.

6.2.4 GetWriteCycle

See chapter 2.5.1.7 and 4.3.14 for further information.

6.2.5 GetSectorSwitchStatus

See chapter 4.3.15 for further information.

6.2.6 ForceSectorSwitch

See chapter 0 for further information.

6.2.7 Fee_ConvertBlockConfig

This Service, and all related types, settings, etc. add the capability of performing data conversion after a configuration update. Note that this extension is optional; it must be ordered explicitly.
For more information refer to chapter 2.8.

6.2.8 Fee_SuspendWrites / Fee_ResumeWrites

See chapters 2.10, 4.3.18 and 4.3.19 for further information.

6.2.9 Fee_EnableFss / Fee_DisableFss

See chapters 2.10, 4.3.20 and 4.3.21 for further information.

6.3 Limitations

AUTOSAR does not specify how a FEE implementation shall organize flash memory. Additionally there are no requirements on performance or robustness. Following restrictions result from postulating and implementing such requirements.

6.3.1 Partitions

Current implementation limits the maximum number of partitions to 4.

6.3.2 Flash Usage

Fee cannot provide whole configured flash memory for user data storage.

Net payload to be stored in flash memory is roundabout 25% of assigned flash memory, or more precisely, 50% of the smaller logical sector's size.

In addition to memory overhead caused by internal management information to be stored along with user data, and to overhead caused by flash space to be reserved for dynamic allocation (in order to process write requests), the FEE must adhere to HW's alignment requirements. Usually this alignment is determined by flash's smallest writable entity, i.e. the flash page size. However, on some platforms, alignment must be even more stringent, because read and write accesses must be secured, so that they don't endanger already stored information.

This means: it might be possible to add stuffing bytes carrying no information. Depending on flash device and related alignment requirements, this overhead might become significant, additionally reducing the total amount of user data that may be stored in flash.

6.3.3 Performance

Due to dynamic flash allocation, dependent on configuration and actual write accesses, the performance, by means of request processing time (i.e. number of Fee_MainFunction call cycles), fluctuates. Starting with an empty flash, job processing time increases over time up to a maximum search effort (one logical sector is completely filled). When the other sector is initially used, search efforts reduce again.

Additionally worst case blocking times are mostly affected by clean-up operation (copy recent data instances) and sector erase. Latter one is determined by HW; usually erase timings are at scales of hundreds of milliseconds per 16kBytes. Their frequency highly depends on flash usage (data amount as well as write frequency).

Expert Knowledge
The worst case – a complete sector switch that includes copying all data and erase the logical sector – is a very exceptional case, as it would mean that nothing was successfully copied so far, but flash is full, i.e. erase is necessary in order to remain writeable.

6.3.4 Aborts/Resets

Aborts/resets during write operations result in artifacts in flash memory, increasing flash usage. Frequent resets may prevent FEE from completing a clean-up operation (Sector Switch), but cause much wasted space due to the artifacts.

Finally the partition may overflow. There is no space left to copy data from one logical sector to the other one, which is the precondition to perform a sector erase without any recent data instance. This overflow may be reported by FEE, and an erase may be executed, accepting the risk of data loss. For more information, refer to chapter 3.3.5.3.

Additionally the FEE allows marking data blocks as essential for ECU's operation; they must never be lost (chapter 2.11).

Finally, FEE provides services for handling under-voltage situations in safer ways; see chapter 2.10.

6.3.5 Write Cycle and Erase Cycle Counters

Blocks' Write Cycle and Sectors' Erase Counters allow monitoring flash usage over time. Especially, they allow re-evaluating initial estimations of blocks write cycles.

Erase cycle counters can be used to estimate total flash usage over ECU's life-time. A key parameter is the number of erase cycles per physical sector, which is limited on Flash EEPROM devices.

Precisions of both kinds of cycle counters are limited. Especially interrupted accesses and/or resets reduce precision, for certain reasons.

Caution
Blocks' Write Cycle and Sectors' Erase Cycle counters shall be used for statistical purposes only. Especially, they shall not be used to affect code execution in any way.

6.3.6 Fls Page sizes

FEE does only support Flash devices (FLS implementations) with "usual" accessing schemes, i.e. any programming operation (addresses and lengths) must be based on integer powers of 2.

Read access must be possible in a byte-wise fashion (this is also required by AUTOSAR).

To be usable by FEE an FLS implementation has to consider this restriction, even if the HW itself does not adhere to it. That means, in such cases the FLS has to provide a suitable abstraction.

6.3.7 Fee_ForceSectorSwitch and unformatted Partitions

If `Fee_ForceSectorSwitch` (refer to chapter 4.3.16) is requested on a Flash containing a unformatted partitions (i.e. both logical sector headers are erased), FEE performs three sector erases on those partitions in total. There is one redundant erase, but it does not notably affect flash's endurance. Since the whole situation is very exceptional (usually only initially), we did not add special handling.

7 Glossary and Abbreviations

7.1 Glossary

Term	Description
DaVinci Configurator	Tool to create a consistent and optimized ECU configuration. Generation and validation tool for MICROSAR components.
Block Tag	Identifies a block in data flash. See also: Block Id
Block Id	Identifier used to uniquely identify blocks in a partition. Stored in data flash, it consists of Block Tag and Data Index. In conjunction with partition ID, every block in FEE can be uniquely identified. The Block Id abstracts from 16Bit Block Number passed via API; it depends on current configuration, in particular on Dataset Selection Bits. The block tag is one cornerstone in FEE's update capability: As long as a Block Id does not change across configuration updates, it can be found in data flash, data may be kept, if block length also remained unchanged.

Table 7-1 Glossary

7.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
ECUM	ECU Manager
EEPROM	Electrically Erasable Programmable Read Only Memory
FBL	Flash Bootloader
FEE	Flash EEPROM Emulation Module
FLS	Flash Driver
HIS	Hersteller Initiative Software
MEMIF	Memory Abstraction Interface Module
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
NVM	NVRAM Manager
NVRAM	Non Volatile Random Access Memory
SchM	(AUTOSAR) Scheduling Manager

SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 7-2 Abbreviations

8 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector-informatik.com