VECTOR

MICROSAR FLS

Technical Reference

MCAL Emulation in VTT

Version 1.2.0

Authors	Christian Leder, Bethina Mausz
Status	Released

Document Information

History

Author	Date	Version	Remarks
Peter Lang	2013-10-05	1.00.00	Creation of document
Christian Leder	2014-05-23	1.00.01	> Description of function `Fls_JobResult()` corrected. > New ASR architecture figure added.
Christian Leder	2015-02-05	1.01.00	> Global renaming of Vip to Vtt > Usage of template 5.11.0 for the Technical reference
Bethina Mausz	18.06.2016	1.02.00	> FEAT-1842, support of external driver. Restriction: Only one driver per system.

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_SWS_FlashDriver.pdf	V3.2.0
[2]	AUTOSAR	AUTOSAR_SWS_DevelopmentErrorTracer.pdf	V3.2.0
[3]	AUTOSAR	AUTOSAR_SWS_DiagnosticEventManager.pdf	V4.2.0
[4]	AUTOSAR	AUTOSAR_TR_BSWModuleList.pdf	V1.6.0

Caution
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

Illustrations

Tables

1 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
1.0.x	Initial version of the Vip FLS driver
2.0.x	Global renaming of Vip to Vtt

Table 1-1 Component history

2 Introduction

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

Supported AUTOSAR Release*:	4	
Supported Configuration Variants:	pre-compile	
Vendor ID:	FLS_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	FLS_MODULE_ID	092 decimal (according to ref. [4])

* For the detailed functional specification please also refer to the corresponding AUTOSAR SWS.

The internal Flash driver offers asynchronous memory services. The flash driver does not buffer data to be read or written. It uses application data buffers that are referenced by a pointer passed via the API.

In this emulated driver, the contents of the flash memory are written to and read from a text file on the PC (File extension nvram).

The main tasks of the FLS driver are:

- Handle read, write, erase and optional compare functionality
- User requested status reporting
- Automatic status reporting via callbacks
- Error handling.

2.1 Architecture Overview

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

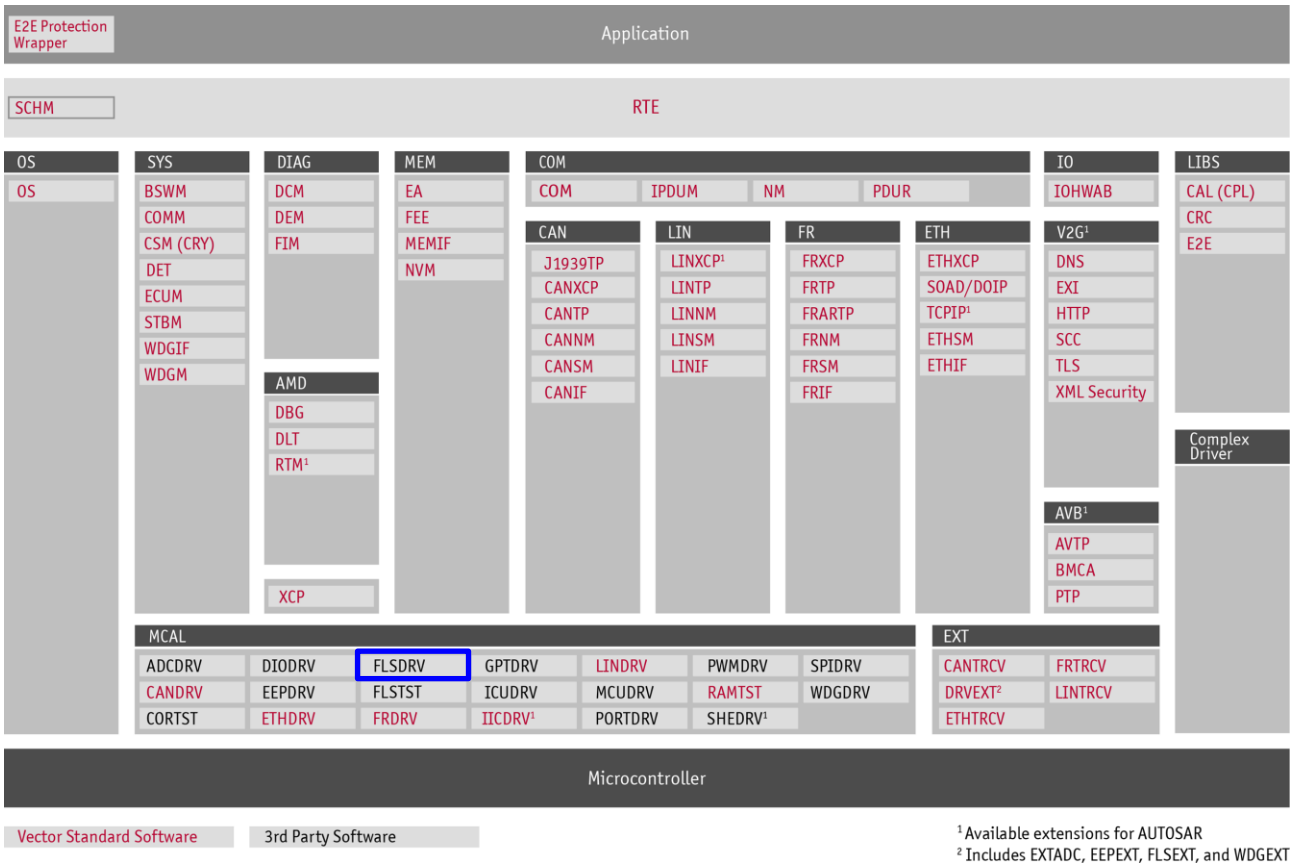

Figure 2-1 AUTOSAR 4.x Architecture Overview

The next figure shows the interfaces to adjacent modules of the FLS. These interfaces are described in chapter 5.

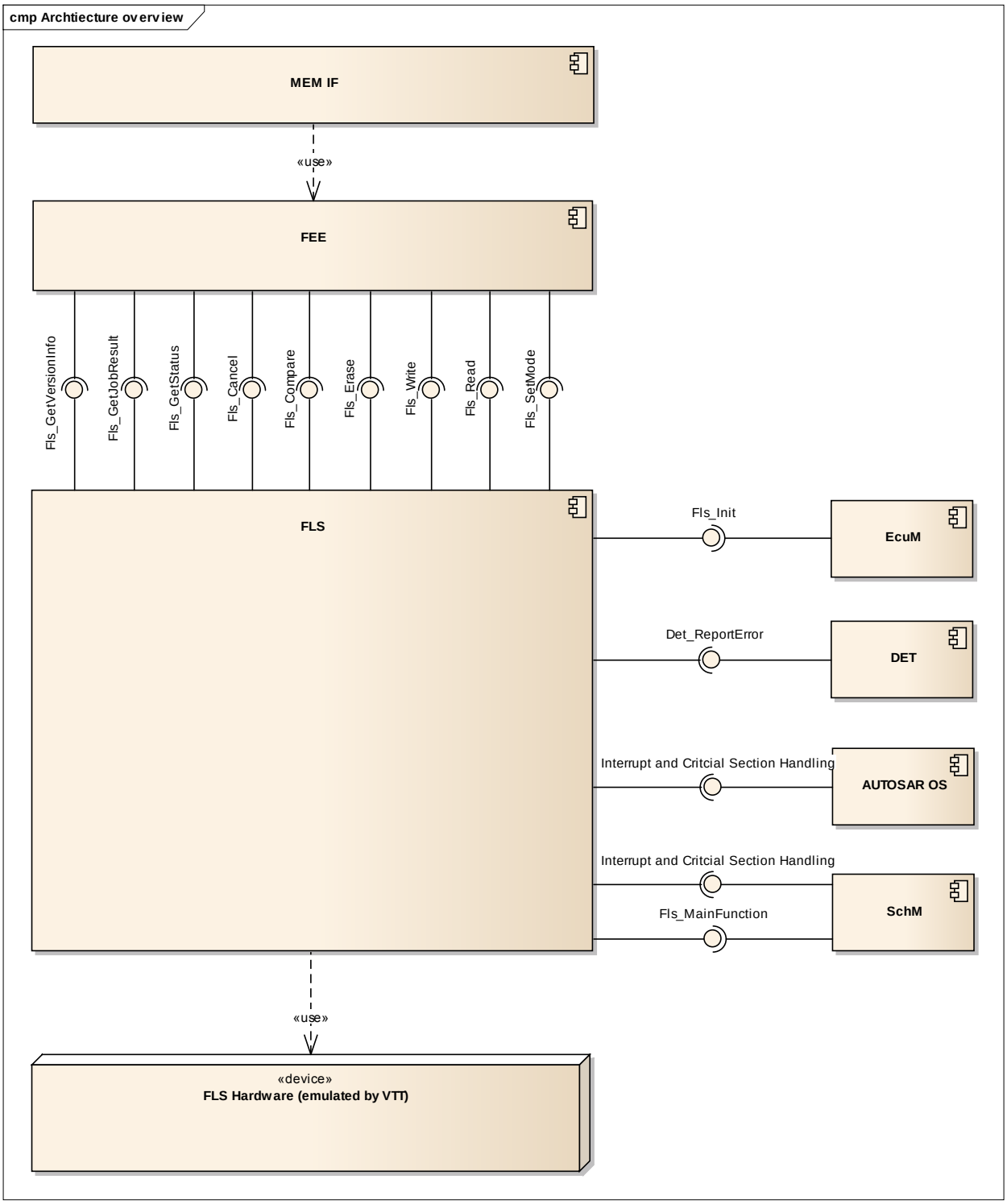

Figure 2-2 Interfaces to adjacent modules of the FLS

3 Functional Description

3.1 Features

The features listed in the following tables cover the complete functionality specified for the FLS.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 3-1 Supported AUTOSAR standard conform features

> Table 3-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further FLS functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

> Table 3-3 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
Asynchronous service for reading data from FLASH
Asynchronous service for writing data to FLASH
Asynchronous service for erasing data from FLASH
Asynchronous service for comparing FLASH data with data from memory (e.g. RAM)
Asynchronous service for reading data from FLASH

Table 3-1 Supported AUTOSAR standard conform features

3.1.1 Deviations

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

Not Supported AUTOSAR Standard Conform Features
Flash access code cannot be loaded into RAM and executed
Flash erase code cannot be loaded into RAM and executed
The verify checks for development mode are not implemented
The sector protection is not realized because only Data Flash can be used with Fee
Only polling mode is supported.

Table 3-2 Not supported AUTOSAR standard conform features

3.1.2 Additions/ Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
In addition to the existing checks required by the AUTOSAR standard, the parameter versioninfo passed to the service `Fls_GetVersionInfo()` is checked for not referencing `NULL_PTR`. If it does, the error `FLS_E_PARAM_VINFO` is reported to DET instead of `FLS_E_PARAM_POINTER`

Features Provided Beyond The AUTOSAR Standard
Two additional APIs are implemented within the emulated driver: > Fls_Copy > Fls_ReadSync

Table 3-3 Features provided beyond the AUTOSAR standard

3.1.3 Limitations

3.1.3.1 Diagnostic Event Manager

Due to the fact that the FLS is emulated, reporting of hardware errors to the DEM is not supported. Because of compatibility reasons, the DEM has to be configured in DaVinci Configurator.

3.2 Emulation

This driver is an emulation of an FLS module.

Caution

Be careful using while loops in order to poll any status.

The user has to ensure, that the application does not block the emulation. So, within every while loop the following function call has to be called:

```
while(ANY_STATUS == temp_status)
{
  Schedule();
}
```

Use the function call Schedule() which is available once the header file of the module FLS is included.

3.3 Initialization

The FLS module is being initialized by calling `Fls_Init(&FlsConfigSet)`. All global variables are initialized by calling `Fls_InitMemory()`. So, `Fls_InitMemory()` has to be called prior to `Fls_Init()`.

3.4 States

3.4.1 Module States

The module FLS provides the following global states:

- `MEMIF_UNINIT`: FLS is not initialized
- `MEMIF_IDLE`: Currently no active read-, write-, erase- or compare-job
- `MEMIF_BUSY`: Read-, write-, erase- or compare-job is ongoing

3.4.2 Job States

The FLS provides the following job states:

- `MEMIF_JOB_OK`: Job finished successfully
- `MEMIF_JOB_CANCELLED`: Fls_Cancel() has been called
- `MEMIF_JOB_FAILED`: Fls_SimulateError() has been called
- `MEMIF_BLOCK_INCONSISTENT`: Compare job detected inconsistencies.

3.5 Main Functions

`Fls_MainFunction` has to be called cyclically for processing read-, write-, compare- or erase-jobs.

3.6 Error Handling

3.6.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [2], if development error reporting is enabled (i.e. pre-compile parameter `FLS_DEV_ERROR_DETECT==STD_ON`).

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()`.

The reported FLS ID is 092.

The reported service IDs identify the services which are described in 5.2. The following table presents the service IDs and the related services:

Service ID	Service
0x00	Fls_Init
0x01	Fls_Erase
0x02	Fls_Write
0x03	Fls_Cancel
0x04	Fls_GetStatus
0x05	Fls_GetJobResult
0x06	Fls_MainFunction
0x07	Fls_Read

Service ID	Service
0x08	Fls_Compare
0x09	Fls_SetMode
0x10	Fls_GetVersionInfo
0x20	Fls_Copy
0x21	Fls_ReadSync

Table 3-4 Service IDs

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

Error Code		Description
0x01	FLS_E_PARAM_CONFIG	Pointer to the configuration set is referencing NULL_PTR
0x02	FLS_E_PARAM_ADDRESS	Address out of range.
0x03	FLS_E_PARAM_LENGTH	Length out of range
0x04	FLS_E_PARAM_DATA	The pointer to data is a NULL_PTR
0x05	FLS_E_UNINIT	Driver was not initialized
0x06	FLS_E_BUSY	Driver is busy
0x15	FLS_E_PARAM_VINFO	The version info pointer is a NULL_PTR
0x20	FLS_E_ADDRESS_OVERLAP	Source- and Targetaddress overlap at copying process

Table 3-5 Errors reported to DET

3.6.1.1 Parameter Checking

AUTOSAR requires that API functions check the validity of their parameters. The checks in Table 3-6 are internal parameter checks of the API functions. These checks are for development error reporting and can be en-/disabled.

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

Service \ Check	FLS_E_PARAM_ADDRESS	FLS_E_PARAM_LENGTH	FLS_E_PARAM_DATA	FLS_E_UNINIT	FLS_E_BUSY	FLS_E_PARAM_VINFO	FLS_E_PARAM_CONFIG	FLS_E_ADDRESS_OVERLAP
Fls_Init					■		■	
Fls_Erase		■		■	■			
Fls_Write	■		■	■	■			

Service \ Check	FLS_E_PARAM_ADDRESS	FLS_E_PARAM_LENGTH	FLS_E_PARAM_DATA	FLS_E_UNINIT	FLS_E_BUSY	FLS_E_PARAM_VINFO	FLS_E_PARAM_CONFIG	FLS_E_ADDRESS_OVERLAP
Fls_Cancel								
Fls_GetStatus								
Fls_GetJobResult								
Fls_Mainfunction								
Fls_Read	■	■	■	■	■			
Fls_Compare	■	■	■	■	■			
Fls_SetMode			■	■	■			
Fls_GetVersionInfo						■		
Fls_Copy	■	■		■	■			■
Fls_ReadSync	■	■	■	■	■			

Table 3-6 Development Error Reporting: Assignment of checks to services

3.6.2 Production Code Error Reporting

Info
Production errors are not supported in this emulation.

4 Integration

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

4.1 Scope of Delivery

The delivery of the FLS contains the files which are described in the chapters 4.1.1 and 4.1.2:

4.1.1 Static Files

File Name	Description
Fls.h	The module header defines the interface of the FLS. This file must be included by upper layer software components
Fls.c	This C-source contains the implementation of the module's functionalities
DrvFls_VttCanoe01Asr.jar	This jar-file contains the generator and the validator for the DaVinci Configurator
VTTFls_bswmd.arxml	Basic Software Module Description according to AUTOSAR for VTT Emulation
Fls_bswmd.arxml	Optional Basic Software Module Description. Placeholder for real target (semiconductor manufacturer) in VTT only use case

Table 4-1 Static files

4.1.2 Dynamic Files

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

File Name	Description
Fls_Cfg.h	The configuration-header contains the static configuration part of this module
Fls_PBcfg.c	The configuration-source contains the object independent part of the runtime configuration
Fls_VendorId_ApiInfix.c	The source contains the wrapper APIs which maps the vendor/infix specific APIs to VTTFls APIs. This file is generated in case an API-Infix is configured and the Infix PMU is not used, because in this case the wrapper files are otherwise available (*refer to: TechnicalReference_Fls_VTT_AddOn_AURIX_FeeUsage.pdf*).
Fls_VendorId_ApiInfix.h	The header contains the vendor/infix specific API declarations. It also contains the extern declaration of `Fls_MainFunction`. This file is generated in case an API-Infix is configured and the Infix PMU is not used, because in this case the wrapper files are otherwise available (*refer to: TechnicalReference_Fls_VTT_AddOn_AURIX_FeeUsage.pdf*).

Table 4-2 Generated files

4.2 Include Structure

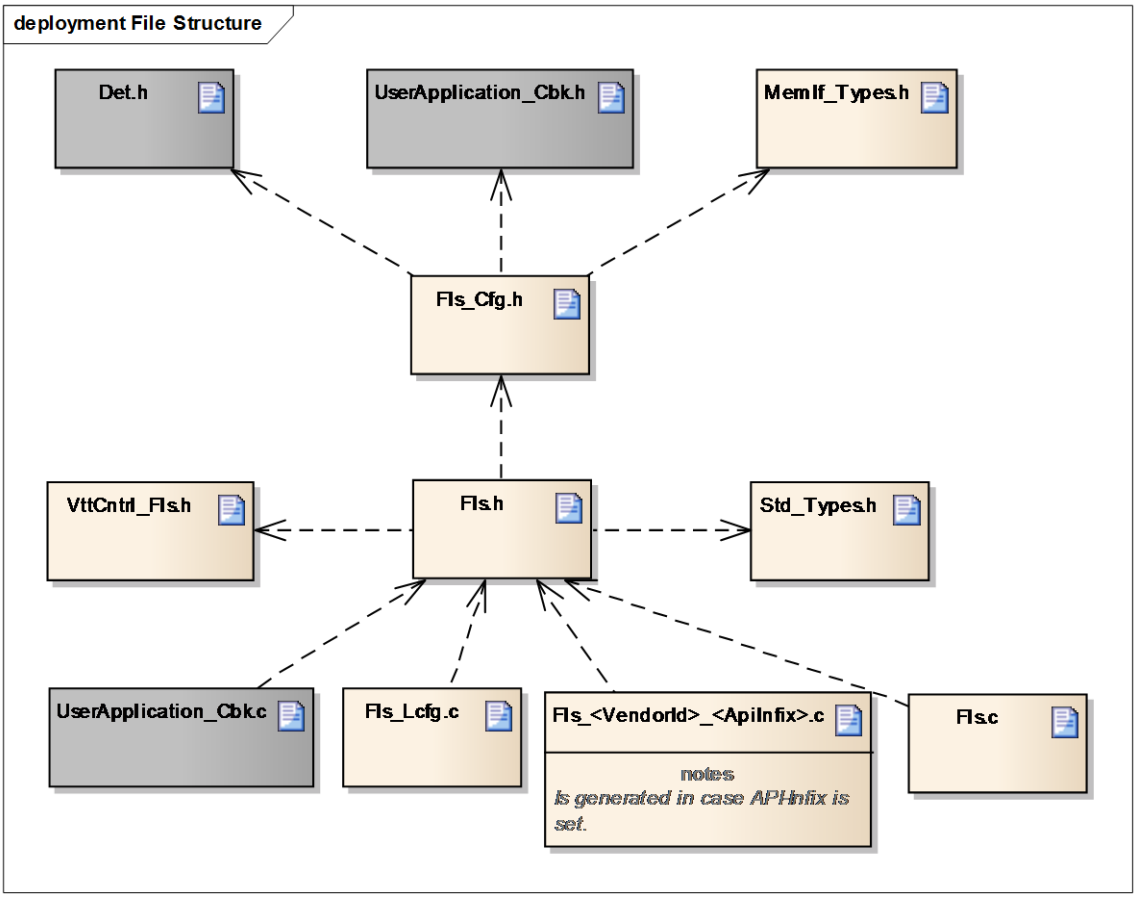

Figure 4-1 Include Structure

4.3 Dependencies on SW Modules

4.3.1 AUTOSAR OS (Optional)

An operating system can be used for task scheduling, interrupt handling, global suspend and restore of interrupts and creating of the Interrupt Vector Table.

4.3.2 DET (Optional)

The FLS module depends on the DET (by default) in order to report development errors. Detection and reporting of development errors can be enabled or disabled by the switch "Enable Development Error Detection".

4.3.3 SchM (Optional)

Beside the AUTOSAR OS the Schedule Manager provides functions that module FLS calls at begin and end of critical sections. Besides, the Schedule Manager is responsible for calling the main functions.

4.3.4 EcuM (Optional)

The EcuM cares for the initialization of the module FLS.

5 API Description

For an interfaces overview please see Figure 2-2.

5.1 Type Definitions

5.1.1 FLS types

The types defined by the FLS are described in this chapter.

Type Name	C-Type	Description	Value Range
`Fls_AddressType`	`uint32`	Used as address offset from the configured FLASH base address to access a certain FLASH memory area.	0 … 20971520
`Fls_LengthType`	`uint32`	This type specifies the number of bytes to read/write/erase/compare .	0 … 20971520

Table 5-1 Type definitions

5.1.2 Imported Types

The following types are imported from the module MemIf.

Type Name	Reference
MemIf_StatusType	Defined in MemIf_Types.h
MemIf_JobResultType	Defined in MemIf_Types.h
MemIf_ModeType	Defined in MemIf_Types.h

Table 5-2 Imported Types

5.2 Services provided by FLS

5.2.1 Fls_InitMemory

Prototype	
`void` **`Fls_InitMemory`** `(void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
This service initializes the global variables in case the startup code does not work	
Particularities and Limitations	
> This function is synchronous. > This function is non-reentrant. > Module must not be initialized	
Expected Caller Context	
> Called during startup	

Table 5-3 Fls_InitMemory

5.2.2 Fls_Init

Prototype	
`void` **`Fls_Init`** `(P2CONST(Fls_ConfigType, AUTOMATIC, FLS_APPL_CONST) ConfigPtr)`	
Parameter	
`ConfigPtr`	Pointer to the configuration struct of the FLS
Return code	
-	-
Functional Description	
This service initializes the module FLS. The FLASH driver state is set to `MEMIF_IDLE` and the job status is set to `MEMIF_JOB_OK`.	
Particularities and Limitations	
> This function is synchronous. > This function is non re-entrant. > Module must not be initialized.	
Expected Caller Context	
> ECU State Manager or comparable software module, responsible for driver initialization after startup.	

Table 5-4 Fls_Init

5.2.3 Fls_Erase

Prototype	
`Std_ReturnType Fls_Erase ( MemIf_AddressType TargetAddress, MemIf_LengthType Length )`	
Parameter	
`TargetAddress`	Address in FLASH memory, to which data should be erase Min: 0 Max: `FLS_TOTAL_SIZE - 1`
`Length`	Amount of bytes to erase Min: 1 Max: `FLS_TOTAL_SIZE - TargetAddress`
Return code	
`Std_ReturnType`	`E_OK`, success. `E_NOT_OK`, fail or request not accepted.
Functional Description	
This service requests an erase job, that is, the job's data (passed as parameters) is stored internally and the service returns. The job itself is processed asynchronously by executing `Fls_MainFunction()` cyclically.	
Particularities and Limitations	
> This function is asynchronous. > This function is non reentrant. > This service may only be called if the module has been initialized before. > This service may only be called while the module is in state `MEMIF_IDLE`.	
Expected Caller Context	
> Task context	

Table 5-5 Fls_Erase

5.2.4 Fls_Write

Prototype
`Std_ReturnType Fls_Write ( MemIf_AddressType TargetAddress, const uint8* SourceAddressPtr, MemIf_LengthType Length )`

Parameter	
TargetAddress	Address in FLASH memory, to which data should be written Min: 0 Max: FLS_TOTAL_SIZE - 1
SourceAddressPtr	Reference to the buffer whose data shall be written
Length	Amount of data in bytes to write Min: 1 Max: FLS_TOTAL_SIZE - TargetAddress
Return code	
Std_ReturnType	E_OK, success. E_NOT_OK, fail or request not accepted.
Functional Description	
This service requests a write job, that is, the job's data (passed as parameters) is stored internally and the service returns. The job itself is processed asynchronously by executing Fls_MainFunction() cyclically.	
Particularities and Limitations	
> This function is asynchronous. > This function is non reentrant. > This service may only be called if the module has been initialized before. > This service may only be called while the module is in state MEMIF_IDLE.	
Expected Caller Context	
> Task context	

Table 5-6 Fls_Write

5.2.5 Fls_Cancel

Prototype	
void **Fls_Cancel** (void)	
Parameter	
-	-
Return code	
-	-
Functional Description	
This service allows cancelling a currently processed job synchronously. New jobs can be requested right after this service has returned. In case no job is pending, the service is left without further action. A pending job is cancelled and the error notification is called (synchronously), if configured. Data sets may be incomplete, if a job is aborted.	

Particularities and Limitations
> This function is synchronous.
> This function is non reentrant.
> This function is configurable.
> This service may only be called if the module has been initialized before.
Expected Caller Context
> Task context

Table 5-7 Fls_Cancel

5.2.6 Fls_GetStatus

Prototype	
`MemIf_StatusType` **`Fls_GetStatus`** `(void)`	
Parameter	
-	-
Return code	
`MemIf_StatusType`	`MEMIF_UNINIT`, module has not been initialized before `MEMIF_IDLE`, no job is processed currently `MEMIF_BUSY`, module is busy processing a job
Functional Description	
This function returns the current status of the driver.	
Particularities and Limitations	
> This function is synchronous.	
> This function is reentrant.	
> This function is configurable.	
Expected Caller Context	
> Task context	

Table 5-8 Fls_GetStatus

5.2.7 Fls_GetJobResult

Prototype	
`MemIf_JobResultType` **`Fls_GetJobResult`** `(void)`	
Parameter	
-	-

Return code	
MemIf_JobResultType	MEMIF_JOB_OK, last processed job has finished successfully MEMIF_JOB_FAILED, last processed job has finished with errors. MEMIF_JOB_PENDING, job is being processed currently MEMIF_JOB_CANCELED, last processed job has been cancelled by Fls_Cancel() MEMIF_BLOCK_INCONSISTENT, last processed (compare) job has finished successfully, but data did not match
Functional Description	
This service returns information about the result of the latest or currently processed job.	
Particularities and Limitations	
> This function is synchronous. > This function is reentrant. > This service may only be called if the module has been initialized before. > In case module FLS has not been initialized before, this service will return MEMIF_JOB_OK. > This function is configurable.	
Expected Caller Context	
> No restrictions	

Table 5-9 Fls_GetJobResult

5.2.8 Fls_Read

Prototype	

Std_ReturnType **Fls_Read**
(
  MemIf_AddressType SourceAddress,
  uint8* TargetAddressPtr,
  MemIf_LengthType Length
)	

Parameter	
SourceAddress	Address in FLASH memory, from which data should be read Min: 0 Max: FLS_TOTAL_SIZE - 1
TargetAddressPtr	Reference to the buffer to which the read data shall be copied
Length	Amount of data in bytes to read Min: 1 Max: FLS_TOTAL_SIZE - SourceAddress
Return code	
Std_ReturnType	E_OK, success. E_NOT_OK, fail or request not accepted.

Functional Description
This service requests a read job, that is, the job's data (passed as parameters) is stored internally and the service returns. The job itself is processed asynchronously by executing `Fls_MainFunction()` cyclically.
Particularities and Limitations
> This function is asynchronous. > This function is non reentrant. > This service may only be called if the module has been initialized before. > This service may only be called while the module is in state `MEMIF_IDLE`.
Expected Caller Context
> Task context

Table 5-10 Fls_Read

5.2.9 Fls_Compare

Prototype

```
Std_ReturnType Fls_Compare
(
  Fls_AddressType SourceAddress,
  const uint8* TargetAddressPtr,
  Fls_LengthType Length
)
```

Parameter	
`SourceAddress`	Address in FLASH memory, whose data should be compared Min: 0 Max: `FLS_TOTAL_SIZE - 1`
`TargetAddressPtr`	Reference to the buffer whose data shall be compared to FLASH
`Length`	Amount of data in bytes to compare Min: 1 Max: `FLS_TOTAL_SIZE - SourceAddress`
Return code	
`Std_ReturnType`	`E_OK`, success. `E_NOT_OK`, fail or request not accepted.
Functional Description	
This service requests a compare job, that is, the job's data (passed as parameters) is stored internally and the service returns. The job itself is processed asynchronously by executing `Fls_MainFunction()` cyclically.	

Particularities and Limitations
> This function is asynchronous.
> This function is non reentrant.
> This service may only be called if the module has been initialized before.
> This service may only be called while the module is in state `MEMIF_IDLE`.
> This function is configurable.
Expected Caller Context
> Task context

Table 5-11 Fls_Compare

5.2.10 Fls_SetMode

Prototype	
`void` **`Fls_SetMode`** `(MemIf_ModeType Mode)`	
Parameter	
`Mode`	Switch module FLS to this job processing mode. Valid values are `MEMIF_MODE_FAST` and `MEMIF_MODE_SLOW`.
Return code	
-	-
Functional Description	
This service switches to the job processing mode passed in parameter Mode. This mode determines the amount of bytes processed in the execution of `Fls_MainFunction()`.	
Particularities and Limitations	
> This function is synchronous.	
> This function is non reentrant.	
> This service may only be called if the module has been initialized before.	
> This service may only be called while the module is in state `MEMIF_IDLE`.	
> This function is configurable	
Expected Caller Context	
> Task context	

Table 5-12 Fls_SetMode

5.2.11 Fls_GetVersionInfo

Prototype

`void` **`Fls_GetVersionInfo`**
`(`
`  P2VAR(Std_VersionInfoType, AUTOMATIC, FLS_APPL_DATA) versioninfo`
`)`

Parameter	
`versioninfo`	Reference to the structure to which the information should be written
Return code	
-	-
Functional Description	
This function returns the version information of the module. The version information includes: > Module Id > Vendor Id > Software version numbers	
Particularities and Limitations	
> This function is synchronous. > This function is non reentrant. > This function is configurable.	
Expected Caller Context	
> Task context	

Table 5-13 Fls_GetVersionInfo

5.2.12 Fls_MainFunction

Prototype	
`void` **`Fls_Mainfunction`** `(void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
This service has to be executed periodically for asynchronous job processing. The amount of data being processed within one call to this service is limited by the configured read- and write-sizes depending on the current mode (`MEMIF_MODE_SLOW` or `MEMIF_MODE_FAST`). After all data of a job has been processed completely, job end notification is called, or, if errors occurred, job error notification is performed.	
Particularities and Limitations	
> Module has to be initialized before	
Expected Caller Context	
> This function should be executed periodically e.g. by the Basic Software Scheduler (SchM)	

Table 5-14 Fls_MainFunction

5.2.13 Fls_ReadSync

Prototype	

`MemIf_JobResultType` **`Fls_ReadSync`**
`(`
`  MemIf_AddressType FlsAddress,`
`  uint8 * DataBufferPtr,`
`  MemIf_LengthType Length`
`)`	

Parameter	
`FlsAddress`	Address in FLASH memory, from which data should be read Min: 0 Max: `FLS_TOTAL_SIZE - 1`
`DataBufferPtr`	Reference to the buffer to which the read data shall be copied
`Length`	Amount of data in bytes to read Min: 1 Max: `FLS_TOTAL_SIZE - FlsAddress`
Return code	
`Std_ReturnType`	`E_OK`, success. `E_NOT_OK`, fail or request not accepted.
Functional Description	
This function uses the given address, pointer and length to read data from flash synchronously.	
Particularities and Limitations	
> This function is synchronous. > This function is reentrant. > This service may only be called if the module has been initialized before. > This service may only be called while the module is in state `MEMIF_IDLE`. > This function is configurable.	
Expected Caller Context	
> Task context	

Table 5-15 Fls_ReadSync

5.2.14 Fls_Copy

Prototype	

`Std_ReturnType` **`Fls_Read`**
`(`
`  Fls_AddressType TargetAddress,`
`  Fls_AddressType SourceAddress,`
`  Fls_LengthType Length`
`)`	

Parameter	
`TargetAddress`	Address in FLASH memory, to which data should be written Min: 0 Max: `FLS_TOTAL_SIZE - 1`
`SourceAddress`	Reference to the buffer whose data shall be copied
`Length`	Amount of data in bytes to write
Return code	
`Std_ReturnType`	`E_OK`, success. `E_NOT_OK`, fail or request not accepted.
Functional Description	
This function uses the given source address to read data from flash and to write the data to the given target address. The given length indicates the amount of bytes to copy.	
Particularities and Limitations	
> This function is asynchronous. > This function is non reentrant. > This service may only be called if the module has been initialized before. > This service may only be called while the module is in state `MEMIF_IDLE`. > This function is configurable.	
Expected Caller Context	
> Task context	

Table 5-16 Fls_Copy

5.2.15 Fls_SimulateError

Prototype	
`Std_ReturnType` **`Fls_SimulateError`** `(void)`	
Parameter	
-	-
Return code	
`Std_ReturnType`	`E_OK`: Error has been successfully simulated `E_NOT_OK`: Error has not been simulated

Functional Description
This service is used for simulating errors. If it is called and a job is currently pending, then the job is aborted and the job result is set to `MEMIF_JOB_FAILED`. If no job is pending, function returns immediately (return code `E_NOT_OK`).
Particularities and Limitations
> Module has to be initialized before
Expected Caller Context
> Task Context

Table 5-17 Fls_SimulateError()

5.3 Services used by FLS

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

Component	API
DET	Det_ReportError

Table 5-18 Services used by the FLS

5.4 Configurable Interfaces

5.4.1 Notifications

At its configurable interfaces the FLS defines notifications that can be mapped to callback functions provided by other modules. The mapping is not statically defined by the FLS but can be performed at configuration time. The function prototypes that can be used for the configuration have to match the appropriate function prototype signatures, which are described in the following sub-chapters.

5.4.1.1 Job End Notification

Prototype	
`void` **`<JobEndNotificationName>`** `(void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
Job end notification functions have to adhere to this function prototype. The job end notification is called by `Fls_MainFunction()` when a job is finished successfully.	
Particularities and Limitations	
> This function is synchronous. > This function cannot be called from the API.	

Call context
> Task context on successful job execution.

Table 5-19 Job End Notification

5.4.1.2 Job Error Notification

Prototype	
`void <JobErrorNotificationName> (void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
The job error notification is called by `Fls_MainFunction()` when > a job can't be finished because of errors (i.e. a call of Fls`_SimulateError()` occurred) > a (compare) job finished successfully, but data did not match > a job is cancelled by `Fls_Cancel()`	
Particularities and Limitations	
> This function is synchronous. > This function cannot be called from the API.	
Call context	
> Task context on successful job execution.	

6 Configuration

6.1 Configuration Variants

The FLS supports the configuration variants

> VARIANT-PRE-COMPILE

The configuration classes of the FLS parameters depend on the supported configuration variants. For their definitions please see the VTTFls_bswmd.arxml file.

6.2 Configuration with DaVinci Configurator 5

The FLS module is configured with the help of the configuration tool DaVinci Configurator 5 (CFG5). The definition of each parameter is given in the corresponding BSWMD file.

7 Glossary and Abbreviations

7.1 Glossary

Term	Description
CANoe	Tool for simulation and testing of networks and electronic control units.
DaVinci Configurator	Configuration and generation tool for MICROSAR components

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 State Manager
FEE	Flash EEPROM Emulation
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
MemIf	Memory Interface
SchM	Basic Software Scheduler
VTT	vVIRTUALtarget

Table 7-2 Abbreviations

8 Contact

Visit our website for more information on

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

www.vector.com