Document Title	Specification of Module Flash Driver
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	025
Document Classification	Standard

Document Version	2.4.1
Document Status	Final
Part of Release	3.2
Revision	3

Document Change History

Date	Version	Changed by	Change Description
28.02.2014	2.4.1	AUTOSAR Release Management	• Editorial changes
17.05.2012	2.4.0	AUTOSAR Administration	• Links to sequence charts updated to generated artifacts
27.04.2011	2.3.0	AUTOSAR Administration	• Requirements for timeout supervision added / extended • Legal disclaimer revised
23.06.2008	2.2.2	AUTOSAR Administration	Legal disclaimer revised
23.01.2008	2.2.1	AUTOSAR Administration	Table formatting corrected
11.12.2007	2.2.0	AUTOSAR Administration	• NULL pointer check added to Fls_Compare • NULL pointer check detailed (in general) • Restriction removed to allow re-initialization of module • Tables in chapters 8 and 10 generated from UML model • Document meta information extended • Small layout adaptations made
14.02.2007	2.1.0	AUTOSAR Administration	• File include structure updated • Type usage corrected • Compare Job results adapted • API towards DEM corrected • Legal disclaimer revised • Release Notes added • "Advice for users" revised • "Revision Information" added

Document Change History

Date	Version	Changed by	Change Description
10.04.2006	2.0.0	AUTOSAR Administration	Document structure adapted to common Release 2.0 SWS Template • new functionality: Read, Compare and SetMode functions • scalability: functionality can be configured (on/off) • adapted to new MemHwA architecture
10.07.2004	1.0.0	AUTOSAR Administration	Initial release

Disclaimer

This specification and the material contained in it, as released by AUTOSAR is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the specification.

The material contained in this specification is protected by copyright and other types of Intellectual Property Rights. The commercial exploitation of the material contained in this specification requires a license to such Intellectual Property Rights.

This specification may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only.
For any other purpose, no part of the specification may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The AUTOSAR specifications have been developed for automotive applications only. They have neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Advice for users

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

1 Introduction and functional overview

This document specifies the functionality, API and the configuration of the AUTOSAR Basic Software module Flash Driver.

This specification is applicable to drivers for both internal and external flash memory.

The flash driver provides services for reading, writing and erasing flash memory and a configuration interface for setting / resetting the write / erase protection if supported by the underlying hardware.

In application mode of the ECU, the flash driver is only to be used by the Flash EEPROM emulation module for writing data. It is not intended to write program code to flash memory in application mode. This shall be done in boot mode which is out of scope of AUTOSAR.

A driver for an internal flash memory accesses the microcontroller hardware directly and is located in the Microcontroller Abstraction Layer. An external flash memory is usually connected via the microcontroller’s data / address busses (memory mapped access), the flash driver then uses the handlers / drivers for those busses to access the external flash memory device. The driver for an external flash memory device is located in the ECU Abstraction Layer.

FLS088: The functional requirements and the functional scope are the same for both types of drivers. Hence the API is semantically identical.

2 Acronyms and abbreviations

Abbreviation / Acronym:	***Description:***
DET	Development Error Tracer – module to which development errors are reported.
DEM	Diagnostic Event Manager – module to which production relevant errors are reported.
AC	(Flash) access code – abbreviation introduced to keep the names of the configuration parameters reasonably short.

Further definitions of terms used throughout this document

Term:	***Definition***
Flash sector	A flash sector is the smallest amount of flash memory that can be erased in one pass. The size of the flash sector depends upon the flash technology and is therefore hardware dependent.
Flash page	A flash page is the smallest amount of flash memory that can be programmed in one pass. The size of the flash page depends upon the flash technology and is therefore hardware dependent.
Flash access code	Internal flash driver routines called by the main function (job processing function) to erase or write the flash hardware.

3 Related documentation

3.1 AUTOSAR deliverables

[1] List of Basic Software Modules,
AUTOSAR_SoftwareModuleList.pdf

[2] Layered Software Architecture,
AUTOSAR_LayeredSoftwareArchitecture.pdf

[3] General Requirements on Basic Software Modules,
AUTOSAR_SRS_General.pdf

[4] General Requirements on SPAL,
AUTOSAR_SRS_SPAL_General.pdf

[5] Requirements on Flash Driver
AUTOSAR_SRS_Flash_Driver.pdf

[6] Requirements on Memory Hardware Abstraction Layer,
AUTOSAR_SRS_MemHW_AbstractionLayer.pdf

[7] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[8] AUTOSAR Basic Software Module Description Template,
AUTOSAR_BSW_Module_Description.pdf

3.2 Related standards and norms

[9] HIS Flash Driver Specification
HIS flash driver v130.pdf on
http://www.automotive-his.de/download/

4 Constraints and assumptions

4.1 Limitations

- The flash driver only erases or programs complete flash sectors respectively flash pages, i.e. it does not offer any kind of re-write strategy since it does not use any internal buffers.
- The flash driver does not provide mechanisms for providing data integrity (e.g. checksums, redundant storage, etc.).

4.2 Applicability to car domains

No restrictions.

5 Dependencies to other modules

5.1 File structure

5.1.1 Code file structure

FLS159: The code file structure shall not be defined within this specification completely. At this point it shall be pointed out that the code-file structure shall include the following files named:

- Fls_Lcfg.c – for link time configurable parameters and
- Fls_PBcfg.c – for post build time configurable parameters.

These files shall contain all link time and post-build time configurable parameters.

FLS179: Pre- and post-compile configuration parameters shall be located outside the source code of the module to allow for automatic (tool based) configuration.

5.1.2 Header file structure

FLS107: The Fls module shall comply with the following file structure:

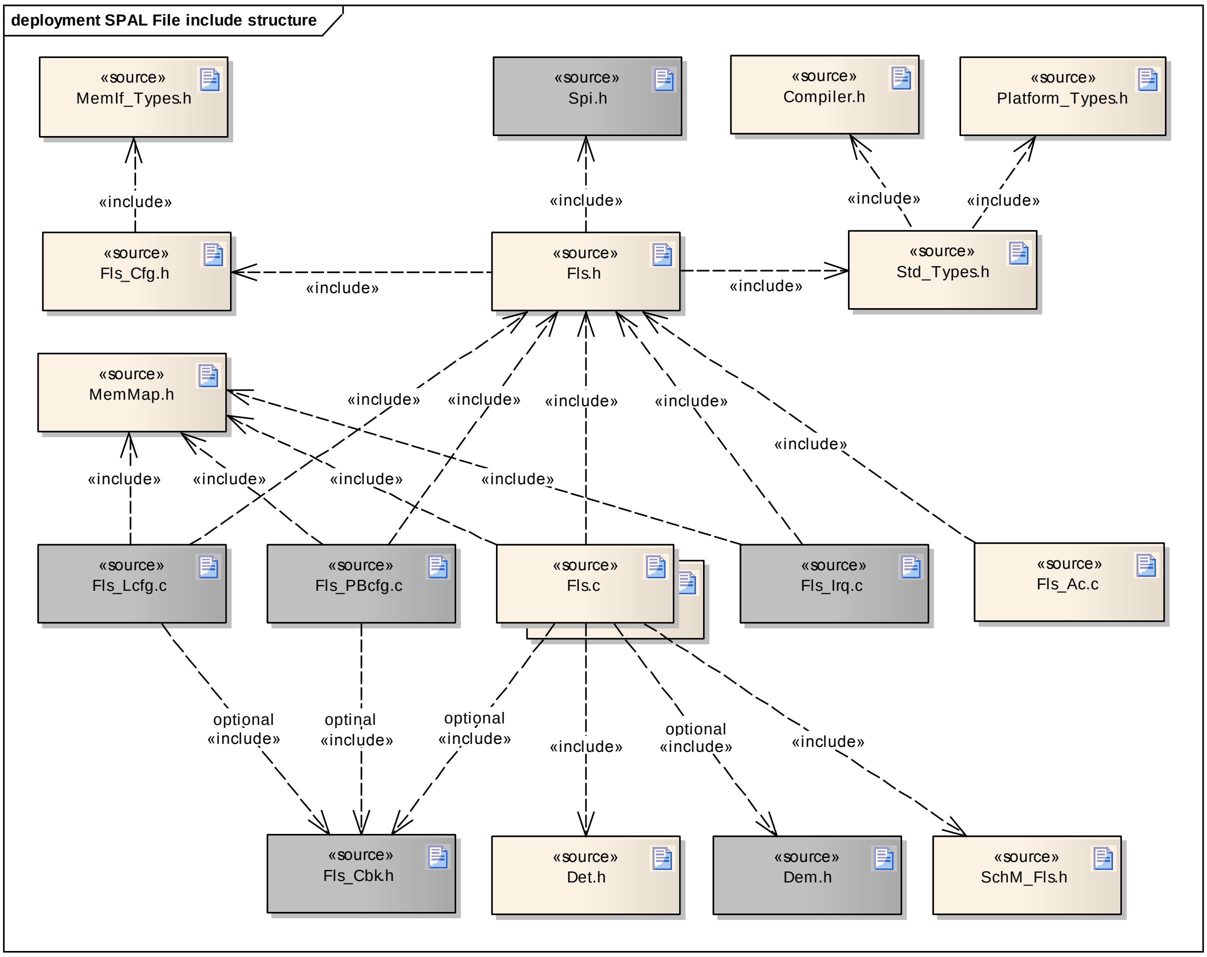


Figure 1: File include structure

Note: The files shown in grey are optional and might not be present for all implementations and/or configurations of a specific implementation of the Fls module.

FLS073: Types and definitions common to several flash driver instances shall be given in the header file `MemIf_Types.h`. Types and definitions specific for one flash driver shall be given in the header file `Fls.h`. This file shall be included in the flash driver's implementation module `Fls.c`.

5.2 System clock

If the hardware of the internal flash memory depends on the system clock, changes to the system clock (e.g. PLL on → PLL off) may also affect the clock settings of the flash memory hardware.

5.3 Communication or I/O drivers

If the flash memory is located in an external device, the access to this device shall be enacted via the corresponding communication respectively I/O driver.

6 Requirements traceability

Document: General Requirements on Basic Software Modules

Requirement	***Satisfied by***
[BSW00344] Reference to link-time configuration	Not applicable (this module does not provide any link-time parameters)
[BSW00404] Reference to post build time configuration	FLS014, FLS173, FLS174
[BSW00405] Reference to multiple configuration sets	FLS014, FLS173, FLS174
[BSW00345] Pre-compile-time configuration	FLS171, FLS172
[BSW159] Tool-based configuration	FLS179
[BSW167] Static configuration checking	FLS205, FLS206
[BSW171] Configurability of optional functionality	FLS172, FLS183, FLS184, FLS185, FLS186, FLS187, FLS188
[BSW170] Data for reconfiguration of AUTOSAR SW-components	Not applicable (this module does not depend on faults, signal qualities, …)
[BSW00380] Separate C-File for configuration parameters	FLS159, FLS179
[BSW00419] Separate C-Files for pre-compile time configuration parameters	FLS179
[BSW00381] Separate configuration header file for pre-compile time parameters	FLS107
[BSW00412] Separate H-File for configuration parameters	FLS107
BSW00383] List dependencies of configuration files	External flash driver
[BSW00384] List dependencies to other modules	Chapter 5
[BSW00387] Specify the configuration class of callback function	Not applicable (this module does not provide any callback routines)
[BSW00388] Introduce containers	Chapter 10.2
[BSW00389] Containers shall have names	Chapter 10.2
[BSW00390] Parameter content shall be unique within the module	Chapter 10.2
[BSW00391] Parameter shall have unique names	Chapter 10.2
[BSW00392] Parameters shall have a type	Chapter 10.2
[BSW00393] Parameters shall have a range	Chapter 10.2
[BSW00394] Specify the scope of the parameters	Chapter 10.2
BSW00395] List the required parameters (per parameter)	Chapter 10.2
[BSW00396] Configuration classes	Chapter 0
[BSW00397] Pre-compile-time parameters	Chapter 10.2,
[BSW00398] Link-time parameters	Not applicable (this module does not provide any link-time parameters)
[BSW00399] Loadable Post-build time parameters	Chapter 10.2
[BSW00400] Selectable Post-build time parameters	Chapter 10.2
[BSW00402] Published information	Chapter 10.3
[BSW00375] Notification of wake-up reason	Not applicable (this module does not wake up the ECU)
[BSW101] Initialization interface	FLS014

Requirement	***Satisfied by***
[BSW00416] Sequence of Initialization	Not applicable (requirement on system architecture, not on a single module)
[BSW00406] Check module initialization	FLS268
[BSW168] Diagnostic Interface of SW components	Not applicable (no use case)
[BSW00407] Function to read out published parameters	Chapter 8.3.10
[BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	Not applicable (this module does not provide an AUTOSAR interface)
[BSW00424] BSW main processing function task allocation	Not applicable (requirement on system design, not on a single module)
[BSW00425] Trigger conditions for schedulable objects	Chapter 8.5
[BSW00426] Exclusive areas in BSW modules	Not applicable (this module does not provide any exclusive areas)
[BSW00427] ISR description for BSW modules	Not applicable (no ISR's defined for this module, usage of interrupts is implementation specific)
[BSW00428] Execution order dependencies of main processing functions	Not applicable (this module does provide only one main processing function)
[BSW00429] Restricted BSW OS functionality access	Not applicable (requirement on the implementation, not for the specification)
[BSW00431] The BSW Scheduler module implements task bodies	Not applicable (requirement on the BSW scheduler module)
[BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path	See Chapter 8.5
[BSW00433] Calling of main processing functions	Not applicable (requirement on system design, not on a single module)
[BSW00434] The Schedule Module shall provide an API for exclusive areas	Not applicable (this module does not provide any exclusive areas)
[BSW00336] Shutdown interface	Not applicable (no use case).
[BSW00337] Classification of errors	FLS004, FLS007
[BSW00338] Detection and Reporting of development errors	FLS077
[BSW00369] Do not return development error codes via API	FLS267
[BSW00339] Reporting of production relevant error status	Not applicable (this module only provides production relevant error events, no error status)
[BSW00421] Reporting of production relevant error events	FLS006, FLS104 , FLS105 , FLS106, FLS154
[BSW00422] Debouncing of production relevant error status	Not applicable (requirement on the DEM)
[BSW00420] Production relevant error event rate detection	Not applicable (requirement on the DEM)
[BSW00417] Reporting of Error Events by Non-Basic Software	Not applicable (this is a BSW mdoule)

Requirement	***Satisfied by***
[BSW00323] API parameter checking	FLS015, FLS020, FLS021, FLS026, FLS027, FLS097, FLS098
[BSW004] Version check	FLS205, FLS206
[BSW00409] Header files for production code error IDs	FLS160, FLS107
[BSW00385] List possible error notificatons	FLS004, FLS007
[BSW00386] Configuration for detecting an error	FLS077, FLS162, FLS163, FLS172
[BSW161] Microcontroller abstraction	Not applicable (requirement on AUTOSAR architecture, not a single module)
[BSW162] ECU layout abstraction	Not applicable (requirement on AUTOSAR architecture, not a single module)
[BSW00324] Do not use HIS I/O Library	Not applicable (architecture decision)
[BSW005] No hard coded horizontal interfaces within MCAL	Not applicable (requirement on AUTOSAR architecture, not a single module)
[BSW00415] User dependent include files	Not applicable (only one user for this module)
[BSW164] Implementation of interrupt service routines	FLS193
[BSW00325] Runtime of interrupt service routines	FLS193
[BSW00326] Transition from ISRs to OS tasks	Not applicable (requirement on implementatio, not on specification)
[BSW00342] Usage of source code and object code	Not applicable (requirement on AUTOSAR architecture, not a single module)
[BSW00343] Specification and configuration of time	FLS178
[BSW160] Human-readable configuration data	Not applicable (requirement on documentation, not on specification)
[BSW007] HIS MISRA C	Not applicable (requirement on implementation, not on specification)
[BSW00300] Module naming convention	Not applicable (requirement on implementation, not on specification)
[BSW00413] Accessing instances of BSW modules	Conflict: This is currently not reflected in the driver’s specification. This requirement will have impact on almost all BSW modules, therefore it can not be implemented within the Release 2.0 timeframe.
[BSW00347] Naming separation of different instances of BSW drivers	Not applicable (requirement on the implementation, not on the specification)
[BSW00305] Self-defined data types naming convention	Chapter 8.2
[BSW00307] Global variables naming convention	Not applicable (requirement on the implementation, not on the specification)
[BSW00310] API naming convention	Chapter 8.3
[BSW00373] Main processing function naming convention	Chapter 8.5.1
[BSW00327] Error values naming convention	FLS004, FLS007

Requirement	***Satisfied by***
[BSW00335] Status values naming convention	Chapter 8.1
[BSW00350] Development error detection keyword	FLS077, FLS162, FLS172
[BSW00408] Configuration parameter naming convention	Chapter 10.2
[BSW00410] Compiler switches shall have defined values	Chapter 10.2
[BSW00411] Get version info keyword	Chapter 10.2
[BSW00346] Basic set of module files	FLS107
[BSW158] Separation of configuration from implementation	FLS107
[BSW00314] Separation of interrupt frames and service routines	Not applicable (this module does not implement any ISRs)
[BSW00370] Separation of callback interface from API	Not applicable (this module does not provide any callback routines)
[BSW00348] Standard type header	Not applicable (standard header files included via interface header file)
[BSW00353] Platform specific type header	Not applicable (standard header files included via interface header file)
[BSW00361] Compiler specific language extension header	Not applicable (standard header files included via interface header file)
[BSW00301] Limit imported information	FLS107
[BSW00302] Limit exported information	Not applicable (requirement on the implementation, not on the specification)
[BSW00328] Avoid duplication of code	Not applicable (requirement on the implementation, not on the specification)
[BSW00312] Shared code shall be reentrant	Not applicable (requirement on the implementation, not on the specification)
[BSW006] Platform independency	Not applicable (this is a module of the microcontroller abstraction layer)
[BSW00357] Standard API return type	Chapter 8.3.2, Chapter 8.3.3. Chapter 8.3.7, Chapter 8.3.8
[BSW00377] Module specific API return types	Chapter 8.3.5, Chapter 8.3.6
[BSW00304] AUTOSAR integer data types	Not applicable (requirement on implementation, not for specification)
[BSW00355] Do not redefine AUTOSAR integer data types	Not applicable (requirement on implementation, not for specification)
[BSW00378] AUTOSAR boolean type	Not applicable (requirement on implementation, not for specification)
[BSW00306] Avoid direct use of compiler and platform specific keywords	Not applicable (requirement on implementation, not for specification)
[BSW00308] Definition of global data	Not applicable (requirement on implementation, not for specification)

Requirement	***Satisfied by***
[BSW00309] Global data with read-only constraint	Not applicable (requirement on implementation, not for specification)
[BSW00371] Do not pass function pointers via API	Not applicable (no function pointers in this specification)
[BSW00358] Return type of init() functions	Chapter 8.3.1
[BSW00414] Parameter of init function	Chapter 8.3.1, FLS194
[BSW00376] Return type and parameters of main processing functions	Chapter 8.5.1
[BSW00359] Return type of callback functions	Not applicable (this module does not provide any callback routines)
[BSW00360] Parameters of callback functions	Not applicable (this module does not provide any callback routines)
[BSW00329] Avoidance of generic interfaces	Chapter 8.3 (explicit interfaces defined)
[BSW00330] Usage of macros / inline functions instead of functions	Not applicable (requirement on implementation, not for specification)
[BSW00331] Separation of error and status values	FLS004, FLS267
[BSW009] Module User Documentation	Not applicable (requirement on documentation, not on specification)
[BSW00401] Documentation of multiple instances of configuration parameters	Not applicable (all configuration parameters are single instance only)
[BSW172] Compatibility and documentation of scheduling strategy	Not applicable (no internal scheduling policy)
[BSW010] Memory resource documentation	Not applicable (requirement on documentation, not on specification)
[BSW00333] Documentation of callback function context	Not applicable (requirement on documentation, not for specifciation)
[BSW00374] Module vendor identification	FLS178
[BSW00379] Module identification	FLS178
[BSW003] Version identification	FLS178
[BSW00318] Format of module version numbers	FLS178
[BSW00321] Enumeration of module version numbers	Not applicable (requirement on implementation, not for specification)
[BSW00341] Microcontroller compatibility documentation	Not applicable (requirement on documentation, not on specification)
[BSW00334] Provision of XML file	Not applicable (requirement on documentation, not on specification)

Document: General Requirements on SPAL

Requirement	***Satisfied by***
[BSW12263] Object code compatible configuration concept	FLS173, FLS174
[BSW12056] Configuration of notification mechanisms	FLS173, FLS174
[BSW12267] Configuration of wakeup sources	Not applicable (this module does not wake up the ECU / MCU)
[BSW12057] Driver module initialization	FLS014
[BSW12163] Driver module de-initialization	Not applicable (no use case)
[BSW12125] Initialization of hardware resources	FLS086
[BSW12461] Responsibility for register initialization	FLS086
[BSW12462] Provide settings for register initialization	Not applicable (requirement on documentation not on specification)
BSW12463] Combine and forward settings for register initialization	Not applicable (requirement on configuration, not on specification)
[BSW12068] MCAL initialization sequence	Not applicable (not a requirement for this driver but for system integration)
[BSW12069] Wake-up notification of ECU State Manager	Not applicable (the flash driver does not wake the ECU / MCU)
[BSW157] Notification mechanisms of drivers and handlers	Chapter 8.3.5, Chapter 8.6.3, FLS164, FLS006
[BSW12169] Control of operation mode	FLS155
[BSW12063] Raw value mode	Not applicable (the flash driver does not interpret the flash data)
[BSW12075] Use of application buffers	FLS002, FLS003
[BSW12129] Resetting of interrupt flags	FLS232, FLS233, FLS234
[BSW12064] Change of operation mode during running operation	Not applicable (the flash driver does not support different modes)
[BSW12448] Behavior after development error detection	FLS015, FLS020, FLS021, FLS026, FLS027, FLS097, FLS098
[BSW12067] Setting of wake-up conditions	Not applicable (the flash driver does not wake the ECU / MCU)
[BSW12077] Non-blocking implementation	Chapter 8.5.1
[BSW12078] Runtime and memory efficiency	Not applicable (requirement on implementation, not on specification)
[BSW12092] Access to drivers	Not applicable (requirement on system design, not on a single module)
[BSW12265] Configuration data shall be kept constant	FLS191
[BSW12264] Specification of configuration items	FLS172, FLS174

Document: Requirements on Flash Driver

Requirement	***Satisfied by***
[BSW12132] Flash driver static configuration	FLS048, FLS171
[BSW12133] Publication of flash properties	FLS177, FLS178
[BSW12134] Flash read function	FLS236, FLS237, FLS238, FLS239, FLS097, FLS098
[BSW12135] Flash write function	FLS223, FLS224, FLS225, FLS226, FLS026, FLS027
[BSW12136] Flash erase function	FLS218, FLS219, FLS220, FLS221, FLS020, FLS021
BSW13301 Flash compare function	FLS241, FLS242, FLS243, FLS244, FLS150, FLS151., FLS152, FLS153, FLS186
[BSW12137] Flash cancel function	FLS229, FLS230, FLS183
[BSW12138] Flash driver status function	FLS034, FLS184
BSW13302 Flash driver mode selection function	FLS155, FLS156, FLS187
[BSW12159] Flash address check	FLS020, FLS021, FLS026, FLS027, FLS097, FLS098
[BSW12158] Flash blank check	FLS055
[BSW12141] Flash write verification	FLS056
[BSW12160] Flash erase verification	FLS022
[BSW12143] Flash driver job management	FLS016, FLS268, FLS023, FLS030, FLS032, FLS100
[BSW12144] Flash driver job processing function	FLS037, FLS038, FLS039, See Chapter 8.5
BSW13303Job processing – normal mode	FLS040
BSW13304 Job processing – fast mode	FLS040
[BSW12193] Load flash access code to RAM on job start	FLS140, FLS141
[BSW12194] Execute flash access code from RAM	FLS212, FLS213
BSW13300 Remove flash access code from RAM	FLS143
[BSW12147] Functional scope	FLS088
[BSW12182] External flash driver static configuration	FLS174
[BSW12107] Check Flash type	FLS144
[BSW12145] Flash driver job processing execution time	FLS040, FLS176, FLS182
[BSW12083] Use HIS specification as basis	Not applicable (the module provides comparable functionality but different API and different design rules)
[BSW12184] Limit read access blocking times	FLS040
[BSW12148] Common Flash API	FLS088
[BSW12149] Microcontroller independency	Not applicable (requirement on implementation, not on specification)

7 Functional specification

7.1 General design rules

FLS001: The FLS module shall offer asynchronous services for operations on flash memory (read/erase/write).

FLS002: The FLS module shall not buffer data. The FLS module shall use application data buffers that are referenced by a pointer passed via the API.

FLS003: The FLS module shall not ensure data consistency of the given application buffer.

It is the responsibility of the FLS module's environment to ensure consistency of flash data during a flash read or write operation.

FLS205: The FLS module shall check static configuration parameters statically (at the latest during compile time) for correctness.

FLS206: The FLS module shall validate the version information in the FLS module header and source files for consistency (e.g. by comparing the version information in the module header and source files with a pre-processor macro).

FLS208: The FLS module shall combine all available flash memory areas into one linear address space (denoted by the parameters `FlsBaseAddress` and `FlsTotalSize`).

FLS209: The FLS module shall map the address and length parameters for the read, write, erase and compare functions as "virtual" addresses to the physical addresses according to the physical structure of the flash memory areas.

As long as the restrictions regarding the alignment of those addresses are met it is allowed that a read, write or erase job crosses the boundaries of a physical flash memory area.

7.2 Error classification

FLS160: Values for production code Event Ids are assigned externally by the configuration of the Dem. They are published in the file `Dem_IntErrId.h` and included via `Dem.h`.

FLS161: Development error values are of type uint8.

FLS004: The FLS module shall be able to detect the following errors and exceptions depending on its configuration (development/production):

Type or error	*Relevance*	*Related error code*	*Value [hex]*

API service called with wrong parameter	Development	`FLS_E_PARAM_CONFIG`
		`FLS_E_PARAM_ADDRESS`
		`FLS_E_PARAM_LENGTH`
		`FLS_E_PARAM_DATA`
			0x01
			0x02
			0x03
			0x04

API service called without module initialization	Development	`FLS_E_UNINIT`	0x05
API service called while driver still busy	Development	`FLS_E_BUSY`	0x06
Erase verification (blank check) failed	Development	`FLS_E_VERIFY_ERASE_FAILED`	0x07
Write verification (compare) failed	Development	`FLS_E_VERIFY_WRITE_FAILED`	0x08
Timeout exceeded	Development	`FLS_E_TIMEOUT`	0x09
Flash erase failed (HW)	Production	`FLS_E_ERASE_FAILED`	Assigned by DEM
Flash write failed (HW)	Production	`FLS_E_WRITE_FAILED`	Assigned by DEM
Flash read failed (HW)	Production	`FLS_E_READ_FAILED`	Assigned by DEM
Flash compare failed (HW)	Production	`FLS_E_COMPARE_FAILED`	Assigned by DEM
Expected hardware ID not matched (see [FLS144])	Production	`FLS_E_UNEXPECTED_FLASH_ID`	Assigned by DEM

7.3 Error detection

FLS077: The detection of development errors shall be configurable (on/off) at pre-compile time. The switch `FlsDevErrorDetect` (see chapter 10) shall activate or deactivate the detection of all development errors.

FLS162: If the `FlsDevErrorDetect` switch is enabled, API parameter checking is enabled. The detailed description of the detected errors can be found in chapter 7.2 and chapter 8.3.

FLS163: The detection of production code errors cannot be switched off.

7.4 Error notification

FLS164: Detected development errors shall be reported to `Det_ReportError` service of the Development Error Tracer (DET) if the pre-processor switch `FlsDevErrorDetect` is set (see chapter 10).

FLS006: Production relevant errors shall be reported to the Diagnostic Event Manager.

FLS267: The error codes shall not be used as return values of the called function.

FLS007: Additional errors that are detected because of specific implementation and/or specific hardware properties shall be added in the flash driver's implementation documentation. The classification and enumeration shall be compatible with the errors listed above [FLS004].

7.5 External flash driver

FLS144: During the initialization of the external flash driver, the FLS module shall check the hardware ID of the external flash device against the corresponding published parameter. If a hardware ID mismatch occurs, the FLS module shall report the error code `FLS_E_UNEXPECTED_FLASH_ID` to the Diagnostic Event Manager (DEM), set the FLS module status to `FLS_E_UNINIT` and shall not initialize itself.

A complete list of required parameters is specified in the SPI Handler/Driver Software Specification (Chapter "Configuration Specification", marked as "SPI User").

7.6 Loading, executing and removing the flash access code

Technical background information: Flash technology or flash memory segmentation may require that the routines that access the flash hardware (internal erase and write routines) are executed from RAM because reading the flash - for instruction fetch needed for code execution - is not allowed while programming the flash.

FLS137: The FLS module's implementer shall place the code of the flash access routines into a separate C-module `Fls_ac.c`.

FLS215: The FLS module's flash access routines shall only disable interrupts and wait for the completion of the erase / write command if necessary (that is if it has to be ensured that no other code is executed in the meantime).

FLS211: The FLS module's implementer shall keep the execution time for the flash access code as short as possible.

FLS140: The FLS module's erase routine shall load the flash access code for erasing the flash memory to the location in RAM pointed to by the erase function pointer contained in the flash drivers configuration set if the FLS module is configured to load the flash access code to RAM on job start.

FLS141: The FLS module's write routine shall load the flash access code for writing the flash memory to the location in RAM pointed to by the write function pointer contained in the flash drivers configuration set if the FLS module is configured to load the flash access code to RAM on job start.

FLS212: The FLS module's main processing routine shall execute the flash access code routines.

FLS213: The FLS module's main processing routine shall access the flash access code routines by means of the respective function pointer contained in the FLS module's configuration set (post-compile parameters) regardless whether the flash access code routines have been loaded to RAM or whether they can be executed directly from (flash) ROM.

FLS143: After an erase or write job has been finished or cancelled, the FLS module's main processing routine shall unload (i.e. overwrite) the flash access code (internal erase / write routines) from RAM if they have been loaded to RAM by the flash driver.

FLS214: The FLS module shall only load the access code to the RAM if the access code cannot be executed out of flash ROM.

8 API specification

8.1 Imported types

FLS248:

Module	*Imported Type*
Dem	Dem_EventIdType
MemIf	MemIf_JobResultType
	MemIf_ModeType
	MemIf_StatusType
Std_Types	Std_ReturnType
	Std_VersionInfoType

8.2 Type definitions

8.2.1 Fls_ConfigType

Name:	`Fls_ConfigType`	
Type:	`Structure`	
Range:	`Hardware dependend structure`	Structure to hold the flash driver configuration set. The contents of the initialisation data structure are specific to the flash memory hardware.
Description:	A pointer to such a structure is provided to the flash driver initialization routine for configuration of the driver and flash memory hardware.	

8.2.2 Fls_AddressType

Name:	`Fls_AddressType`		
Type:	`Unsigned Integer`		
Range:	`8 / 16 / 32 bits`	`--`	Size depends on target platform and flash device.
Description:	Used as address offset from the configured flash base address to access a certain flash memory area.		

FLS216: The type Fls_AddressType shall have 0 as lower limit for each flash device.

FLS217: The FLS module shall add a device specific base address to the address type Fls_AddressType if necessary.

8.2.3 Fls_LengthType

Name:	Fls_LengthType		
Type:	Unsigned Integer		
Range:	Same as Fls_AddressType	--	Shall be the same type as Fls_AddressType because of arithmetic operations. Size depends on target platform and flash device.
Description:	Specifies the number of bytes to read/write/erase/compare.		

8.3 Function definitions

8.3.1 Fls_Init

FLS249:

Service name:	Fls_Init	
Syntax:	void Fls_Init(const Fls_ConfigType* ConfigPtr)	
Service ID[hex]:	0x00	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	ConfigPtr	Pointer to flash driver configuration set.
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	Initializes the Flash Driver.	

FLS014: The function Fls_Init shall initialize the FLS module (software) and all flash memory relevant registers (hardware) with parameters provided in the given configuration set.

FLS191: The function Fls_Init shall store the pointer to the given configuration set in a local variable in order to allow the FLS module access to the configuration set contents during runtime.

FLS086: The function Fls_Init shall initialize all FLS module global variables and those controller registers that are needed for controlling the flash device and that do not influence or depend on other (hardware) modules. Registers that can influence or depend on other modules shall be initialized by a common system module.

FLS015: If development error detection for the module Fls is enabled: the function Fls_Init shall check the (hardware specific) contents of the given configuration set for being within the allowed range. If this is not the case, it shall raise the development error FLS_E_PARAM_CONFIG.

FLS016: The function Fls_Init shall set the FLS module state to MEMIF_IDLE and the flash job result to MEMIF_JOB_OK after having finished the FLS module initialization.

FLS268: If development error detection for the module Fls is enabled: the function `Fls_Init` shall check that the FLS module is currently not busy (FLS module state is not `MEMIF_BUSY`). If this check fails, the function `Fls_Init` shall raise the development error `FLS_E_BUSY`.

FLS048: If supported by hardware, the function `Fls_Init` shall set the flash memory erase/write protection as provided in the configuration set.

FLS271: If not applicable (i.e. for configuration variant PC), a NULL pointer shall be passed to the initialization routine. In this case the check for this NULL pointer shall be omitted.

8.3.2 Fls_Erase

FLS250:

Service name:	Fls_Erase	

Syntax:	`Std_ReturnType Fls_Erase(`
	`    Fls_AddressType TargetAddress,`
	`    Fls_LengthType Length`
	`)`	

Service ID[hex]:	0x01	
Sync/Async:	Asynchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TargetAddress	Target address in flash memory. This address offset will be added to the flash memory base address. Min.: 0 Max.: FLS_SIZE - 1
	Length	Number of bytes to erase Min.: 1 Max.: FLS_SIZE - TargetAddress
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: erase command has been accepted E_NOT_OK: erase command has not been accepted
Description:	Erases flash sector(s).	

FLS218: The job of the function `Fls_Erase` shall erase one or more complete flash sectors.

FLS219: The function `Fls_Erase` shall copy the given parameters to FLS module internal variables, initiate an erase job, set the FLS module status to `MEMIF_BUSY`, set the job result to `MEMIF_JOB_PENDING` and return with `E_OK`.

FLS220: The FLS module shall execute the job of the function `Fls_Erase` asynchronously within the FLS module's main function.

FLS221: The job of the function `Fls_Erase` shall erase a flash memory block starting from `FlsBaseAddress` + `TargetAddress` of size `Length`.

Note: Length will be rounded up to the next full sector boundary since only complete flash sectors can be erased.

FLS020: If development error detection for the module Fls is enabled: the function Fls_Erase shall check that the erase start address (FlsBaseAddress + TargetAddress) is aligned to a flash sector boundary and that it lies within the specified lower and upper flash address boundaries. If this check fails, the function Fls_Erase shall reject the erase request, raise the development error FLS_E_PARAM_ADDRESS and return with E_NOT_OK.

FLS021: If development error detection for the module Fls is enabled: the function Fls_Erase shall check that the erase length is greater than 0 and that the erase end address (erase start address + length) is aligned to a flash sector boundary and that it lies within the specified upper flash address boundary. If this check fails, the function Fls_Erase shall reject the erase request, raise the development error FLS_E_PARAM_LENGTH and return with E_NOT_OK.

FLS065: If development error detection for the module Fls is enabled: the function Fls_Erase shall check that the FLS module has been initialized. If this check fails, the function Fls_Erase shall reject the erase request, raise the development error FLS_E_UNINIT and return with E_NOT_OK.

FLS023: If development error detection for the module Fls is enabled: the function Fls_Erase shall check that the FLS module is currently not busy. If this check fails, the function Fls_Erase shall reject the erase request, raise the development error FLS_E_BUSY and return with E_NOT_OK.

FLS145: If possible, e.g. with interrupt controlled implementations, the FLS module shall start the first round of the erase job directly within the function Fls_Erase to reduce overall runtime.

8.3.3 Fls_Write

FLS251:

Service name:	Fls_Write	

Syntax:	`Std_ReturnType Fls_Write(`
	`    Fls_AddressType TargetAddress,`
	`    const uint8* SourceAddressPtr,`
	`    Fls_LengthType Length`
	`)`	

Service ID[hex]:	0x02	
Sync/Async:	Asynchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TargetAddress	Target address in flash memory. This address offset will be added to the flash memory base address. Min.: 0 Max.: FLS_SIZE - 1
	SourceAddressPtr	Pointer to source data buffer
	Length	Number of bytes to write Min.: 1 Max.: FLS_SIZE - TargetAddress
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: write command has been accepted E_NOT_OK: write command has not been accepted
Description:	Writes one or more complete flash pages.	

FLS223: The job of the function `Fls_Write` shall write one or more complete flash pages to the flash device.

FLS224: The function `Fls_Write` shall copy the given parameters to Fls module internal variables, initiate a write job, set the FLS module status to `MEMIF_BUSY`, set the job result to `MEMIF_JOB_PENDING` and return with `E_OK`.

FLS225: The FLS module shall execute the write job of the function `Fls_Write` asynchronously within the FLS module's main function.

FLS226: The job of the function `Fls_Write` shall program a flash memory block with data provided via `SourceAddressPtr` starting from `FlsBaseAddress + TargetAddress` of size `Length.`

FLS026: If development error detection for the module Fls is enabled: the function `Fls_Write` shall check that the write start address (`FlsBaseAddress + TargetAddress)` is aligned to a flash page boundary and that it lies within the specified lower and upper flash address boundaries. If this check fails, the function `Fls_Write` shall reject the write request, raise the development error `FLS_E_PARAM_ADDRESS` and return with `E_NOT_OK.`

FLS027: If development error detection for the module Fls is enabled: the function `Fls_Write` shall check that the write length is greater than 0, that the write end address (write start address + length) is aligned to a flash page boundary and that it lies within the specified upper flash address boundary. If this check fails, the function

Fls_Write shall reject the write request, raise the development error FLS_E_PARAM_LENGTH and return with E_NOT_OK.

FLS066: If development error detection for the module Fls is enabled: the function Fls_Write shall check that the FLS module has been initialized. If this check fails, the function Fls_Write shall reject the write request, raise the development error FLS_E_UNINIT and return with E_NOT_OK.

FLS030: If development error detection for the module Fls is enabled: the function Fls_Write shall check that the FLS module is currently not busy. If this check fails, the function Fls_Write shall reject the write request, raise the development error FLS_E_BUSY and return with E_NOT_OK.

FLS157: If development error detection for the module Fls is enabled: the function Fls_Write shall check the given data buffer pointer for not being a null pointer. If the data buffer pointer is a null pointer, the function Fls_Write shall reject the write request, raise the development error FLS_E_PARAM_DATA and return with E_NOT_OK.

FLS146: If possible, e.g. with interrupt controlled implementations, the FLS module shall start the first round of the write job directly within the function Fls_Write to reduce overall runtime.

8.3.4 Fls_Cancel

FLS252:

Service name:	Fls_Cancel

Syntax:	`void Fls_Cancel(`
	`)`

Service ID[hex]:	0x03
Sync/Async:	Synchronous
Reentrancy:	Non Reentrant
Parameters (in):	None
Parameters (in-out):	None
Parameters (out):	None
Return value:	None
Description:	Cancels an ongoing job.

FLS229: The function Fls_Cancel shall cancel an ongoing flash read, write, erase or compare job.

FLS230: The function Fls_Cancel shall abort a running job synchronously so that directly after returning from this function a new job can be started.

FLS032: The function Fls_Cancel shall reset the FLS module's internal job processing variables (like address, length and data pointer) and set the FLS module state to FLS_IDLE.

FLS033: The function `Fls_Cancel` shall set the job result to `MEMIF_JOB_CANCELED` if the job result currently has the value `MEMIF_JOB_PENDING`. Otherwise the function `Fls_Cancel` shall leave the job result unchanged.

FLS147: If configured, the function `Fls_Cancel` shall call the error notification function to inform the caller about the cancellation of a job.

The FLS module's states and data of the affected flash memory cells are undefined when canceling an ongoing job with the function `Fls_Cancel`.

FLS183: The function `Fls_Cancel` shall be pre-compile time configurable `On/Off` by the configuration parameter `FlsCancelApi.`

8.3.5 Fls_GetStatus

FLS253:

Service name:	Fls_GetStatus	

Syntax:	`MemIf_StatusType Fls_GetStatus(`
	`)`	

Service ID[hex]:	0x04	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	MemIf_StatusType	--
Description:	Returns the driver state.	

FLS034: The function `Fls_GetStatus` shall return the FLS module state synchronously.

FLS184: The function `Fls_GetStatus` shall be pre-compile time configurable `On/Off` by the configuration parameter `FlsGetStatusApi.`

8.3.6 Fls_GetJobResult

FLS254:

Service name:	Fls_GetJobResult	

Syntax:	`MemIf_JobResultType Fls_GetJobResult(`
	`)`	

Service ID[hex]:	0x05	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	MemIf_JobResultType	--
Description:	Returns the result of the last job.	

FLS035: The function `Fls_GetJobResult` shall return the result of the last job synchronously.

FLS036: The erase, write, read and compare functions shall share the same job result, i.e. only the result of the last job can be queried. The FLS module shall overwrite the job result with `MEMIF_JOB_PENDING` if the FLS module has accepted a new job.

FLS185: The function `Fls_GetJobResult` shall be pre-compile time configurable `On/Off` by the configuration parameter `FlsGetJobResultApi`.

8.3.7 Fls_Read

FLS256:

Service name:	Fls_Read	

Syntax:	`Std_ReturnType Fls_Read(`
	`    Fls_AddressType SourceAddress,`
	`    uint8* TargetAddressPtr,`
	`    Fls_LengthType Length`
	`)`	

Service ID[hex]:	0x07	
Sync/Async:	Asynchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	SourceAddress	Source address in flash memory. This address offset will be added to the flash memory base address. Min.: 0 Max.: FLS_SIZE - 1
	Length	Number of bytes to read Min.: 1 Max.: FLS_SIZE - SourceAddress
Parameters (in-out):	None	
Parameters (out):	TargetAddressPtr	Pointer to target data buffer
Return value:	Std_ReturnType	E_OK: read command has been accepted E_NOT_OK: read command has not been accepted
Description:	Reads from flash memory.	

FLS236: The function `Fls_Read` shall read from flash memory.

FLS237: The function `Fls_Read` shall copy the given parameters to FLS module internal variables, initiate a read job, set the FLS module status to `MEMIF_BUSY`, set the FLS module job result to `MEMIF_JOB_PENDING` and return with `E_OK`.

FLS238: The FLS module shall execute the read job of the function `Fls_Read` asynchronously within the FLS module's main function.

FLS239: The read job of the function `Fls_Read` shall copy a continuous flash memory block starting from `FlsBaseAddress` + `SourceAddress` of size `Length` to the buffer pointed to by `TargetAddressPtr`.

FLS097: If development error detection for the module Fls is enabled: the function `Fls_Read` shall check that the read start address (`FlsBaseAddress` + `SourceAddress)` lies within the specified lower and upper flash address boundaries. If this check fails, the function `Fls_Read` shall reject the read job, raise development error `FLS_E_PARAM_ADDRESS` and return with `E_NOT_OK`.

FLS098: If development error detection for the module Fls is enabled: the function `Fls_Read` shall check that the read length is greater than 0 and that the read end address (read start address + length) lies within the specified upper flash address boundary. If this check fails, the function `Fls_Read` shall reject the read job, raise the development error `FLS_E_PARAM_LENGTH` and return with `E_NOT_OK`.

FLS099: If development error detection for the module Fls is enabled: the function `Fls_Read` shall check that the driver has been initialized. If this check fails, the function `Fls_Read` shall reject the read request, raise the development error `FLS_E_UNINIT` and return with `E_NOT_OK`.

FLS100: If development error detection for the module Fls is enabled: the function `Fls_Read` shall check that the driver is currently not busy. If this check fails, the function `Fls_Read` shall reject the read request, raise the development error `FLS_E_BUSY` and return with `E_NOT_OK`.

FLS158: If development error detection for the module Fls is enabled: the function `Fls_Read` shall check the given data buffer pointer for not being a null pointer. If the data buffer pointer is a null pointer, the function `Fls_Read` shall reject the read request, raise the development error `FLS_E_PARAM_DATA` and return with `E_NOT_OK`.

FLS240: The FLS module's environment shall only call the function `Fls_Read` after the FLS module has been initialized.

8.3.8 Fls_Compare

FLS257:

Service name:	Fls_Compare	

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

Service ID[hex]:	0x08	
Sync/Async:	Asynchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	SourceAddress	Source address in flash memory. This address offset will be added to the flash memory base address. Min.: 0 Max.: FLS_SIZE - 1
	TargetAddressPtr	Pointer to target data buffer
	Length	Number of bytes to compare Min.: 1 Max.: FLS_SIZE - SourceAddress
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: compare command has been accepted E_NOT_OK: compare command has not been accepted
Description:	Compares the contents of an area of flash memory with that of an application data buffer.	

FLS241: The function `Fls_Compare` shall compare the contents of an area of flash memory with that of an application data buffer.

FLS242: The function Fls_Compare shall copy the given parameters to Fls module internal variables, initiate a compare job, set the status to MEMIF_BUSY, set the job result to MEMIF_JOB_PENDING and return with E_OK.

FLS243: The FLS module shall execute the job of the function Fls_Compare asynchronously within the FLS module's main function.

FLS244: The job of the function Fls_Compare shall compare a continuous flash memory block starting from FlsBaseAddress + SourceAddress of size Length with the buffer pointed to by TargetAddressPtr.

FLS150: If development error detection for the module Fls is enabled: the function Fls_Compare shall check that the compare start address (FlsBaseAddress + SourceAddress) lies within the specified lower and upper flash address boundaries. If this check fails, the function Fls_Compare shall reject the compare job, raise the development error FLS_E_PARAM_ADDRESS and return with E_NOT_OK.

FLS151: If If development error detection for the module Fls is enabled: the function Fls_Compare shall check that the given length is greater than 0 and that the compare end address (compare start address + length) lies within the specified upper flash address boundary. If this check fails, the function Fls_Compare shall reject the compare job, raise the development error FLS_E_PARAM_LENGTH and return with E_NOT_OK.

FLS152: If development error detection for the module Fls is enabled: the function Fls_Compare shall check that the driver has been initialized. If this check fails, the function Fls_Compare shall reject the compare job, raise the development error FLS_E_UNINIT and return with E_NOT_OK.

FLS153: If development error detection for the module Fls is enabled: the function Fls_Compare shall check that the driver is currently not busy. If this check fails, the function Fls_Compare shall reject the compare job, raise the development error FLS_E_BUSY and return with E_NOT_OK.

FLS273: If development error detection for the module Fls is enabled: the function Fls_Compare shall check the given data buffer pointer for not being a null pointer. If the data buffer pointer is a null pointer, the function Fls_Compare shall reject the request, raise the development error FLS_E_PARAM_DATA and return with E_NOT_OK.

FLS186: The function Fls_Compare shall be pre-compile time configurable On/Off by the configuration parameter FlsCompareApi.

8.3.9 Fls_SetMode

FLS258:

Service name:	Fls_SetMode

Syntax:	void Fls_SetMode(MemIf_ModeType Mode)	
Service ID[hex]:	0x09	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	Mode	MEMIF_MODE_SLOW: Slow read access / normal SPI access. MEMIF_MODE_FAST: Fast read access / SPI burst access.
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	Sets the flash driver's operation mode.	

FLS155: The function `Fls_SetMode` shall set the FLS module's operation mode to the given "Mode" parameter.

FLS156: If development error detection for the module Fls is enabled: the function `Fls_SetMode` shall check that the FLS module is currently not busy. If this check fails, the function `Fls_SetMode` shall reject the set mode request and raise the development error code `FLS_E_BUSY`.

FLS187: The function `Fls_SetMode` shall be pre-compile time configurable `On/Off` by the configuration parameter `FlsSetModeApi`.

8.3.10 Fls_GetVersionInfo

FLS259:

Service name:	Fls_GetVersionInfo	
Syntax:	void Fls_GetVersionInfo(Std_VersionInfoType* VersioninfoPtr)	
Service ID[hex]:	0x10	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	None	
Parameters (in-out):	None	
Parameters (out):	VersioninfoPtr	Pointer to where to store the version information of this module.
Return value:	None	
Description:	Returns the version information of this module.	

FLS165: The function `Fls_GetVersionInfo` shall return the version information of the FLS module. The version information includes:
- Module Id
- Vendor Id
- Vendor specific version numbers (BSW00407).

FLS166: The function `Fls_GetVersionInfo` shall be pre-compile time configurable `On/Off` by the configuration parameter `FlsVersionInfoApi`.

FLS247: If source code for caller and callee of the function `Fls_GetVersionInfo` is available, the FLS module should realize this function as a macro. The FLS module should define this macro in the module's header file.

8.4 Call-back notifications

This chaper lists all functions provided by the Fls module to lower layer modules.

FLS193: Depending on implementation, callback routines provided and/or invoked by the FLS module may be called on interrupt level. The module providing those routines has therefore to make sure that their runtime is reasonably short, i.e. since callbacks may be propagated upward through several software layers.

8.5 Scheduled functions

This chapter lists all functions provided by the Fls module and called directly by the Basic Software Module Scheduler.

FLS269: The Fls module shall provide only one scheduled function. Reading from / writing to flash memory cannot usually be done simultaneously and the overhead for synchronizing two scheduled functions would outweigh the benefits.

8.5.1 Fls_MainFunction

FLS255:

Service name:	Fls_MainFunction

Syntax:	`void Fls_MainFunction(`
	`)`

Service ID[hex]:	0x06
Timing:	FIXED_CYCLIC
Description:	Performs the processing of jobs.

FLS037: The function `Fls_MainFunction` shall perform the processing of the flash read, write, erase and compare jobs.

FLS266: The function Fls_MainFunction shall accept only one read, write, erase or compare job at a time.

FLS038: When a job has been initiated, the FLS module's environment shall call the function `Fls_MainFunction` cyclically until the job is finished.

Note: The function Fls_MainFunction may also be called cyclically if no job is currently pending.

FLS039: The function `Fls_MainFunction` shall return without any action if no job is pending.

FLS040: The function `Fls_MainFunction` shall only process as much data in one call cycle as statically configured for the current job type (read, write, erase or compare) and the current FLS module's operating mode (normal, fast).

FLS104: The function Fls_MainFunction shall set the job result to MEMIF_JOB_FAILED and report the error code FLS_E_ERASE_FAILED to the DEM if a flash erase job fails due to a hardware error.

FLS105: The function Fls_MainFunction shall set the job result to MEMIF_JOB_FAILED and report the error code FLS_E_WRITE_FAILED to the DEM if a flash write job fails due to a hardware error.

FLS106: The function Fls_MainFunction shall set the job result to MEMIF_JOB_FAILED and report the error code FLS_E_READ_FAILED to the DEM if a flash read job fails due to a hardware error.

FLS154: The function Fls_MainFunction shall set the job result to MEMIF_JOB_FAILED and report the error code FLS_E_COMPARE_FAILED to the DEM if a flash compare job fails due to a hardware error.

FLS200: The function Fls_MainFunction shall set the job result to MEMIF_BLOCK_INCONSISTENT if the compared data from a flash compare job are not equal.

FLS022: If development error detection for the module Fls is enabled:: After a flash block has been erased, the function Fls_MainFunction shall compare the contents of the addressed memory area against the value of an erased flash cell to check that the block has been completely erased. If this check fails, the function Fls_MainFunction shall set the FLS module's job result to MEMIF_JOB_FAILED and raise development error FLS_E_VERIFY_ERASE_FAILED.

FLS055: If development error detection for the module Fls is enabled:: Before writing a flash block, the function Fls_MainFunction shall compare the contents of the addressed memory area against the value of an erased flash cell to check that the block has been completely erased. If this check fails, the function Fls_MainFunction shall set the FLS module's job result to MEMIF_JOB_FAILED and raise development error FLS_E_VERIFY_ERASE_FAILED.

FLS056: If development error detection for the module Fls is enabled:: After writing a flash block, the function Fls_MainFunction shall compare the contents of the reprogrammed memory area against the contents of the provided application buffer to check that the block has been completely reprogrammed. If this check fails, the function Fls_MainFunction shall set the FLS module's job result to MEMIF_JOB_FAILED and raise the development error FLS_E_VERIFY_WRITE_FAILED.

FLS052: After a read, erase, write or compare job has been finished, the function Fls_MainFunction shall set the FLS module's job result to MEMIF_JOB_OK if it is currently in state MEMIF_JOB_PENDING. Otherwise, it shall leave the result unchanged. Furthermore, the function Fls_MainFunction shall set the FLS module's state to MEMIF_IDLE and call the job end notification function if configured [FLS173].

FLS232: The configuration parameter FlsUseInterrupts shall switch between interrupt and polling controlled job processing if this is supported by the flash memory hardware.

FLS233: The FLS module's implementer shall locate the interrupt service routine in Fls_Irq.c.

FLS234: If interrupt controlled job processing is supported and enabled with the configuration parameter FlsUseInterrupts, the interrupt service routine shall reset the interrupt flag, check for errors reported by the underlying hardware, reload the hardware finite state machine for the next round of the pending job or call the appropriate notification routine if the job is finished or aborted.

FLS235: The function Fls_MainFunction shall process jobs without hardware interrupt support (e.g. read jobs).

FLS272: If development error detection for the module Fls is enabled: the function Fls_MainFunction shall provide a timeout monitoring for the currently running job, that is it shall supervise the deadline of the read / compare / erase or write job.

FLS359: If development error detection for the module Fls is enabled: the function Fls_MainFunction shall check, whether the configured maximum erase time (see FLS298_Conf FlsEraseTime) has been exceeded. If this is the case, the function Fls_MainFunction shall raise the development error FLS_E_TIMEOUT.

FLS360: If development error detection for the module Fls is enabled: the function Fls_MainFunction shall check, whether the expected maximum write time (see note below) has been exceeded. If this is the case, the function Fls_MainFunction shall raise the development error FLS_E_TIMEOUT.

Note: The expected maximum write time depends on the current mode of the Fls module (see FLS258), the configured number of bytes to write in this mode (see FLS278_Conf and FLS277_Conf respectively), the size of a single flash page (see FLS281_Conf) and last the maximum time to write one flash page (see FLS301_Conf). The number of bytes to write divided by the size of one flash page yields the number of pages to write in one cycle. This multiplied with the maximum write time for one flash page gives you the expected maximum write time.

FLS361: If development error detection for the module Fls is enabled: the function Fls_MainFunction shall check, whether the expected maximum read / compare time (see note below) has been exceeded. If this is the case, the function Fls_MainFunction shall raise the development error FLS_E_TIMEOUT.

Note: There are no published timings for read / compare (as these would mostly depend on whether the flash device is internal or external e.g. connected via SPI). The solution should be similar as for write jobs above: the configured number of bytes to read (and to compare) per cycle is matched to the expected read / compare times which should be supervised by the Fls_MainFunction. If this is not detailed enough there are two possibilities:

- *specify expected read / compare times (difficult because of the dependency mentioned above)*
- *leave read / compare jobs out of the timeout supervision (change FLS272).*

FLS117: If development error detection for the module Fls is enabled: the function `Fls_MainFunction` shall check that the FLS module has been initialized. If this check fails, the function `Fls_MainFunction` shall raise the development error `FLS_E_UNINIT`.

FLS196: The function `Fls_MainFunction` shall at the most issue one sector erase command (to the hardware) in each cycle.

Note: The requirement above shall ensure that maximum one sector is erased sequentially within one cycle of the driver's main function. If the hardware is capable of erasing more than one sector in parallel, this shall not be restricted by this specification.

8.6 Expected Interfaces

This chapter lists all functions the Fls module requires from other modules.

8.6.1 Mandatory Interfaces

This chapter defines all interfaces which are required to fulfill the core functionality of the module.

FLS260:

API function	***Description***
Dem_ReportErrorStatus	Reports errors to the DEM.

Note: If the flash device is connected via SPI, also the SPI interfaces are required to fulfill the modules core functionality. Which interfaces are needed exactly shall not be detailed further in this specification.

8.6.2 Optional Interfaces

This chapter defines all interfaces which are required to fulfill an optional functionality of the module.

FLS261:

API function	***Description***
Det_ReportError	Service to report development errors.

8.6.3 Configurable interfaces

In this chapter, all interfaces are listed for which the target function can be configured. The target function is usually a call-back function. The names of these kind of interfaces is not fixed because they are configurable.

FLS109: The job processing callback notifications shall be configurable as function pointers within the initialization data structure (`Fls_ConfigType`).

FLS110: The callback notifications shall have no parameters and no return value.

FLS111: If a job processing callback notification is configured as null pointer, the corresponding callback routine shall not be executed.

FLS262:

Service name:	Fee_JobEndNotification

Syntax:	`void Fee_JobEndNotification(`
	`)`

Sync/Async:	Synchronous
Reentrancy:	Don't care
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Description:	This callback function is called when a job has been completed with a positive result.

FLS167: The FLS module shall call the callback function `Fee_JobEndNotification` when the module has completed a job with a positive result:

- Read job finished & OK
- Write job finished & OK
- Erase job finished & OK
- Compare job finished & memory blocks are the same

FLS263:

Service name:	Fee_JobErrorNotification

Syntax:	`void Fee_JobErrorNotification(`
	`)`

Sync/Async:	Synchronous
Reentrancy:	Don't care
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Description:	This callback function is called when a job has been cancelled or finished with negative result.

FLS168: The FLS module shall call the callback function `Fee_JobErrorNotification` when the module has cancelled or finished a job with a negative result:

- Read job aborted or failed
- Write job aborted or failed
- Erase job aborted or failed
- Compare job aborted or failed
- Compare job finished and memory blocks differ

9 Sequence diagrams

9.1 Initialization

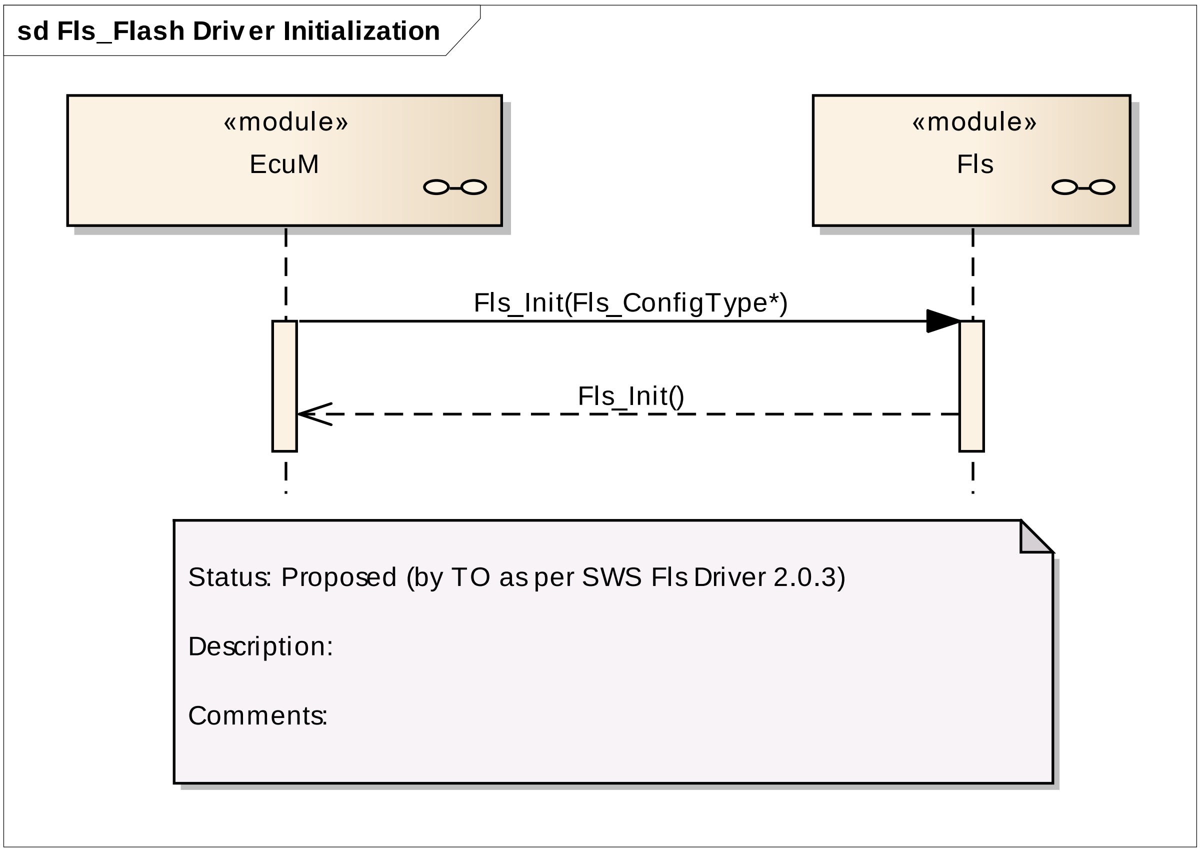

sd Fls_Flash Driver Initialization
«module»
EcuM
«module»
Fls
Fls_Init(Fls_ConfigType*)
Fls_Init()
Status: Proposed (by TO as per SWS Fls Driver 2.0.3)
Description:
Comments:

Figure 2: Flash driver initialization sequence

9.2 Synchronous functions

The following sequence diagram shows the function `Fls_GetJobResult` as an example for the synchronous functions of this module. The same sequence applies also to the functions `Fls_GetStatus` and `Fls_SetMode`.

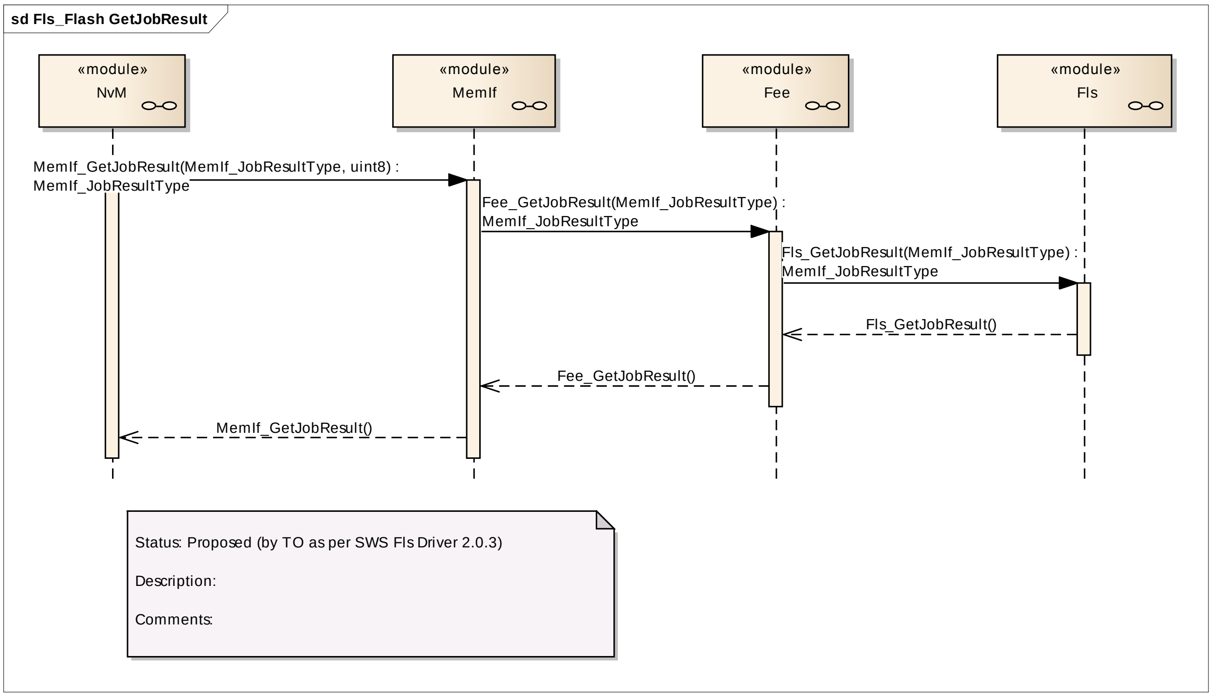
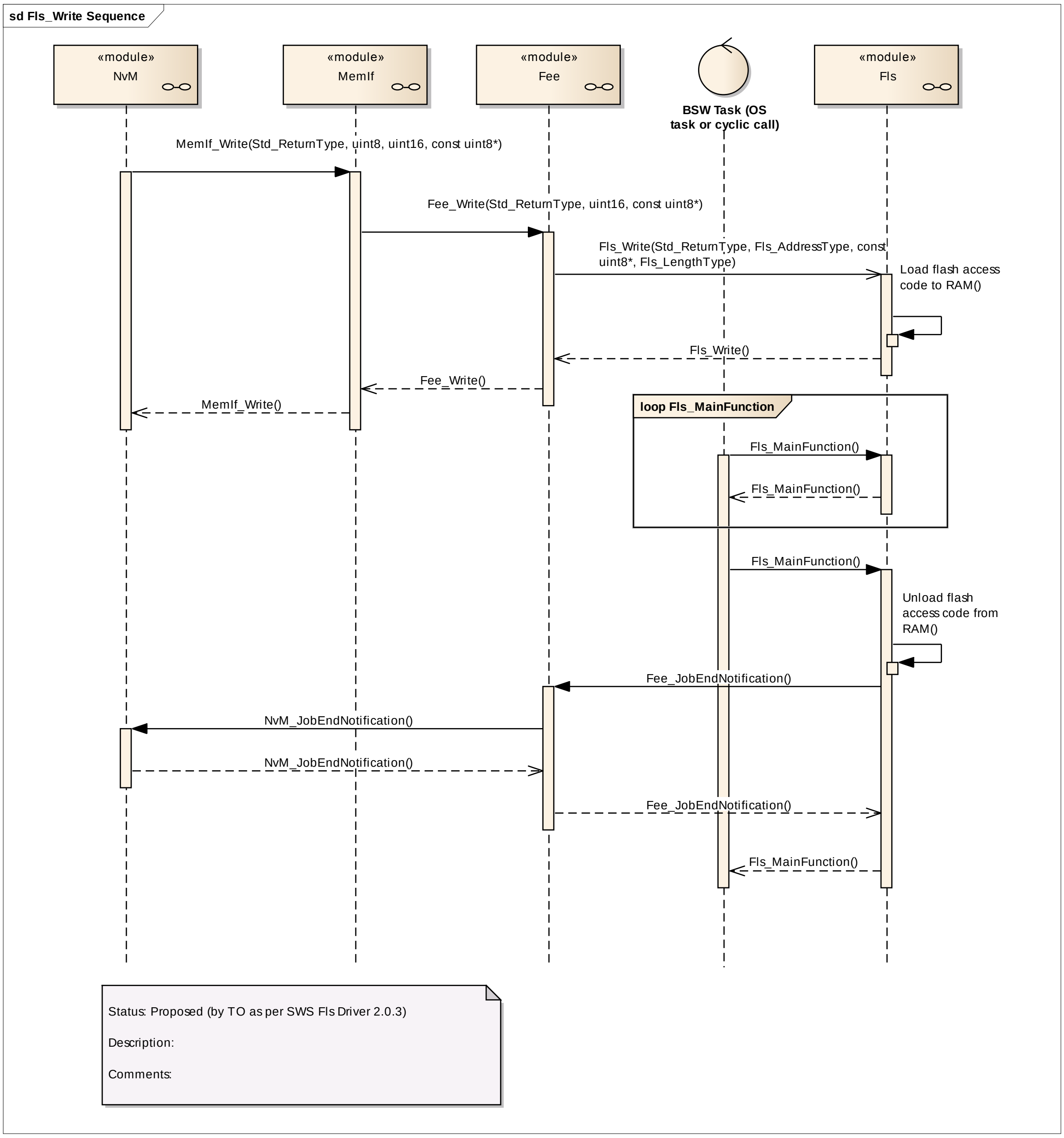

sd Fls_Flash GetJobResult
«module»
NvM
«module»
MemIf
«module»
Fee
«module»
Fls
MemIf_GetJobResult(MemIf_JobResultType, uint8) :
MemIf_JobResultType
Fee_GetJobResult(MemIf_JobResultType) :
MemIf_JobResultType
Fls_GetJobResult(MemIf_JobResultType) :
MemIf_JobResultType
Fls_GetJobResult()
Fee_GetJobResult()
MemIf_GetJobResult()
Status: Proposed (by TO as per SWS Fls Driver 2.0.3)
Description:
Comments:

Figure 3: Fls_GetJobResult

9.3 Asynchronous functions

The following sequence diagram shows the flash write function (with the configuration option `FlsAcLoadOnJobStart` set) as an example for the asynchronous functions of this module. The same sequence applies to the erase, read and compare jobs, with the only difference that for the read and compare jobs no flash access code needs to be loaded to / unloaded from RAM.



Status: Proposed (by TO as per SWS Fls Driver 2.0.3)

Description:

Comments:

Figure 4: Flash write sequence, flash access code loaded on job start

9.4 Canceling a running job

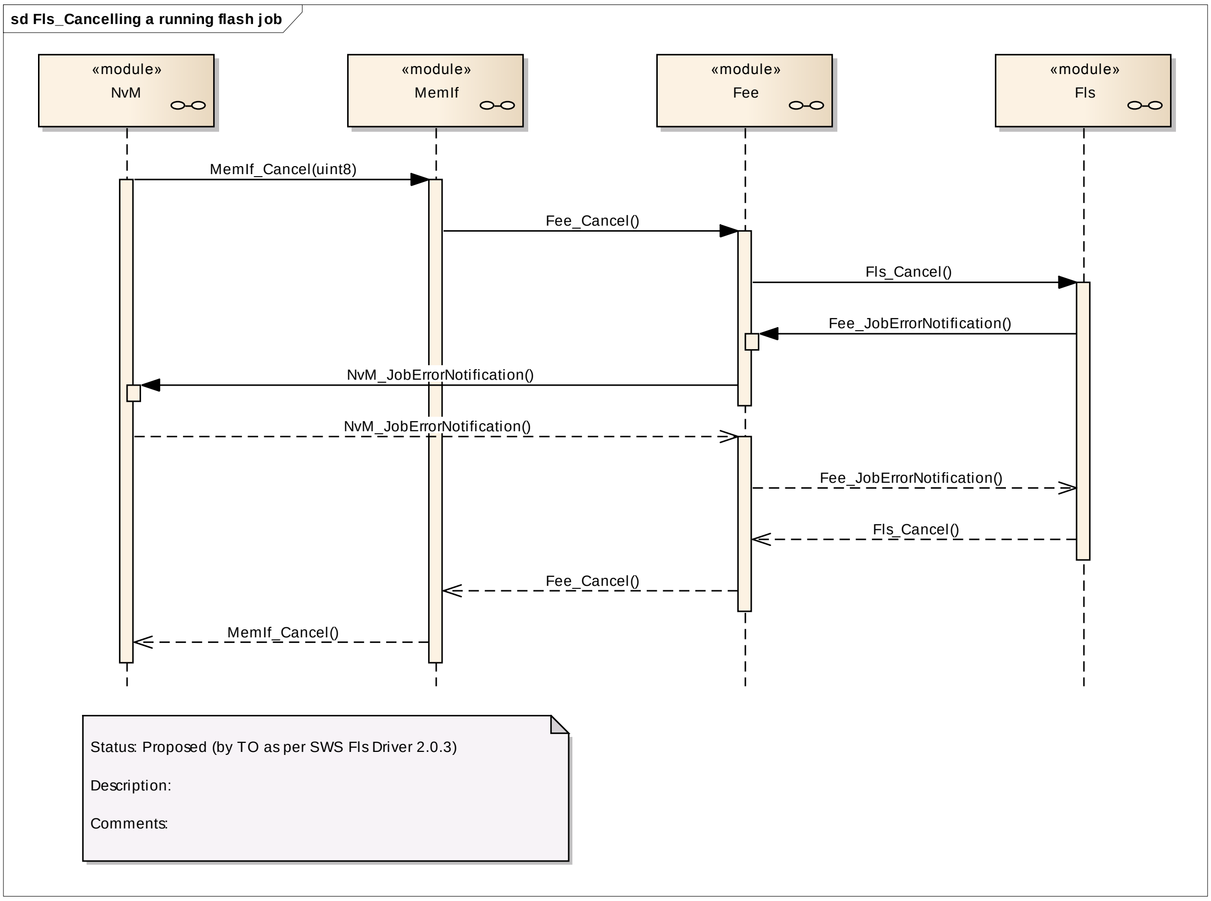

sd Fls_Cancelling a running flash job
«module» NvM
«module» MemIf
«module» Fee
«module» Fls
MemIf_Cancel(uint8)
Fee_Cancel()
Fls_Cancel()
Fee_JobErrorNotification()
NvM_JobErrorNotification()
NvM_JobErrorNotification()
Fee_JobErrorNotification()
Fls_Cancel()
Fee_Cancel()
MemIf_Cancel()
Status: Proposed (by TO as per SWS Fls Driver 2.0.3)
Description:
Comments:

Figure 5: Canceling a running flash job

FLS049: The FLS module's environment shall not call the function `Fls_Cancel` during a running `Fls_MainFunction` invocation.

This can be achieved by one of the following scheduling configurations:

- Possibility 1: The job functions of the NVRAM manager and the flash driver are synchronized (e.g. called sequentially within one task)
- Possibility 2: The task that calls the `Fls_MainFunction` function can not be preempted by another task.

10 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification Chapter 10.1 describes fundamentals. It also specifies a template (table) you shall use for the parameter specification. We intend to leave Chapter 10.1 in the specification to guarantee comprehension.

Chapter 10.2 specifies the structure (containers) and the parameters of the module Flash Driver.

Chapter 10.3 specifies published information of the module <Module Name>.

10.1 How to read this chapter

In addition to this section, it is highly recommended to read the documents:

- AUTOSAR Layered Software Architecture [2]
- AUTOSAR ECU Configuration Specification [7]
  This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration metamodel in detail.

The following is only a short survey of the topic and it will not replace the ECU Configuration Specification document.

10.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or ECU configuration.

The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term “configuration class” (of a parameter) shall be used in order to refer to a specific configuration point in time.

10.1.2 Containers

Containers structure the set of configuration parameters. This means:

- *all* configuration parameters are kept in containers.
- (sub-) containers can reference (sub-) containers. It is possible to assign a multiplicity to these references. The multiplicity then defines the possible number of instances of the contained parameters.

10.1.3 Specification template for configuration parameters

The following tables consist of three sections:

- the general section
- the configuration parameter section
- the section of included/referenced containers

Pre-compile time - specifies whether the configuration parameter shall be of configuration class *Pre-compile time* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Pre-compile time*.
--	The configuration parameter shall never be of configuration class *Pre-compile time*.

Link time - specifies whether the configuration parameter shall be of configuration class *Link time* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Link time*.
--	The configuration parameter shall never be of configuration class *Link time*.

Post Build - specifies whether the configuration parameter shall be of configuration class *Post Build* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Post Build* and no specific implementation is required.
L	*Loadable* – the configuration parameter shall be of configuration class *Post Build* and only one configuration parameter set resides in the ECU.
M	*Multiple* – the configuration parameter shall be of configuration class *Post Build* and is selected out of a set of multiple parameters by passing a dedicated pointer to the init function of the module.
--	The configuration parameter shall never be of configuration class *Post Build*.

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe Chapters 10.2 and Chapter 10.3.

10.2.1 Variants

FLS203: Variant PC: Only pre-compile time parameters

FLS204: Variant PB: FlsConfigSet (see FLS174) as post build time configurable

FLS194: The initialization function of the FLS module shall always have a pointer as a parameter, even though for Variant PC no configuration set shall be given. Instead a null pointer shall be passed to the initialization function. This means that in contradiction to BSW00414, only one interface for initialization shall be implemented and it shall not depend on the modules configuration which interface the calling software module shall use.

10.2.2 Fls

Module Name	*Fls*
Module Description	Configuration of the Fls (internal or external flash driver) module. Its multiplicity describes the number of flash drivers present, so there will be one container for each flash driver in the ECUC template. When no flash driver is present then the multiplicity is 0.

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
FlsConfigSet	1..*	Container for runtime configuration parameters of the flash driver. Implementation Type: Fls_ConfigType.
FlsGeneral	1	Container for general parameters of the flash driver. These parameters are always pre-compile.
FlsPublishedInformation	1	Additional published parameters not covered by CommonPublishedInformation container. Note that these parameters do not have any configuration class setting, since they are published information.

The table above specifies parameters that shall be configured during system generation. These parameters shall be located in the file `Fls_Cfg.h`. Further hardware or implementation specific parameters can be added if necessary.

10.2.3 FlsGeneral

SWS Item	**FLS172 :**
Container Name	FlsGeneral{Fls_ModuleConfiguration}
Description	Container for general parameters of the flash driver. These parameters are always pre-compile.
Configuration Parameters	

SWS Item	**FLS284 :**		
Name	FlsAcLoadOnJobStart {FLS_AC_LOAD_ON_JOB_START}		
Description	The flash driver shall load the flash access code to RAM whenever an erase or write job is started and unload (overwrite) it after that job has been finished or canceled. true: Flash access code loaded on job start / unloaded on job end or error. false: Flash access code not loaded to / unloaded from RAM at all.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**FLS169 :**		
Name	FlsBaseAddress {FLS_BASE_ADDRESS}		
Description	The flash memory start address (see also FLS118). FLS169: This parameter defines the lower boundary for read / write / erase and compare jobs.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	

	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	FLS285 :		
Name	FlsCancelApi {FLS_CANCEL_API}		
Description	Compile switch to enable and disable the Fls_Cancel function. true: API supported / function provided. false: API not supported / function not provided		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	FLS286 :		
Name	FlsCompareApi {FLS_COMPARE_API}		
Description	Compile switch to enable and disable the Fls_Compare function. true: API supported / function provided. false: API not supported / function not provided		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	FLS287 :		
Name	FlsDevErrorDetect {FLS_DEV_ERROR_DETECT}		
Description	Pre-processor switch to enable and disable development error detection (see FLS077). true: Development error detection enabled. false: Development error detection disabled.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	FLS288 :		
Name	FlsDriverIndex		
Description	Index of the driver, used by FEE.		
Multiplicity	1		
Type	IntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 254		
Default value	--		
ConfigurationClass	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	

Scope / Dependency	scope: module

SWS Item	**FLS289 :**		
Name	FlsGetJobResultApi {FLS_GET_JOB_RESULT_API}		
Description	Compile switch to enable and disable the Fls_GetJobResult function. true: API supported / function provided. false: API not supported / function not provided		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**FLS290 :**		
Name	FlsGetStatusApi {FLS_GET_STATUS_API}		
Description	Compile switch to enable and disable the Fls_GetStatus function. true: API supported / function provided. false: API not supported / function not provided		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**FLS291 :**		
Name	FlsSetModeApi {FLS_SET_MODE_API}		
Description	Compile switch to enable and disable the Fls_SetMode function. true: API supported / function provided. false: API not supported / function not provided		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**FLS170 :**		
Name	FlsTotalSize {FLS_TOTAL_SIZE}		
Description	The total amount of flash memory in bytes (see also FLS118). FLS170: This parameter in conjunction with FLS_BASE_ADDRESS defines the upper boundary for read / write / erase and compare jobs.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	

Scope / Dependency	scope: module

SWS Item	**FLS292 :**		
Name	FlsUseInterrupts {FLS_USE_INTERRUPTS}		
Description	Job processing triggered by hardware interrupt. true: Job processing triggered by interrupt (hardware controlled). false: Job processing not triggered by interrupt (software controlled)		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module dependency: Only available if supported by underlying flash hardware		

SWS Item	**FLS293 :**		
Name	FlsVersionInfoApi {FLS_VERSION_INFO_API}		
Description	Pre-processor switch to enable / disable the API to read out the modules version information. true: Version info API enabled. false: Version info API disabled.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

No Included Containers

10.2.4 FlsConfigSet

SWS Item	**FLS174 :**
Container Name	FlsConfigSet{Fls_ConfigSet} [Multi Config Container]
Description	Container for runtime configuration parameters of the flash driver. Implementation Type: Fls_ConfigType.
Configuration Parameters	

SWS Item	**FLS270 :**		
Name	FlsAcErase {FLS_AC_ERASE}		
Description	Address offset in RAM to which the erase flash access code shall be loaded. Used as function pointer to access the erase flash access code.		
Multiplicity	1		
Type	FunctionNameDef		
Default value	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module		

SWS Item	**FLS271 :**		
Name	FlsAcWrite {FLS_AC_WRITE}		
Description	Address offset in RAM to which the write flash access code shall be loaded. Used as function pointer to access the write flash access code.		
Multiplicity	1		
Type	FunctionNameDef		
Default value	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

SWS Item	**FLS272 :**		
Name	FlsCallCycle {FLS_CALL_CYCLE}		
Description	Cycle time of calls of the flash driver's main function.		
Multiplicity	1		
Type	FloatParamDef		
Range	0 .. INF		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: Only relevant if deadline monitoring for internal functionality has to be done in software (e.g. erase / write timings)		

SWS Item	**FLS273 :**		
Name	FlsJobEndNotification {FLS_JOB_END_NOTIFICATION}		
Description	Mapped to the job end notification routine provided by some upper layer module, typically the Fee module.		
Multiplicity	1		
Type	FunctionNameDef		
Default value	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module		

SWS Item	**FLS274 :**		
Name	FlsJobErrorNotification {FLS_JOB_ERROR_NOTIFICATION}		
Description	Mapped to the job error notification routine provided by some upper layer module, typically the Fee module.		
Multiplicity	1		
Type	FunctionNameDef		
Default value	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	

	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module		

SWS Item	**FLS275 :**		
Name	FlsMaxReadFastMode {FLS_MAX_READ_FAST_MODE}		
Description	The maximum number of bytes to read or compare in one cycle of the flash driver's job processing function in fast mode.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: The minimum number might depend on the underlying flash device or communication driver, e.g. if the access to an external flash device is done via SPI and the minimum transfer size on SPI is four bytes.		

SWS Item	**FLS276 :**		
Name	FlsMaxReadNormalMode {FLS_MAX_READ_NORMAL_MODE}		
Description	The maximum number of bytes to read or compare in one cycle of the flash driver's job processing function in normal mode.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: The minimum number might depend on the underlying flash device or communication driver, e.g. if the access to an external flash device is done via SPI and the minimum transfer size on SPI is four bytes.		

SWS Item	**FLS277 :**		
Name	FlsMaxWriteFastMode {FLS_MAX_WRITE_FAST_MODE}		
Description	The maximum number of bytes to write in one cycle of the flash driver's job processing function in fast mode.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: FLS182: This value has to correspond to the settings in FLS_PAGE_LIST. The minimum number is defined by the size of one flash page and therefore depends on the underlying flash device.		

SWS Item	**FLS278 :**
Name	FlsMaxWriteNormalMode {FLS_MAX_WRITE_NORMAL_MODE}
Description	The maximum number of bytes to write in one cycle of the flash driver's job processing function in normal mode.

Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: FLS176: This value has to correspond to the settings in FLS_PAGE_LIST. The minimum number is defined by the size of one flash page and therefore depends on the underlying flash device.		

SWS Item	**FLS279 :**		
Name	FlsProtection {FLS_PROTECTION}		
Description	Erase/write protection settings. Only relevant if supported by hardware.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: Only relevant if supported by hardware.		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
FlsSectorList	1	List of flashable sectors and pages.

FLS173: The table above specifies the parameters that shall be located in an external data structure of type `Fls_ConfigType`. The organization and location of this data structure shall be up to the implementer. The type declaration shall be located in the file `Fls.h`. Further hardware or implementation specific parameters can be added if necessary.

10.2.5 FlsSectorList

SWS Item	**FLS201 :**
Container Name	FlsSectorList{Fls_SectorList}
Description	List of flashable sectors and pages.
Configuration Parameters	

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
FlsSector	1..*	Configuration description of a flashable sector

10.2.6 FlsSector

SWS Item	**FLS202 :**
Container Name	FlsSector{Fls_Sector}
Description	Configuration description of a flashable sector
Configuration Parameters	

SWS Item	**FLS280 :**
Name	FlsNumberOfSectors {FLS_NUMBER_OF_SECTORS}
Description	Number of continuous sectors with the above characteristics.
Multiplicity	1

Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module		

SWS Item	**FLS281 :**		
Name	FlsPageSize {FLS_PAGE_SIZE}		
Description	Size of one page of this sector. Implementation Type: Fls_LengthType.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: The sector size has to be an integer multiple of the page size.		

SWS Item	**FLS282 :**		
Name	FlsSectorSize {FLS_SECTOR_SIZE}		
Description	Size of this sector. Implementation Type: Fls_LengthType.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module dependency: The sector size has to be an integer multiple of the page size.		

SWS Item	**FLS283 :**		
Name	FlsSectorStartaddress {FLS_SECTOR_STARTADDRESS}		
Description	Start address of this sector. Implementation Type: Fls_AddressType.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: module		

No Included Containers

10.3 Published Information

Published information contains data defined by the implementer of the SW module that does not change when the module is adapted (i.e. configured) to the actual HW/SW environment. It thus contains version and manufacturer information.

FLS177: The following table specifies the information that shall be published in the module's description file. Further hardware or implementation specific information can be added if necessary.

The standard common published information like

- vendorId FLS_VENDOR_ID),
- moduleId (FLS_MODULE_ID),
- arMajorVersion FLS_AR_MAJOR_VERSION),
- arMinorVersion (FLS_ AR_MINOR_VERSION),
- arPatchVersion (FLS_ AR_PATCH_VERSION),
- swMajorVersion (FLS_SW_MAJOR_VERSION),
- swMinorVersion (FLS_ SW_MINOR_VERSION),
- swPatchVersion (FLS_ SW_PATCH_VERSION),
- vendorApiInfix (FLS_VENDOR_API_INFIX)

is provided in the BSW Module Description Template (see [8], Figure 4.1 and Figure 7.1). Additional published parameters are listed below if applicable for this module.

10.3.1 FlsPublishedInformation

SWS Item	**FLS178 :**
Container Name	FlsPublishedInformation
Description	Additional published parameters not covered by CommonPublishedInformation container. Note that these parameters do not have any configuration class setting, since they are published information.
Configuration Parameters	

SWS Item	**FLS294 :**		
Name	FlsAcLocationErase {FLS_AC_LOCATION_ERASE}		
Description	Position in RAM, to which the erase flash access code has to be loaded. Only relevant if the erase flash access code is not position independent. If this information is not provided it is assumed that the erase flash access code is position independent and that therefore the RAM position can be freely configured.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

SWS Item	**FLS295 :**		
Name	FlsAcLocationWrite {FLS_AC_LOCATION_WRITE}		
Description	Position in RAM, to which the write flash access code has to be loaded. Only relevant if the write flash access code is not position independent. If this information is not provided it is assumed that the write flash access code is position independent and that therefore the RAM position can be freely configured.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants

Scope / Dependency	scope: module		

SWS Item	**FLS296 :**		
Name	FlsAcSizeErase {FLS_AC_SIZE_ERASE}		
Description	Number of bytes in RAM needed for the erase flash access code.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

SWS Item	**FLS297 :**		
Name	FlsAcSizeWrite {FLS_AC_SIZE_WRITE}		
Description	Number of bytes in RAM needed for the write flash access code.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

SWS Item	**FLS298 :**		
Name	FlsEraseTime {FLS_ERASE_TIME}		
Description	Maximum time to erase one complete flash sector.		
Multiplicity	1		
Type	FloatParamDef		
Range	-INF .. INF		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

SWS Item	**FLS299 :**		
Name	FlsErasedValue {FLS_ERASED_VALUE}		
Description	The contents of an erased flash memory cell.		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

SWS Item	**FLS300 :**		
Name	FlsExpectedHwId {FLS_EXPECTED_HW_ID}		
Description	Unique identifier of the hardware device that is expected by this driver (the device for which this driver has been implemented). Only relevant for external flash drivers.		
Multiplicity	1		
Type	StringParamDef		
Default value	--		
regularExpression	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

SWS Item	**FLS198 :**

Name	FlsSpecifiedEraseCycles {FLS_SPECIFIED_ERASE_CYCLES}		
Description	Number of erase cycles specified for the flash device (usually given in the device data sheet). FLS198: If the number of specified erase cycles depends on the operating environment (temperature, voltage, ...) during reprogramming of the flash device, the minimum number for which a data retention of at least 15 years over the temperature range from -40°C .. +125°C can be guaranteed shall be given. Note: If there are different numbers of specified erase cycles for different flash sectors of the device this parameter has to be extended to a parameter list (similar to the sector list above).		
Multiplicity	1		
Type	IntegerParamDef		
Range	..		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

SWS Item	**FLS301 :**		
Name	FlsWriteTime {FLS_WRITE_TIME}		
Description	Maximum time to program one complete flash page.		
Multiplicity	1		
Type	FloatParamDef		
Range	-INF .. INF		
Default value	--		
ConfigurationClass	***Published Information***	X	All Variants
Scope / Dependency	scope: module		

No Included Containers

FLS177: The following table specifies the information that shall be published in the module’s description file. Further hardware or implementation specific information can be added if necessary.