Integration Manual

for S32K14X FLS Driver

Document Number: IM2FLSASR4.2 Rev0002R1.0.2
Rev. 1.0

Contents

Section number **Title** **Page**

Chapter 1
Revision History

Chapter 2
Introduction

Chapter 3
Building the Driver

Chapter 4
Function calls to module

Chapter 5
Module requirements

Chapter 1
Revision History

Table 1-1. Revision History

Revision	Date	Author	Description
1.0	26/04/2019	NXP MCAL Team	Updated version for ASR 4.2.2S32K14XR1.0.2

Chapter 2
Introduction

This integration manual describes the integration requirements for FLS Driver for S32K14X microcontrollers.

2.1 Supported Derivatives

The software described in this document is intented to be used with the following microcontroller devices of NXP Semiconductors .

Table 2-1. S32K14X Derivatives

NXP Semiconductors	s32k148_lqfp144, s32k148_lqfp176, s32k148_mapbga100, s32k146_lqfp144, s32k146_lqfp100, s32k146_lqfp64, s32k146_mapbga100, s32k144_lqfp100, s32k144_lqfp64, s32k144_mapbga100, s32k142_lqfp100, s32k142_lqfp64, s32k118_lqfp48, s32k118_lqfp64, s32k142_lqfp48, s32k144_lqfp48, s32k148_lqfp100

All of the above microcontroller devices are collectively named as S32K14X .

2.2 Overview

AUTOSAR (AUTomotive Open System ARchitecture) is an industry partnership working to establish standards for software interfaces and software modules for automobile electronic control systems.

AUTOSAR

- paves the way for innovative electronic systems that further improve performance, safety and environmental friendliness.

- is a strong global partnership that creates one common standard: "Cooperate on standards, compete on implementation".
- is a key enabling technology to manage the growing electrics/electronics complexity. It aims to be prepared for the upcoming technologies and to improve cost-efficiency without making any compromise with respect to quality.
- facilitates the exchange and update of software and hardware over the service life of the vehicle.

2.3 About this Manual

This Technical Reference employs the following typographical conventions:

Boldface type: Bold is used for important terms, notes and warnings.

Italic font: Italic typeface is used for code snippets in the text. Note that C language modifiers such "const" or "volatile" are sometimes omitted to improve readability of the presented code.

Notes and warnings are shown as below:

Note

This is a note.

2.4 Acronyms and Definitions

Table 2-2. Acronyms and Definitions

Term	Definition
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
DET	Default Error Tracer
ECC	Error Correcting Code
VLE	Variable Length Encoding
N/A	Not Applicable
MCU	Micro Controller Unit
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
FEE	Flash EEPROM Emulation
FLS	Flash
XML	Extensible Markup Language

2.5 Reference List

Table 2-3. Reference List

#	Title	Version
1	Specification of FLS Driver	AUTOSAR Release 4.2.2
2	S32K14X Reference Manual	Reference Manual, Rev. 9, 9/2018
3	S32K142 Mask Set Errata for Mask 0N33V (0N33V)	30/11/2017
4	S32K144 Mask Set Errata for Mask 0N57U (0N57U)	30/11/2017
5	S32K146 Mask Set Errata for Mask 0N73V (0N73V)	30/11/2017
6	S32K148 Mask Set Errata for Mask 0N20V (0N20V)	25/10/2018
7	S32K118 Mask Set Errata for Mask 0N97V (0N97V)	07/01/2019

Chapter 3
Building the Driver

This section describes the source files and various compilers, linker options used for building the Autosar FLS driver for NXP SemiconductorsS32K14X . It also explains the EB Tresos Studio plugin setup procedure.

3.1 Build Options

The FLS driver files are compiled using

- Green Hills Multi 7.1.4 / Compiler 2017.1.4
- (Linaro GCC 6.3-2017.06~dev) 6.3.1 20170509 (Wed Jan 24 16:21:45 CST 2018 build.sh rev=g27a1317 s=L631 Earmv7 -V release_g27a1317_build_Fed_Earmv7)
- IAR: V8.11.2

The compiler, linker flags used for building the driver are explained below:

Note

The TS_T40D2M10I2R0 plugin name is composed as follow:

TS_T = Target_Id

D = Derivative_Id

M = SW_Version_Major

I = SW_Version_Minor

R = Revision

(i.e. Target_Id = 40 identifies CORTEXM architecture and Derivative_Id = 2 identifies the S32K14X)

3.1.1 GHS Compiler/Linker/Assembler Options

Table 3-1. Compiler Options

Option	Description
-cpu=cortexm4	Selects target processor: Arm Cortex M4
-cpu=cortexm0plus	Selects target processor: Arm Cortex M0+
-ansi	Specifies ANSI C with extensions. This mode extends the ANSI X3.159-1989 standard with certain useful and compatible constructs.
-Osize	Optimize for size.
-dual_debug	Enables the generation of DWARF, COFF, or BSD debugging information in the object file
-G	Generates source level debugging information and allows procedure call from debugger's command line.
--no_exceptions	Disables support for exception handling
-Wundef	Generates warnings for undefined symbols in preprocessor expressions
-Wimplicit-int	Issues a warning if the return type of a function is not declared before it is called
-Wshadow	Issues a warning if the declaration of a local variable shadows the declaration of a variable of the same name declared at the global scope, or at an outer scope
-Wtrigraphs	Issues a warning for any use of trigraphs
-Wall	Enables all the warnings about constructions that some users consider questionable, and that are easy to avoid even in conjunction with macros.
--prototype_errors	Generates errors when functions referenced or called have no prototype
--incorrect_pragma_warnings	Valid #pragma directives with wrong syntax are treated as warnings
-noslashcomment	C++ like comments will generate a compilation error
-preprocess_assembly_files	Preprocesses assembly files
-nostartfile	Do not use Start files
--short_enum	Store enumerations in the smallest possible type
-c	Produces an object file (called input-file.o) for each source file.
--no_commons	Allocates uninitialized global variables to a section and initializes them to zero at program startup.
-keeptempfiles	Prevents the deletion of temporary files after they are used. If an assembly language file is created by the compiler, this option will place it in the current directory instead of the temporary directory. Produces an object file (called input-file.o) for each source file.
-list	Creates a listing by using the name of the object file with the .lst extension. Assembler option
-DAUTOSAR_OS_NOT_USED	-D defines a preprocessor symbol and optionally can set it to a value. AUTOSAR_OS_NOT_USED: By default in the package, the drivers are compiled to be used without Autosar OS. If the drivers are used with Autosar OS, the compiler option '-DAUTOSAR_OS_NOT_USED' must be removed from project options
-DDISABLE_MCAL_INTERMODULE_ASR_CHECK	-D defines a preprocessor symbol to disable the inter-module version check for AR_RELEASE versions. DISABLE_MCAL_INTERMODULE_ASR_CHECK: By default in the package, drivers are compiled to perform the inter-module version check as per Autosar BSW004. When the inter-module version check needs to be disabled then the DISABLE_MCAL_INTERMODULE_ASR_CHECK global define must be added to the list of compiler options.
-DGHS	-D defines a preprocessor symbol and optionally can set it to a value. This one defines the GHS preprocessor symbol.

Table 3-2. Assembler Options

Option	Description
-cpu=cortexm4	Selects target processor: Arm Cortex M4
-cpu=cortexm0plus	Selects target processor: Arm Cortex M0+
-c	Produces an object file (called input-file.o) for each source file.
-preprocess_assembly_files	Preprocesses assembly files
-asm=list	Creates a listing by using the name of the object file with the .lst extension. Assembler option

Table 3-3. Linker Options

Option	Description
-Mn	Map file numeric ordering
-delete	Removal from the executable of functions that are unused and unreferenced
-v	Display removed unused functions
-ignore_debug_references	Ignores relocations from DWARF debug sections when using -delete.
-map	Creates a detailed map file
-keepmap	Keep the map file in the event of a link error
-lstartup	Link libstartup library -Run-time environment startup routines
-lsys	Link libsys library -Run-time environment system routines
-larch	Link libarch library -Target-specific run-time support. Any file produced by the Green Hills Compiler may depend on symbols in this library.
-lansi	Link libansi library -the standard C library
-L(/lib/thumb2)	Link thumb2 library
-lutf8_s32	Include utf8_s32.a to use the Wide Character Functions

3.1.2 IAR Compiler/Linker/Assembler Options

Table 3-4. Compiler Options

Option	Description
--cpu=Cortex-M4	Selects target processor: Arm Cortex M4
--cpu=Cortex-M0+	Selects target processor: Arm Cortex M0+
--cpu_mode=thumb	Selects generating code that executes in Thumb state.
--endian=little	Specifies the endianess of core: little endian.
-Ohz	Sets the optimization level to High, favoring size.
-c	Produces an object file (called input-file.o) for each source file.
--no_clustering	Disables static clustering optimizations.
--no_mem_idioms	Makes the compiler to not optimize code sequences that clear, set, or copy a memory region.
--no_explicit_zero_opt	Places the zero initialized variables in data section instead of bss.
--debug	Makes the compiler include information in the object modules.

Table continues on the next page...

Table 3-4. Compiler Options (continued)

Option	Description
--diag_suppress=Pa050	Suppresses diagnostic messages (warnings) about non-standard line endings.
-DAUTOSAR_OS_NOT_USED	-D defines a preprocessor symbol and optionally can set it to a value. AUTOSAR_OS_NOT_USED: By default in the package, the drivers are compiled to be used without Autosar OS. If the drivers are used with Autosar OS, the compiler option '-DAUTOSAR_OS_NOT_USED' must be removed from project options
-DIAR	-D defines a preprocessor symbol and optionally can set it to a value. This one defines the IAR preprocessor symbol.
--require_prototypes	Forces the compiler to verify that all functions have proper prototypes.
--no_wrap_diagnostics	Disables line wrapping of diagnostic messages issued by compiler.
--no_system_include	Disables the automatic search for system include files.
-e	Enables language extensions. This option is needed by FLS driver which uses _packed structures.

Table 3-5. Assembler Options

Option	Description
--cpu=Cortex-M4	Selects target processor: Arm Cortex M4
--cpu=Cortex-M0+	Selects target processor: Arm Cortex M0+
--cpu_mode=thumb	Selects generating code that executes in Thumb state.
-g	Use this option to disable the automatic search for system include files.

Table 3-6. Linker Options

Option	Description
--cpu=Cortex-M4	Selects target processor: Arm Cortex M4
--cpu=Cortex-M0+	Selects target processor: Arm Cortex M0+
--map filename	Produces a map file.
--no_library_search	Disables automatic runtime library search.
--entry _start	Treats the symbol _start as a root symbol and as the start of the application.
--enable_stack_usage	Enables stack usage analysis.
--skip_dynamic_initialization	Suppress dynamic initialization during system startup.
--no_wrap_diagnostics	Disables line wrapping of diagnostic messages issued by linker.
--config	Specifies the configuration file to be used by the linker.

3.1.3 GCC Compiler/Linker/Assembler Options

Table 3-7. Compiler Options

Option	Description
-c	Produces an object file (called input-file.o) for each source file.
-Os	Use optimization for size.
-ggdb3	Produce debugging information for use by GDB. Level 3 includes extra information, such as all the macro definitions present in the program.
-mcpu=cortex-m4	Selects target processor: Arm Cortex M4
-mcpu=cortex-m0plus	Selects target processor: Arm Cortex M0+
-mthumb	Selects generating code that executes in Thumb state.
-ansi	Specifies ANSI C with extensions.
-mlittle-endian	Generate code for a processor running in little-endian mode.
-fomit-frame-pointer	Removes the frame pointer for all functions, which might make debugging harder.
-msoft-float	Use software floating-point instructions.
-fno-common	Specifies that the compiler should place uninitialized global variables in the data section of the object file, rather than generating them as common blocks.
-Wall	Enables all the warnings about constructions that some users consider questionable, and that are easy to avoid even in conjunction with macros.
-Wextra	Enables some extra warning flags that are not enabled by '-Wall'.
-Wstrict-prototypes	Warn if a function is declared or defined without specifying the argument types.
-Wno-sign-compare	Do not warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.
-fstack-usage	Geneates an extra file that specifies the maximum amount of stack used, on a per-function basis.
-fdump-ipa-all	Enables all inter-procedural analysis dumps.
-Werror=implicit-function-declaration	Generates an error when the prototype of the function is not defined..
-DAUTOSAR_OS_NOT_USED	-D defines a preprocessor symbol and optionally can set it to a value. AUTOSAR_OS_NOT_USED: By default in the package, the drivers are compiled to be used without Autosar OS. If the drivers are used with Autosar OS, the compiler option '-DAUTOSAR_OS_NOT_USED' must be removed from project options
-DGCC	-D defines a preprocessor symbol and optionally can set it to a value. This one defines the GCC preprocessor symbol.

Table 3-8. Assembler Options

Option	Description
-mcpu=cortex-m4	Selects target processor: Arm Cortex M4
-mcpu=cortex-m0plus	Selects target processor: Arm Cortex M0+
-c	Produces an object file (called input-file.o) for each source file.
-mthumb	This option specifies that the assembler should start assembling Thumb instructions.
-x assembler-with-cpp	Indicates that the assembly code contains C directives and the C preprocessor must be run.

Table 3-9. Linker Options

Option	Description
-Map=filename	Print a link map to the file mapfile.
-T scriptfile	Use scriptfile as the linker script. This script replaces ld's default linker script(rather than adding to it), so commandfile must specify everything necessary to describe the output file.
--disable-newlib-supplied-syscalls -specs=nosys.specs	These options support for using newlib on core M0+
-u _printf_float -u _scanf_float	These options support generating profile report.
-nostartfiles	Do not use the standard system startup files when linking
-e _start	Specify that the program entry point is _start
-static	The --static flag tells the linker to link a static, not a dynamically linked
-lc	The -lc flag tells the linker to link this binary against the C library, which is newlib in our case.
-lnosys	The -lnosys flag tells the linker to link this binary against the "nosys" library
$(TOOLCHAIN_DIR)/arm-none-eabi/newlib/lib/thumb/v6-m $(TOOLCHAIN_DIR)/lib/gcc/arm-none-eabi/6.3.1/thumb/v6-m	Library for core M0+
$(TOOLCHAIN_DIR)/arm-none-eabi/newlib/lib/thumb $(TOOLCHAIN_DIR)/arm-none-eabi/newlib/lib)	Library for core M4

3.2 Files required for Compilation

This section describes the include files required to compile, assemble (if assembler code) and link the FLS driver for S32K14X microcontrollers.

To avoid integration of incompatible files, all the include files from other modules shall have the same AR_MAJOR_VERSION and AR_MINOR_VERSION, i.e. only files with the same AUTOSAR major and minor versions can be compiled.

FLS Files

- ..\Fls_TS_T40D2M10I2R0\src\Fls.c
- ..\Fls_TS_T40D2M10I2R0\src\Fls_Ac.c
- ..\Fls_TS_T40D2M10I2R0\src\Fls_Flash.c
- ..\Fls_TS_T40D2M10I2R0\src\Fls_Flash_Const.c
- ..\Fls_TS_T40D2M10I2R0\src\Fls_IPW.c
- ..\Fls_TS_T40D2M10I2R0\src\Fls_Qspi.c
- ..\Fls_TS_T40D2M10I2R0\src\Fls_Qspi_Irq.c
- ..\Fls_TS_T40D2M10I2R0\include\Fls.h

- ..\Fls_TS_T40D2M10I2R0\include\Fls_Api.h
- ..\Fls_TS_T40D2M10I2R0\include\Fls_Flash.h
- ..\Fls_TS_T40D2M10I2R0\include\Fls_Flash_Types.h
- ..\Fls_TS_T40D2M10I2R0\include\Fls_IPW.h
- ..\Fls_TS_T40D2M10I2R0\include\Fls_Qspi.h
- ..\Fls_TS_T40D2M10I2R0\include\Fls_Qspi_Types.h
- ..\Fls_TS_T40D2M10I2R0\include\Fls_Types.h
- ..\Fls_TS_T40D2M10I2R0\include\Reg_eSys_FLASHC.h
- ..\Fls_TS_T40D2M10I2R0\include\Reg_eSys_QSPI.h

- Fls_PBcfg.c - this file should be generated by the user using a configuration/ generation tool
- Fls_Cfg.h - this file should be generated by the user using a configuration/generation tool
- Fls_Cfg.c - this file should be generated by the user using a configuration/generation tool

Other includes files:

Files from MemIf folder:
- ..\MemIf_TS_T40D2M10I2R0\include\MemIf_Types.h

Files from Base common folder
- ..\Base_TS_T40D2M10I2R0\include\Compiler.h
- ..\Base_TS_T40D2M10I2R0\include\Compiler_Cfg.h
- ..\Base_TS_T40D2M10I2R0\include\ComStack_Types.h
- ..\Base_TS_T40D2M10I2R0\include\Fls_MemMap.h
- ..\Base_TS_T40D2M10I2R0\include\Mcal.h
- ..\Base_TS_T40D2M10I2R0\include\Platform_Types.h
- ..\Base_TS_T40D2M10I2R0\include\Std_Types.h
- ..\Base_TS_T40D2M10I2R0\include\Reg_eSys.h
- ..\Base_TS_T40D2M10I2R0\include\Soc_Ips.h
- ..\Base_TS_T40D2M10I2R0\include\Reg_Macros.h
- ..\Base_TS_T40D2M10I2R0\include\SilRegMacros.h

Files from Det folder:
- ..\Det_TS_T40D2M10I2R0\include\Det.h

Files from RTE folder:
- ..\Rte_TS_T40D2M10I2R0\include\SchM_Fls.h

Files from Mcl folder:
- ..\Mcl_TS_T40D2M10I2R0\include\Mcl.h

3.3 Setting up the Plug-ins

The FLS driver was designed to be configured by using the EB Tresos Studio (version EB tresos Studio 23.0.0 b170330-0431 or later.)

- VSMD (Vendor Specific Module Definition) file in EB tresos Studio XDM format: .. \Fls_TS_T40D2M10I2R0\config\Fls.xdm
- VSMD (Vendor Specific Module Definition) file in AUTOSAR compliant EPD format: ..\Fls_TS_T40D2M10I2R0\autosar\(one EPD file for each supported sub-derivative)
- Code Generation Templates for variant aware parameters:
  - ..\Fls_TS_T40D2M10I2R0\generate_PB\src\Fls_PBcfg.c
- Code Generation Templates for parameters without variation points:
  - ..\Fls_TS_T40D2M10I2R0\generate_PC\include\Fls_Cfg.h
  - ..\Fls_TS_T40D2M10I2R0\generate_PC\src\Fls_Cfg.c

Steps to generate the configuration:

1. Copy the module folders Fls_TS_T40D2M10I2R0 , Base_TS_T40D2M10I2R0 , Mcl_TS_T40D2M10I2R0 , Det_TS_T40D2M10I2R0 , Rte_TS_T40D2M10I2R0 , EcuC_TS_T40D2M10I2R0 and Resource_TS_T40D2M10I2R0 into the Tresos plugins folder.
2. Set the desired Tresos Output location folder for the generated sources and header files.
3. Use the EB tresos Studio GUI to modify ECU configuration parameters values.
4. Generate the configuration files.

Chapter 4
Function calls to module

4.1 Function Calls during Start-up

FLS shall be initialized during STARTUP phase of EcuM initialization. The API to be called for this is Fls_Init().

The MCU module should be initialized before the FLS is initialized.

If the FLS driver is used in User Mode, be sure that the Flash memory controller registers are accessible and that accessed Flash memory partition is not protected. For more information please refer to the "Memory Protection Unit" and "Register Protection" chapters in the device reference manual.

The Flash memory physical sectors that are going to be modified by Fls driver (i.e. erase and write operations) have to be unprotected for a successful operation.

Note: if the configured internal flash sectors are protected, the flash driver initialization will fail. The driver is unable to unprotect any protected sectors. Example of unprotect code can be found in Fls driver UM document.

Note: the flash initialization for external(qspi) sectors does not check the protection status of the configured sectors. The application has to ensure that at the end of the initialization, the configured external sectors are in a state which allows the attempted driver operations. In order to aid the configuration of the external memory, at the end of the QSPI IP configuration the FlsQspiInitCallout callout can be implemented in order to configure the appropriate protection settings in the external memory.

If the parameter FlsCheckFlexNvmRatio is checked, the Flash memory FlexNVM partition has to be partitioned for Data flash only, no EEPROM emulation.

Note: if the FlexNVM partition is not configured for Data flash only and 'FlsCheckFlexNvmRatio' configuration parameter switch is checked, the flash driver initialization will fail. The driver is unable to modify the partition. If the FlexNVM

partition is not configured for Data flash only and 'FlsCheckFlexNvmRatio' configuration parameter switch is un-checked, the flash driver initialization will not check the FlexNvm ratio. In this case, only those Data flash sectors which are available in hardware should be configured.

Note: in order to allow the execution of EEP, CSEC drivers, which require D-Flash partitioning, the FlsCheckFlexNvmRatio has to be disabled.

4.2 Function Calls during Shutdown

None.

4.3 Function Calls during Wake-up

None.

4.4 Implementing an Exception Handler in case of non correctable ECC error

When reading an internal FLASH location with a non correctable ECC error a HardFault/ BusFault/DFDIF is thrown.

The size of the internal flash ECC page varies from 8 to 16 bytes, depending on the read path width of the flash block. This restriction, alongside with fact that the cache line size is 16 bytes long, leads to the recommendation of aligning all write operations to 16 bytes, especially in FEE driver in order to avoid corruption of adjacent blocks.

The Flash driver provides an API which can be called inside the user ECC error recovery handler (Hard fault handler driver functions):

- **Fls_CompHandlerReturnType Fls_DsiHandler (Fls_ExceptionDetailsType *);**

This API is available only if the configuration parameter **FlsDsiHandlerApi = true**;

The **Fls_ExceptionDetailsType** data structure contains some information about the details of Exception and in particular:

- a pointer to the statement that generated the ECC (**Fls_InstructionAddressType**);

- an address of the data that caused the error in ECC (**Fls_DataAddressType**);
- details on the type of exception (**uint32**).

Interrupt handler Driver function examines the job executed by the driver and the data contained in the structure **Fls_ExceptionDetailsType** in particular:

- Check whether there is pending read or compare job;
- Check if exception syndrome Indicates Double bit ECC fault reason;
- Data address which cause the exception matches address currently accessed by pending flash read or flash compare job.

If these conditions are verified the interrupt handler driver functions return **FLS_HANDLED_SKIP** and set the job to failed value. This information can be retrieved using **Fls_GetJobResult()** which will return **MEMIF_JOB_FAILED** or, if the job error notification parameter is configured, the notification function will be called.Otherwise, **FLS_UNHANDLED** will be returned

With these information the following recovery strategies may be implemented:

- skip the instruction that caused the error (**FLS_HANDLED_SKIP**);
- perform a controlled shutdown of current activity, do nothing (infinite loop), etc. (**FLS_UNHANDLED**);

In addition to this information, a basic flow and an implementation idea for the fault handler is depicted below. Please take into account that the code below is just an example and it is not guaranteed to work in all scenarios or not have hidden problems.

Note: On S32K118 derivative does not be checked syndrome and data address because it uses the M0++ core missing some registers related to read syndrome(CFSR) and data address(BFAR) */

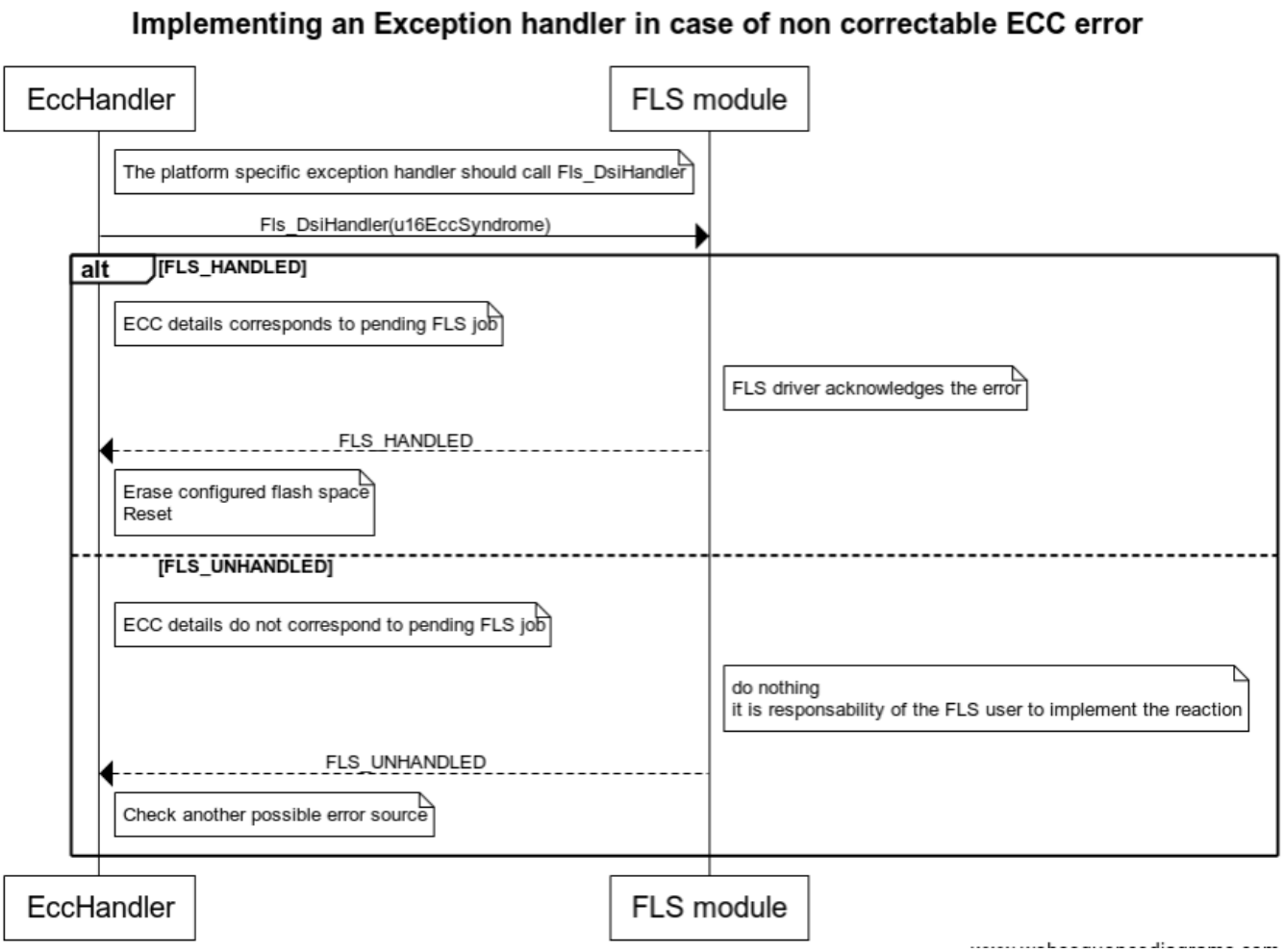

Figure 4-1. Example of ECC Handling

4.5 ECC Management on QSPI Flash

The ECC management on QSPI sectors is dependent on the external memory specific implementation. The driver offers FlsQspiEccCheckCallout, called at the end of each read job to offer the possibility to the application to interrogate the external memory error status.

Depending on the hardware resources available on the external memory, the application can enable and implement the ECC callout in order to check any error bits available and return a failed value in order to mark the current job as failed.

Chapter 5
Module requirements

5.1 Exclusive areas to be defined in BSW scheduler

In the current implementation, FLS is using the services of Run-Time Environment (RTE) for entering and exiting the critical regions. RTE implementation is done by the integrators of the MCAL using OS or non-OS services. For testing the FLS, stubs are used for RTE.

FLS driver has six exclusive areas (EA) FLS_EXCLUSIVE_AREA_00, FLS_EXCLUSIVE_AREA_10, FLS_EXCLUSIVE_AREA_11, FLS_EXCLUSIVE_AREA_12 , FLS_EXCLUSIVE_AREA_13 and FLS_EXCLUSIVE_AREA_14. The purpose of the exclusive areas FLS_EXCLUSIVE_AREA_[10-14] is to make the functions Fls_Erase, Fls_Write, Fls_Read, Fls_Compare and Fls_BlankCheck thread safe and thus protect FLS internal job variables.

The purpose of exclusive area FLS_EXCLUSIVE_AREA_00 is to protect the shared resources of the FTFC flash IP and thus allow the execution of FLS/EEP/CSEC drivers. This exclusive area is placed inside Fls_MainFunction and it is entered for ERASE and WRITE jobs in SYNC mode. This critical region should not be interrupted by the correspondent critical region from EEP or CSEC driver, in order to preserve the values of the FTFC hardware registers(error status, FCCOB command codes).

5.1.1 Critical Region Exclusive Matrix

Below is the table depicting the exclusivity between different critical region IDs from the FLS driver. If there is an “X” in a table, it means that those 2 critical regions cannot interrupt each other.

FLS_EXCLUSIVE_AREA_00 Used in Fls_MainFunction.

FLS_EXCLUSIVE_AREA_10 Used in Fls_Erase.

FLS_EXCLUSIVE_AREA_11 Used in Fls_Write.

FLS_EXCLUSIVE_AREA_12 Used in Fls_Read.

FLS_EXCLUSIVE_AREA_13 Used in Fls_Compare.

FLS_EXCLUSIVE_AREA_14 Used in Fls_BlankCheck.

Table 5-1. Exclusive Areas

	FLS_EXCLUSIVE_AREA_00	FLS_EXCLUSIVE_AREA_10	FLS_EXCLUSIVE_AREA_11	FLS_EXCLUSIVE_AREA_12	FLS_EXCLUSIVE_AREA_13	FLS_EXCLUSIVE_AREA_14
FLS_EXCLUSIVE_AREA_00	x					
FLS_EXCLUSIVE_AREA_10			x	x	x	x
FLS_EXCLUSIVE_AREA_11		x		x	x	x
FLS_EXCLUSIVE_AREA_12		x	x		x	x
FLS_EXCLUSIVE_AREA_13		x	x	x		x
FLS_EXCLUSIVE_AREA_14		x	x	x	x	

5.1.2 Flash access notifications

Two configurable notifications are present, which guard the read and program(write,erase) access to flash: "FlsStartFlashAccessNotif" and "FlsFinishedFlashAccessNotif".

These notifications are used to make Fls_MainFunction Thread Safe. When used for program, an intended purpose is to avoid any read-while-write errors that could be generated by the execution of code from flash by different masters. When used for read, on data flash sectors which do not trigger ECC exceptions, an intended purpose is to ensure that an ECC error was reported by the driver read.

Note: These notifications are the result of unclear Fls SWS document requirement number SWS_Fls_00215.

SWS_Fls_00215: “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).”

On the contrary no BSW module is allowed directly control the global ECU interrupts, the Rte (and OS) module or other mechanisms shall be used for this purposes. The actual implementation/behavior of these notifications is left on the ECU integrator. It means in case no other executed code (task-s) access the ‘code’ or ‘constant data’ from affected Flash area (sector-s) which is being modified by current Fls job (erase or write operations) then the implementation could be ‘void’ (as there is no Flash read-while-write error possible). Also, in case of read, if there is no other master accessing the same flash area or if there is no need to exclusively link an ECC error to the flash driver read, then the implementation could be ‘void’ also. In all other cases you have to block the execution of the code (task-s) which would access this affected Flash area (sector-s).

5.2 Peripheral Hardware Requirements

The FLS driver uses the "Flash Memory" MCU peripheral(FTFC) and the external flash memory peripheral (QuadSPI). For more details about peripherals and their structure refer to MCU reference manual.

Based on the configured type of sectors(external or internal), the FLS driver will spread a job across both internal and external memories. The hardware IP used to complete a job section is decided based on the current sector type. For consistent operations it is recommended to configure and allocate jobs on a single type of memory and avoid mixing internal and external sectors.

Attempts to launch an FTFC command in VLP and HSRUN mode is not supported. For more details, please refer to the reference manual.

FLS flash operations and CSEc or EEP(FlexNVM) operations cannot execute simultaneously, as they are sharing common resources(status bit, flash memory, etc.). In order to avoid conflicts(read-while-write errors, access errors, etc), these drivers should implement the available exclusive area so that the critical regions of all drivers do not interrupt each other. It is recommended to structure the FLS/EEP/CSEC jobs on separate time slots so that there is no attempted simultaneous access to the IP resources. For more details about shared resources, please consult the Reference manual.

If external sectors are to be used, the FLS driver will configure the QuadSPI IP. Prior to FLS driver initialization, the QuadSPI IP should be enabled(clock, power) as required by the application, to a state in which the the FLS driver can configure and use it. All platform settings or specific external memory settings are out of FLS driver scope and are expected to be completed before any job is attempted.

Given the large diversity of implementation solutions in the external memories, the following operations are not fully implemented at the FLS driver level for external sectors: External sector protection/locking/password, External memory reset/cancel, External memory calibration/data learning, External memory error check. In order to aid implementation, callout functions and external APIs are provided in which the application could choose the best approach adapted to attached external memory. More details are provided in the following sections and User Manual.

Sector page size for external memories must be configured to a minimum value and alignment of 16 bytes. Choosing a smaller value or smaller alignment is strongly discouraged because of the internal QuadSPI IP restrictions, which require a minimum write size of 16 bytes for each transaction.

5.3 ISR to configure within OS – dependencies

The following ISR's are used by the FLS driver:

The ISR table is presented below. Depending on the derivative used, some of the ISRs may not be available. For complete details please consult the Reference Manual:

Table 5-2. QuadSPI 0 Interrupts

QuadSPI 0 Interrupts	Hardware interrupt vector
FLS_QSPI_ISR	65

NOTE

In case of AUTOSAR_OS_NOT_USED, the compiler option "-DUSE_HW_VECTOR_MODE" must be added to the list of compiler options to be used with interrupt controller configured to be in hardware vector mode.

5.4 ISR Macro

MCAL drivers use the ISR macro to define the functions that will process hardware interrupts. Depending on whether the OS is used or not, this macro can have different definitions:

a. OS is not used - AUTOSAR_OS_NOT_USED is defined:

i. If USE_SW_VECTOR_MODE is defined:

```
#define ISR(IsrName) void IsrName(void)
```

In this case, drivers' interrupt handlers are normal C functions and the prolog/epilog handle the context save and restore.

ii. If USE_SW_VECTOR_MODE is not defined:

```
#define ISR(IsrName) INTERRUPT_FUNC void IsrName(void)
```

In this case, drivers' interrupt handlers must save and restore the execution context.

Custom OS is used - AUTOSAR_OS_NOT_USED is not defined

```
#define ISR(IsrName) void OS_isr_##IsrName()
```

In this case, OS is handling the execution context when an interrupt occurs. Drivers' interrupt handlers are normal C functions.

Other vendor's OS is used - AUTOSAR_OS_NOT_USED is not defined. Please refer to the OS documentation for description of the ISR macro.

5.5 Other AUTOSAR modules - dependencies

- **Base** The BASE module contains the common files/definitions needed by all MCAL modules.
- **Det** The DET module is used for enabling Development error detection. The API function used are Det_ReportError() or Det_ReportRuntimeError() . The activation / deactivation of Development error detection is configurable using the " FlsDevErrorDetect " or " FlsRuntimeErrorDetect " configuration parameter.
- **Rte:** The RTE module is needed for implementing data consistency of exclusive areas that are used by FLS module.
- **MemIf:** This module allows the NVRAM manager to access several memory abstraction modules.
- **Resource:** Resource module is used to select microcontroller's derivatives.
- **EcuC:** The ECUC module is used for ECU configuration. MCAL modules need ECUC to retrieve the variant information.
- **HardFault/BusFault/DFDIF:** (only if FlsDsiHandlerApi=true) depending on the interrupt configuration.
- **Mcl:** This module provides service for Cache operation.

5.6 Data Cache Restriction

The FLS driver needs to maintain the memory coherency by means of three methods:

1. Disable data cache, or

2. Configure the flash region upon which the driver operates, as non-cacheable, or

3. Enable the FlsSynchronizeCache feature.

Depending on the application configuration and requirements, one option may be more beneficial than other.

If FlsSynchronizeCache parameter is enabled in the configuration, then the FLS driver will call Mcl cache API functions in order to invalidate the cache after each high voltage operation(write,erase)and before each read operation in order to ensure that the cache and the modified flash memory are in sync. The driver will attempt to invalidate only the modified lines from the cache.If the size of the region to be invalidated is greater than half of the cache size,then the entire cache is invalidated.

If FlsSynchronizeCache parameter is disabled, the upper layers have to ensure that the flash region upon which the driver operates is not cached.This can be obtained by either disabling the data cache or by configuring the memory region as non-cacheable.

The cache settings apply to internal flash operations.

Note: Failure to properly inhibit or disable data cache on the flash sectors which the driver operates on, will most likely lead to flash jobs failing. This situation is more likely to be visible when all the check functionality (Fls Erase Blank Check, Fls Write Blank Check, Fls Write Verify Check) is enabled, because more read/program sequences occur and the cache is less likely to be self cleared.

Note: On S32K148 derivative, the cache region allocated to D-Flash memory space has to be set as cache-inhibit, otherwise any access to that space could lead to cache corruption.

5.7 User Mode Support

The Fls module can be run from user mode if **FlsEnableUserModeSupport** is enabled in the configuration.

Fls module will adapt and call the Fls_Flash_InvalidPrefetchBuff_Ram() as trusted function.

5.8 Flash Configuration Field

The program flash memory contains a 16-byte flash configuration field that stores default protection settings (loaded on reset) and security information that allows the MCU to restrict access to the FTFC module. The configuration field is located at address 0x0_0400 - 0x0_040F, in program flash sector FLS_CODE_ARRAY_0_BLOCK_0_S000.

5.8.1 Sector protection

The default state for all flash internal sectors is unsecured (corresponding to erase state, 0xFF, of the flash configuration field). During reset, the reset protection values are loaded from addresses 0x0_0408 - 0x0_040B and 0x0_040F into FPROTx and FDPROT registers. A bit value of '1' means sector unprotected. If a configured sector is protected and marked as unlocked in the configuration file, the module initialization (Fls_Init) will fail. If a configured sector is protected and not marked as unlocked in the configuration file, the module write/erase attempt will fail. For a protected sector to be unprotected, its corresponding bit position from the flash configuration field has to be erased(set to '1') and a restart has to occur so its value is updated in FPROTx, FDPROT register.

Note: an example of unprotecting the flash internal sectors can be found in Fls driver UM document.

The FLS driver does not configure or check the protection status of the external flash sectors. At the end of the QuadSPI IP initialization, the FLS driver provides a callout function(FlsQspiInitCallout) which can be used by the application to check or modify the current status of the external sectors. The driver assumes and requires that the configured external sectors are configured in an usable state at the end of the initialization.

5.8.2 Chip security

The chip security state determines the debugger access to flash. The security state is loaded upon reset from the flash configuration field(FSEC, address 0x0_040C). The reset state of the chip is unsecure, corresponding to a FSEC byte value of 0xFE. The FSEC byte address is the only flash memory region with a default value different than 0xFF. Because of this, erasing the sector in which the flash configuration field resides(FLS_CODE_ARRAY_0_BLOCK_0_S000) and not reprogramming the FSEC byte with 0xFE, will lead to a secured chip upon the next reset(the FSEC value 0xFF corresponds to a secured state

Note: to unsecure a chip, a mass-erase operation has to be triggered by a debugger access. In case the mass-erase debugger option is disabled (FSEC[MEEN] = b10), debugger access to flash is permanently locked.

Chapter 6
Main API Requirements

6.1 Main functions calls within BSW scheduler

Fls_MainFunction (call rate depends on target application, i.e. how fast the data needs to be read/written/compared into Flash memory).

6.2 API Requirements

None

6.3 Calls to Notification Functions, Callbacks, Callouts

The FLS driver provides notifications that are user configurable:

- FlsAcCallback (usually routed to Wdg module durin a long high voltage operation)
- FlsJobEndNotification (usually routed to Fee module)
- FlsJobErrorNotification (usually routed to Fee module)
- FlsStartFlashAccessNotif (used, if needed, to mark the start of a flash read,program access)
- FlsFinishedFlashAccessNotif (used, if needed, to mark the end of a flash read,program access)
- FlsQspiInitCallout (used, if needed, to perform additional checks and configurations on the external memory at the end of FLS initialization)
- FlsQspiResetCallout (used, if needed, to perform reset or cancel on external memory, depending on the available reset mechanisms)

- FlsQspiErrorCheckCallout (used, if needed, to check any errors occurred in external memories during erase or program operations)
- FlsQspiEccCheckCallout (used, if needed, to check any errors occurred in external memories during read operations)

6.4 Tips for FLS integration

Synchronous vs. Asynchronous write mode - internal flash

Asynchronous write mode works in the way, that Fls_MainFunction() just schedules the HW write operation and does not wait for its completion. In the next Fls_MainFunction() it is checked if the write operation is finished. If yes (depends on how often the Fls_MainFunction() is called), another write operation is scheduled. This process is repeated until all data is written. In this mode, FlsMaxWriteFastMode/ FlsMaxWriteNormalMode values are ignored, data is written just by FlsPageSize length.

When synchronous write mode is used, Fls_MainFunction() initializes write operation and also waits for its completion.

So the main differences between these two modes are in the time consumption and number of calls of the Fls_MainFunction(). The Fls_MainFunction() takes less time in asynchronous mode, but the whole write operation uses more Fls_MainFunction() executions.

Example1:

Table 6-1. Example: Synchronous vs. Asynchronous write mode

Fls_MainFunction()	Asynchronous mode	Synchronous mode
Time consumption (avrg/max)	15,7/ 96,8 µs	62,3/ 169,3 µs
Calls needed (avrg/max)	13/ 680	5/377

Example2:

FLS Max Write = 16 Byte, FLS Page Size = 8 Bytes and we are going to write 32 Bytes. In asynchronous write mode it will take at least 5 Fls_MainFunctions() (4 x 8 Bytes + 1 finish job check). In synchronous write mode it will take just 2 Fls_MainFunctions() to finish the write job (2x 16 bytes).

Synchronous, Asynchronous, Interrupt mode - external flash

FLS jobs over external sectors can be configured in the following modes: READ jobs in SYNC or IRQ mode, ERASE and WRITE jobs in SYNC/ASYNC/IRQ mode. ASYNC and IRQ modes are not compatible and only one can be chosen. The SYNC and ASYNC modes for external sectors are similar to the internal sectors. The IRQ mode processes the QuadSPI job using interrupts and forces all jobs to be processed using interrupts, with some restrictions applying.

Note: The recommended operating mode for READ operations is the SYNC mode plus AHB read mode configuration setting. Setting the IRQ mode for a sector will not allow performing the reads using the AHB interface, thus forcing the use of the slower IP read interface. For ERASE/WRITE operations the time it takes for the actual hardware operation to complete inside the external memory is significantly larger than the driver processing time or the communication transfer, thus SYNC or ASYNC modes are recommended, depending on the application implementation.

External flash initialization

The QuadSPI IP is initialized according to the configuration parameters for the allocated hardware unit. The driver assumes and requires that at the end of the initialization both the QuadSPI IP and the external memory are in a state which allows all the requested operations. The configurations listed below might be required in a specific application setup and due to their specific nature of being hardware dependent on the external device, the application might be required to perform additional configurations. Because these settings are hardware dependent, implementing them in the FLS driver is not a feasible solution.

Sector protection: External memories can provide a large number of options for sector protection, locking, passwords, etc. The application should configure the protection settings for the configured external sectors as required by the subsequent jobs(ex: unlock all sectors which have to be erase or written). FlsQspiInitCallout is provided at the end of the FLS initialization for these extra checks and settings.

Special modes enablement: External memories usually provide high performance modes(DDR mode, Quad mode) for which extra configurations might be needed (ex: enable the quad bit in the configuration register in order to use 4 data lines). FlsQspiInitCallout is provided at the end of the FLS initialization for these extra checks and settings.

Data learning/calibration: High performance modes(DDR, quad output) require tight timing constraints which depending on external conditions(pressure, voltage, temperature, line length, etc) might require calibration steps. Calibration(data learning) is usually performed by using a known pattern which is read repeatedly while varying

available timing parameters. FlsQspiInitCallout is provided at the end of the FLS initialization for these extra checks and settings alongside with external APIs(ex:Fls_QspiSetCalibDelayValues) which allow setting the available internal delays.

Reset/cancel: Cancel operations or erroneous situations in which the external memory is not idle at the beginning of a new job, require either waiting for the current internal memory operation to finish or attempting to reset it. The reset mechanism is memory dependent(hardware reset, software reset, etc). FlsQspiResetCallout is provided at the beginning of each job, if the memory is not idle. If not implemented, the FLS driver will poll on the external memory status, waiting for it to become idle.

Error checks: FlsQspiErrorCheckCallout(for erase and write jobs) and FlsQspiEccCheckCallout(for read jobs) are provided at the end of each job in order to allow the application to check if any error occurred during the previous read and mark the current job as failed by returning E_NOT_OK value.

Configuration parameters

FlsProgrammingSize: For internal flash sectors, the available value is: FLS_WRITE_DOUBLE_WORD(8bytes) and it represents the maximum value the internal flash controller is able to write in one hardware operation. For external flash sectors, the available values are: FLS_WRITE_128BYTES_PAGE(128bytes), FLS_WRITE_256BYTES_PAGE(256bytes), FLS_WRITE_512BYTES_PAGE(512bytes), etc, depending on hardware availability. For the external flash sectors, this value should be correlated with the memory internal buffer size and alignment restrictions. Internally, the flash driver will use this value to check the alignment boundaries, in order to avoid the memory buffer wrap-around, otherwise if the boundary location is surpassed then data from the start address of the internal buffer will be overwritten. This parameter setting will not override or increase the MaxWrite parameters in terms of maximum programming size per MainFunction iteration, it will just check the alignment and maximum size per hardware transaction. The number of bytes sent to the memory in a transaction will be limited by the smallest intersection of the following conditions: MaxWrite configuration parameters, sector boundary, FlsProgrammingSize and the maximum size of the QuadSPI TX buffer. If the write is performed in ASYNC mode, then a single hardware transaction is performed per Fls_MainFunction iteration, otherwise in SYNC mode the entire MaxWrite size is written or until a sector boundary is reached.

FlsPageSize: Page size for external memories must be configured to a minimum value and alignment of 16 bytes. Choosing a smaller value or smaller alignment is strongly discouraged because of the internal QuadSPI IP restrictions, which require a minimum write size of 16 bytes for each transaction.

External hardware address: For external flash sectors, the external hardware address parameter represents the address of the sector inside the external memory. This value is added internally to the memory channel offset when launching QuadSPI commands and decoded by the QuadSPI IP, subtracted the offset of the chip select channel and the remainder is sent to the memory as the address parameter of the current command.

Possible values after interrupted HW write

Following value can be read from the flash when the HW write operation is interrupted:

1. Valid value (no ECC exception) and correct (the same as was intended to be written).

2. Valid value (no ECC exception) but incorrect (not the same as was intended to be written) – write operation was interrupted when ECC was not fully written. 1 bit error was improperly detected which lead to unwanted data correction. Example: we are going to write 00 C4 00 00, but the write operation is interrupted by reset. After reset we can see that the flash contains value 10 C4 00 00.

3. Always wrong value (stable ECC error).

4. Value with low margin – sometimes valid value is read (scenario 1 or 2) and sometimes ECC.

For Fls_Flash_InvalidPrefetchBuff_Ram function

The Fls_Flash_InvalidPrefetchBuff_Ram function must be placed and executed from RAM, because it accesses the flash prefetch buffers, which should not be modified while reading code instructions.

Chapter 7
Memory Allocation

7.1 Sections to be defined in MemMap.h

For Post Build data:

```
#ifdef FLS_START_SEC_CONFIG_DATA_8
#undef FLS_START_SEC_CONFIG_DATA_8
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_CONFIG_DATA_8
#undef FLS_STOP_SEC_CONFIG_DATA_8
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif



#ifdef FLS_START_SEC_CONFIG_DATA_UNSPECIFIED
#undef FLS_START_SEC_CONFIG_DATA_UNSPECIFIED
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_CONFIG_DATA_UNSPECIFIED
#undef FLS_STOP_SEC_CONFIG_DATA_UNSPECIFIED
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
```

For Code:

```
#ifdef FLS_START_SEC_CODE
#undef FLS_START_SEC_CODE
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_CODE
#undef FLS_STOP_SEC_CODE
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
```

For Variables:

```
#ifdef FLS_START_SEC_VAR_INIT_BOOLEAN
#undef FLS_START_SEC_VAR_INIT_BOOLEAN
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_VAR_INIT_BOOLEAN
#undef FLS_STOP_SEC_VAR_INIT_BOOLEAN
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif



#ifdef FLS_START_SEC_VAR_INIT_8
#undef FLS_START_SEC_VAR_INIT_8
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_VAR_INIT_8
#undef FLS_STOP_SEC_VAR_INIT_8
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif



#ifdef FLS_START_SEC_VAR_INIT_32
#undef FLS_START_SEC_VAR_INIT_32
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_VAR_INIT_32
#undef FLS_STOP_SEC_VAR_INIT_32
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif



#ifdef FLS_START_SEC_VAR_INIT_UNSPECIFIED
#undef FLS_START_SEC_VAR_INIT_UNSPECIFIED
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_VAR_INIT_UNSPECIFIED
#undef FLS_STOP_SEC_VAR_INIT_UNSPECIFIED
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
```

For Constant data:

```
#ifdef FLS_START_SEC_CONST_32
#undef FLS_START_SEC_CONST_32
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
#ifdef FLS_STOP_SEC_CONST_32
#undef FLS_STOP_SEC_CONST_32
#undef MEMMAP_ERROR
/*no definition -> default compiler settings are used */
#endif
```

```
   #ifdef FLS_START_SEC_CONST_UNSPECIFIED
   #undef FLS_START_SEC_CONST_UNSPECIFIED
   #undef MEMMAP_ERROR
   /*no definition -> default compiler settings are used */
   #endif
   #ifdef FLS_STOP_SEC_CONST_UNSPECIFIED
   #undef FLS_STOP_SEC_CONST_UNSPECIFIED
   #undef MEMMAP_ERROR
   /*no definition -> default compiler settings are used */
   #endif
```

For Ram Code:

For position-independent Access Code:

```
  #ifdef FLS_START_SEC_CODE_AC
    #undef FLS_START_SEC_CODE_AC
    #undef MEMMAP_ERROR
    /* use code relative addressing mode to ensure Position-independent Code */
    #pragma section CODE ".acfls_code_rom" far-code /* Diab example */
#endif
#ifdef FLS_STOP_SEC_CODE_AC
    #undef FLS_STOP_SEC_CODE_AC
    #undef MEMMAP_ERROR
    #pragma section CODE
#endif
```

7.2 Linker command file

Memory shall be allocated for every section defined in FLS_MemMap.h

7.3 Linker command file Access Code section

The "Fls_Flash_AccessCode" function executes the actual hardware write/erase operations.

The "Fls_Flash_AccessCode" function is placed in the driver code inside a specific linker section(".acfls_code_rom"), which can be used in the linker to specifically position this code section.

The "Fls_Flash_AccessCode" function must be executed from a different partition than the ones which contain the current written/erased sector, or it has to be executed from RAM, in order to meet the Read-While-Write restrictions. For more details about access code, see also the User Manual, "6.2 Avoiding RWW problem" chapter.

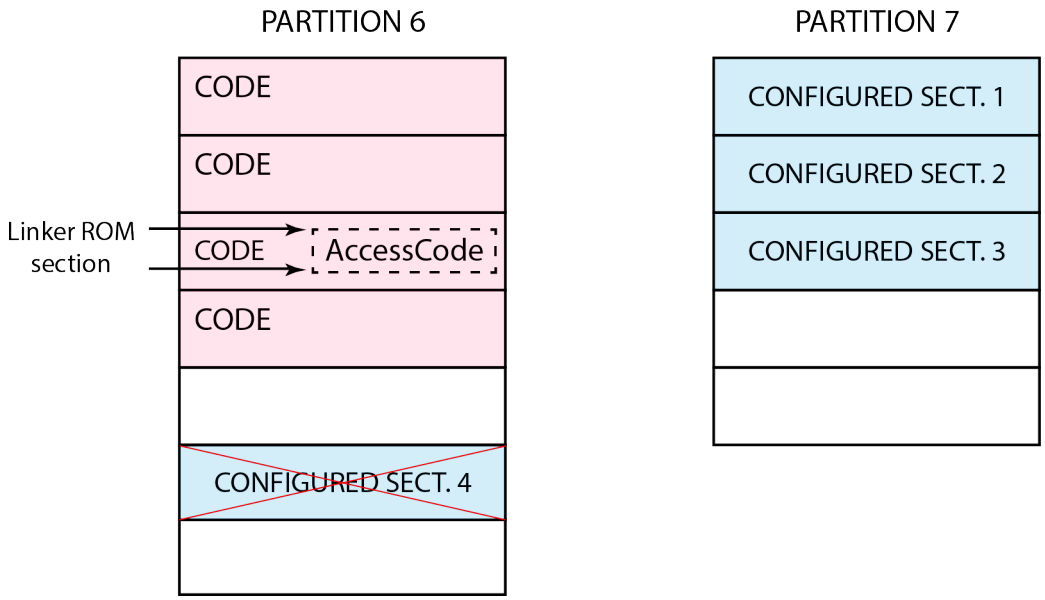

Figure 7-1. "Fls_Flash_AccessCode" function used from Flash

If "Fls_Flash_AccessCode" function is executed from Flash, configuration parameter "FlsAcLoadOnJobStart" must be cleared and the Read-While-Write restrictions apply when configuring the used Flash sectors.

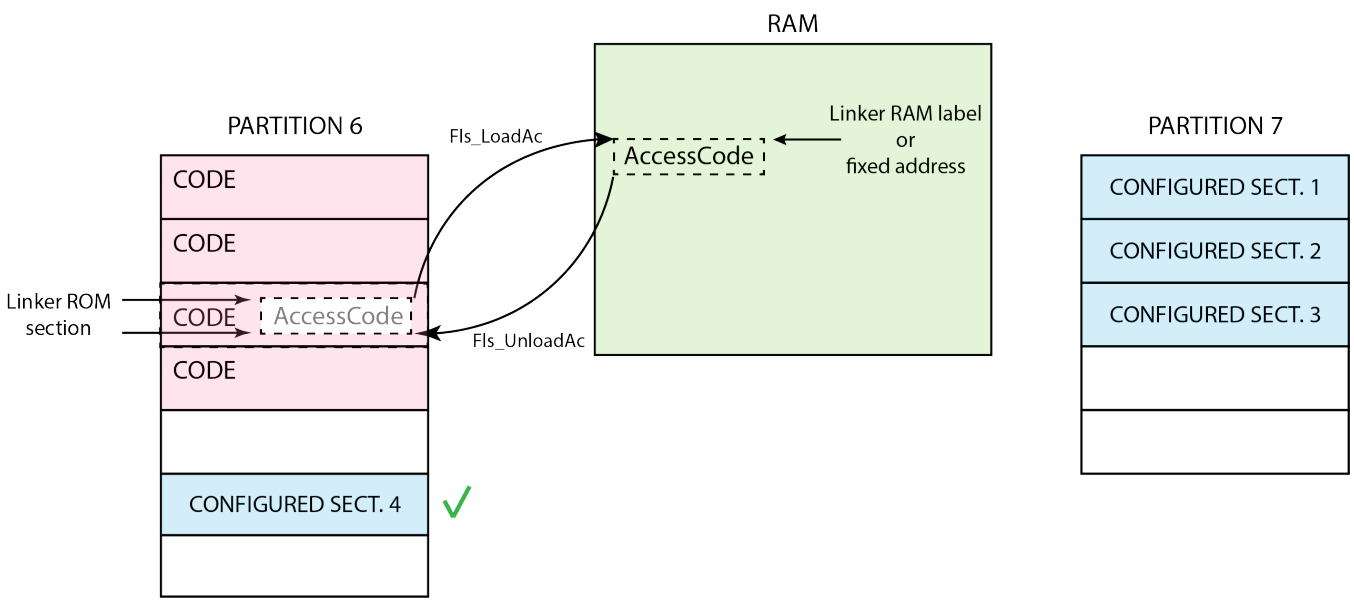

Figure 7-2. "Fls_Flash_AccessCode" function used from RAM

If executed from RAM, configuration parameter "FlsAcLoadOnJobStart" must be set and there have to be defined in the linker file the following symbols:

- Fls_ACEraseRomStart - start address of the section .acfls_code_rom
- Fls_ACEraseSize - size of .acfls_code_rom (word-aligned, in words)

- Fls_ACWriteRomStart - start address of the section .acfls_code_rom
- Fls_ACWriteSize - size of .acfls_code_rom (word-aligned, in words)

and at least 4-bytes aligned space reserved in RAM at locations defined by configuration parameters:

- FlsAcErase of space corresponding to Fls_ACEraseSize (see above)

  FlsAcWrite of space corresponding to Fls_ACWriteSize (see above)

Alternatively, using the following configuration parameters

- FlsAcErasePointer
- FlsAcWritePointer

it is possible to use symbolic name instead of absolute addresses, but in this case the linker should define them.

Note that the linker shall be prevented from .acfls_code_rom section removal, esp. when dead code stripping is enabled, e.g. by using keep directive as shown in the examples below.

DIAB linker command file example:

```
    ....

      GROUP : {
            /* ... */
            .acfls_code_rom(TEXT)  ALIGN(4) : {KEEP (*(.acfls_code_rom))}  /* see
FLS_START_SEC_CODE_AC */

            /* ... */
    }>flash_memory

      GROUP : {
            /* ... */
            .acfls_code_ram(TEXT) ALIGN(4): {}
            . += SIZEOF(.acfls_code_rom);      /* reserved RAM space */
            .acfls_code_ram_end ALIGN(4): {}   /* make sure the size of the reserved space
is 4-bytes aligned */
            /* ... */
    }>int_sram

    /* Fls module access code support */
    Fls_ACEraseRomStart         = ADDR(.acfls_code_rom);
    Fls_ACEraseSize             = (SIZEOF(.acfls_code_rom)+3) / 4; /* Copy 4 bytes at a
time*/

    Fls_ACWriteRomStart         = ADDR(.acfls_code_rom);
    Fls_ACWriteSize             = (SIZEOF(.acfls_code_rom)+3) / 4; /* Copy 4 bytes at a
time*/

    _ERASE_FUNC_ADDRESS_        = ADDR(.acfls_code_ram);  /* AC Erase Ptr */
    _WRITE_FUNC_ADDRESS_        = ADDR(.acfls_code_ram);  /* AC Write Ptr */
```

GHS linker command file example:

Linker command file Access Code section

```
//...
SECTIONS
{
//
// RAM SECTIONS
    .intc_vector                                ABS : > int_sram
    //...
    // reserve space for .acfls_code_ram
    .acfls_code_ram                         ALIGN(4) : {. += SIZEOF(.acfls_code_rom);} > .
    // make sure the size of the reserved space is 4-bytes aligned
    .acfls_code_ram_end                     ALIGN(4) : > .
//
// ROM SECTIONS
//
    //...
    .acfls_code_rom                         ALIGN(4) : > flash_memory

    /* Fls module access code support */
    Fls_ACEraseRomStart         = ADDR(.acfls_code_rom);
    Fls_ACEraseSize             = (SIZEOF(.acfls_code_rom)+3) / 4; /* Copy 4 bytes at a
time*/

    Fls_ACWriteRomStart         = ADDR(.acfls_code_rom);
    Fls_ACWriteSize             = (SIZEOF(.acfls_code_rom)+3) / 4; /* Copy 4 bytes at a
time*/

    _ERASE_FUNC_ADDRESS_        = ADDR(.acfls_code_ram);
    _WRITE_FUNC_ADDRESS_        = ADDR(.acfls_code_ram);
}

OPTION ("-keep=Fls_LLD_AccessCode")
```

Chapter 8
Configuration parameters considerations

Configuration parameter class for Autosar FLS driver fall into the following variants as defined below:

8.1 Configuration Parameters

Specifies whether the configuration parameter shall be of configuration class Post Build an PreCompile.

Table 8-1. Configuration Parameters

Configuration Container	Configuration Parameters	Configuration Variant	Current Implementation
FlsGeneral			
	FlsAcLoadOnJobStart	VariantPostBuild	PreCompile
	FlsBaseAddress	VariantPostBuild	PreCompile
	FlsCancelApi	VariantPostBuild	PreCompile
	FlsCompareApi	VariantPostBuild	PreCompile
	FlsBlankCheckApi	VariantPostBuild	PreCompile
	FlsRuntimeErrorDetect	VariantPostBuild	PreCompile
	FlsDevErrorDetect	VariantPostBuild	PreCompile
	FlsDriverIndex	VariantPostBuild	PreCompile
	FlsGetJobResultApi	VariantPostBuild	PreCompile
	FlsGetStatusApi	VariantPostBuild	PreCompile
	FlsSetModeApi	VariantPostBuild	PreCompile
	FlsTotalSize	VariantPostBuild	PreCompile
	FlsUseInterrupts	VariantPostBuild	PreCompile
	FlsVersionInfoApi	VariantPostBuild	PreCompile
	FlsDsiHandlerApi	VariantPostBuild	PreCompile
	FlsEraseBlankCheck	VariantPostBuild	PreCompile
	FlsWriteBlankCheck	VariantPostBuild	PreCompile
	FlsWriteVerifyCheck	VariantPostBuild	PreCompile

Table continues on the next page...

Table 8-1. Configuration Parameters (continued)

Configuration Container	Configuration Parameters	Configuration Variant	Current Implementation
	FlsMaxEraseBlankCheck	VariantPostBuild	PreCompile
FlsTimeouts			
	FlsAsyncWriteTimeout	VariantPostBuild	PreCompile
	FlsAsyncEraseTimeout	VariantPostBuild	PreCompile
	FlsSyncWriteTimeout	VariantPostBuild	PreCompile
	FlsSyncEraseTimeout	VariantPostBuild	PreCompile
	FlsQspiSyncReadTimeout	VariantPostBuild	PreCompile
	FlsQspiAsyncWriteTimeout	VariantPostBuild	PreCompile
	FlsQspiAsyncEraseTimeout	VariantPostBuild	PreCompile
	FlsQspiSyncWriteTimeout	VariantPostBuild	PreCompile
	FlsQspiSyncEraseTimeout	VariantPostBuild	PreCompile
	FlsQspiIrqReadTimeout	VariantPostBuild	PreCompile
	FlsQspiIrqEraseTimeout	VariantPostBuild	PreCompile
	FlsQspiIrqWriteTimeout	VariantPostBuild	PreCompile
	FlsAbortTimeout	VariantPostBuild	PreCompile
FlsConfigSet			
	FlsAcErase	VariantPostBuild	PostBuild
	FlsAcWrite	VariantPostBuild	PostBuild
	FlsAcErasePointer	VariantPostBuild	PostBuild
	FlsAcWritePointer	VariantPostBuild	PostBuild
	FlsCallCycle	VariantPostBuild	PostBuild
	FlsAcCallback	VariantPostBuild	PostBuild
	FlsJobEndNotification	VariantPostBuild	PostBuild
	FlsJobErrorNotification	VariantPostBuild	PostBuild
	FlsStartFlashAccessNotif	VariantPostBuild	PostBuild
	FlsFinishedFlashAccessNotif	VariantPostBuild	PostBuild
	FlsQspiInitCallout	VariantPostBuild	PostBuild
	FlsQspiResetCallout	VariantPostBuild	PostBuild
	FlsQspiErrorCheckCallout	VariantPostBuild	PostBuild
	FlsQspiEccCheckCallout	VariantPostBuild	PostBuild
	FlsMaxReadFastMode	VariantPostBuild	PostBuild
	FlsMaxReadNormalMode	VariantPostBuild	PostBuild
	FlsMaxWriteFastMode	VariantPostBuild	PostBuild
	FlsMaxWriteNormalMode	VariantPostBuild	PostBuild
	FlsProtection	VariantPostBuild	PostBuild
FlsSector			
	FlsSectorIndex	VariantPostBuild	PostBuild
	FlsPhysicalSectorUnlock	VariantPostBuild	PostBuild
	FlsPhysicalSector	VariantPostBuild	PostBuild

Table continues on the next page...

Table 8-1. Configuration Parameters (continued)

Configuration Container	Configuration Parameters	Configuration Variant	Current Implementation
	FlsPageSize	VariantPostBuild	PostBuild
	FlsSectorSize	VariantPostBuild	PreCompile
	FlsSectorStartaddress	VariantPostBuild	PostBuild
	FlsProgrammingSize	VariantPostBuild	PostBuild
	FlsSectorEraseAsynch	VariantPostBuild	PostBuild
	FlsPageWriteAsynch	VariantPostBuild	PostBuild
	FlsSectorIrqMode	VariantPostBuild	PostBuild
	FlsHwCh	VariantPostBuild	PostBuild
	FlsQspiSectorCh	VariantPostBuild	PostBuild
	FlsSectorHwAddress	VariantPostBuild	PostBuild
FlsHwUnit			
	FlsHwUnitName	VariantPostBuild	PostBuild
	FlsHwUnitReadMode	VariantPostBuild	PostBuild
	FlsHwUnitDqsMode	VariantPostBuild	PostBuild
	FlsHwUnitInputPadsBufferEn	VariantPostBuild	PostBuild
	FlsHwUnitInputClockSel	VariantPostBuild	PostBuild
	FlsHwUnitInternalRefClockSel	VariantPostBuild	PostBuild
	FlsHwUnitDqsBStage2ClkSource	VariantPostBuild	PostBuild
	FlsHwUnitDqsBStage1ClkSource	VariantPostBuild	PostBuild
	FlsHwUnitDqsAStage2ClkSource	VariantPostBuild	PostBuild
	FlsHwUnitDqsAStage1ClkSource	VariantPostBuild	PostBuild
	FlsHwUnitPendingReadBusGasket	VariantPostBuild	PostBuild
	FlsHwUnitBurstReadBusGasket	VariantPostBuild	PostBuild
	FlsHwUnitBurstWriteBusGasket	VariantPostBuild	PostBuild
	FlsHwUnitProgrammableDivider	VariantPostBuild	PostBuild
	IdleSignalDriveIOFB3HighLvl	VariantPostBuild	PostBuild
	IdleSignalDriveIOFB2HighLvl	VariantPostBuild	PostBuild
	IdleSignalDriveIOFA3HighLvl	VariantPostBuild	PostBuild
	IdleSignalDriveIOFA2HighLvl	VariantPostBuild	PostBuild
	FlsHwUnitSamplingEdge	VariantPostBuild	PostBuild
	FlsHwUnitSamplingDly	VariantPostBuild	PostBuild
	FlsHwUnitSamplingPoint	VariantPostBuild	PostBuild
	FlsHwUnitDqsLatencyEnable	VariantPostBuild	PostBuild
	FlsHwUnitFineDelayA	VariantPostBuild	PostBuild
	FlsHwUnitFineDelayB	VariantPostBuild	PostBuild
	FlsHwUnitTdh	VariantPostBuild	PostBuild
	FlsHwUnitTcsh	VariantPostBuild	PostBuild
	FlsHwUnitTcss	VariantPostBuild	PostBuild
	FlsHwUnitEndianness	VariantPostBuild	PostBuild

Table continues on the next page...

Table 8-1. Configuration Parameters (continued)

Configuration Container	Configuration Parameters	Configuration Variant	Current Implementation
	FlsHwUnitReadBufferMode	VariantPostBuild	PostBuild
	FlsHwUnitRxBufferAccessMode	VariantPostBuild	PostBuild
	FlsHwUnitColumnAddressWidth	VariantPostBuild	PostBuild
	FlsHwUnitWordAddressable	VariantPostBuild	PostBuild
	FlsSerialFlashA2TopAddr	VariantPostBuild	PostBuild
	FlsSerialFlashB1TopAddr	VariantPostBuild	PostBuild
	FlsSerialFlashB2TopAddr	VariantPostBuild	PostBuild
	FlsQspiHyperflashLatencyCycles	VariantPostBuild	PostBuild
	FlsExtUnitRegWidth	VariantPostBuild	PostBuild
	FlsBusyBitValue	VariantPostBuild	PostBuild
	FlsBusyBitPositionOffset	VariantPostBuild	PostBuild
	FlsQspiDeviceId	VariantPostBuild	PostBuild
FlsAhbBuffer			
	FlsAhbBufferInstance	VariantPostBuild	PostBuild
	FlsAhbBufferMasterId	VariantPostBuild	PostBuild
	FlsAhbBufferSize	VariantPostBuild	PostBuild
	FlsAhbBufferDataTransferSize	VariantPostBuild	PostBuild
	FlsAhbBufferAllMasters	VariantPostBuild	PreCompile
	FlsAhbBufferHighPriority	VariantPostBuild	PostBuild
FlsLUT			
	FlsLUTIndex	VariantPostBuild	PostBuild

Chapter 9
Integration Steps

This section gives a brief overview of the steps needed for integrating Flash :

- Generate the required FLS configurations. For more details refer to section Files required for Compilation
- Allocate proper memory sections in FLS_MemMap.h and linker command file. For more details refer to section Sections to be defined in MemMap.h
- Compile & build the FLS with all the dependent modules. For more details refer to section Building the Driver

Chapter 10
ISR Reference

None

Chapter 11
External Assumptions for FLS driver

The section presents requirements that must be complied with when integrating FLS driver into the application.

[SMCAL_CPR_EXT165]

<< The option "Fls Hardware Timeout Handling" shall be enabled

in the FLS driver configuration and the timeout value shall be configured to fit the

application timing. >>

[SMCAL_CPR_EXT175]

<< The integrator shall assure the execution of code from system RAM when flash memory configurations need to be change (i.e. PFCR control fields of PFLASH memory need to be change) >>

[SMCAL_CPR_EXT181]

<< The integrator shall make sure that the Memory stack operations (Flash operations) are not executed concurrently with Crypto Stack operations (CSEC operations). >>

[SMCAL_CPR_EXT187]

<< When multicore feature is activated, MainFunction calls shall be scheduled for erase jobs with a period greater than the one described in the reference manual as a hardware limitation for the erase suspend-resume sequence. (e.g. on S32S2XX, this minimum time is defined in the reference manual as 5ms. >>

[SMCAL_CPR_EXT188]

<< When both multicore and timeout features are activated, a large enough timeout value shall be used for jobs because of the HIPRIO semaphore shared usage that can cause job starvation on a core >>

[SMCAL_CPR_EXT190]

<< The Init job resets the driver's state(Example: clear SEMA4s, abort HW jobs, etc). The application shall ensure that, in multicore context, both instances of the driver are initialized before any other job is scheduled, driver is idle, or assume that the current running job may be aborted. >>

[SWS_Fls_00214]

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

NOTE

There is a global configuration parameter to select if access codes are loaded into RAM or not.

[SWS_Fls_00240]

<< The FLS module's environment shall only call the function Fls_Read after the FLS module has been initialized. >>

NOTE

Out of scope sMcal

[SWS_Fls_00038]

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

NOTE

Out of scope sMcal

[SWS_Fls_00260]

<< API function Description

Dem_ReportErrorStatus Queues the reported events from the BSW modules (API is only used by BSW modules). The interface has an asynchronous behavior, because the processing of the event is done within the Dem main function.

OBD Events Suppression shall be ignored for this computation. >>

NOTE

Out of scope sMcal

[SWS_Fls_00261]

<< API function Description

Det_ReportError Service to report development errors. >>

NOTE

Out of scope sMcal

[SWS_Fls_00262]

<< Service name: Fee_JobEndNotification

Syntax: void Fee_JobEndNotification(

void

)

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. >>

NOTE

Configurable interface (Fee_JobEndNotification() callback).

[SWS_Fls_00263]

<< Service name: Fee_JobErrorNotification

Syntax: void Fee_JobErrorNotification(

void

)

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 canceled or finished with negative result. >>

NOTE

Configurable interface (Fee_JobErrorNotification() callback).

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

Document Number IM2FLSASR4.2 Rev0002R1.0.2
Revision 1.0

AUTOSAR