

Integration Manual

for S32K1 FLS Driver

Document Number: IM2FLSASR4.4 Rev0000R1.0.1 Rev. 1.0

1 Revision History	2
2 Introduction	3
2.1 Supported Derivatives	3
2.2 Overview	4
2.3 About This Manual	4
2.4 Acronyms and Definitions	5
2.5 Reference List	5
3 Building the driver	6
3.1 Build Options	6
3.1.1 GCC Compiler/Assembler/Linker Options	6
3.1.2 GHS Compiler/Assembler/Linker Options	9
3.1.3 IAR Compiler/Assembler/Linker Options	12
3.2 Files required for compilation	14
3.2.1 Fls Files	14
3.2.2 Other includes files	15
3.3 Setting up the plugins	16
3.3.1 Location of various files inside the FLS module folder	16
3.3.2 Steps to generate the configuration	16
4 Function calls to module	17
4.1 Function Calls during Start-up	17
4.2 Function Calls during Shutdown	18
4.3 Function Calls during Wake-up	19
5 Module requirements	20
5.1 Exclusive areas to be defined in BSW scheduler	20
5.1.1 Critical Region Exclusive Matrix	22
5.2 Exclusive areas not available on this platform	22
5.3 Peripheral Hardware Requirements	22
5.4 ISR to configure within AutosarOS - dependencies	23
5.5 ISR Macro	23
5.5.1 Without an Operating System	23
5.5.2 With an Operating System	24
5.6 Other AUTOSAR modules - dependencies	24
5.7 Data Cache Restrictions	24
5.8 User Mode support	25
5.8.1 User Mode configuration in the module	25
5.8.2 User Mode configuration in AutosarOS	26
6 Main API Requirements	27

6.1 Main function calls within BSW scheduler	27
6.2 API Requirements	27
6.3 Calls to Notification Functions, Callbacks, Callouts	27
6.3.1 Internal flash access notifications	28
7 Memory allocation	29
7.1 Sections to be defined in Fls_MemMap.h	29
7.2 Linker command file	32
8 Integration Steps	33
9 External assumptions for driver	34
9.1 Application's tasks	35
9.2 External memory	36
10 Tips for FLS integration	37
10.1 Synchronous vs. Asynchronous mode	37
10.2 Speculative prefetch buffer of internal flash	37
10.3 Flash Configuration Field	38
10.3.1 Sector protection	38
10.3.2 Chip security	39
10.4 ECC Management	39
10.4.1 ECC Management on Internal Flash	39
10.4.2 ECC Management on Qspi Flash	43



Chapter 1

Revision History

Revision	Date	Author	Description
1.0	24.02.2022	NXP RTD Team	Prepared for release RTD S32K1 Version 1.0.1

Chapter 2

Introduction

- [Supported Derivatives](#)
- [Overview](#)
- [About This Manual](#)
- [Acronyms and Definitions](#)
- [Reference List](#)

This Integration Manual describes the integration requirements for FLS Driver for S32K1 microcontrollers.

2.1 Supported Derivatives

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

- s32k116_qfn32
- s32k116_lqfp48
- s32k118_lqfp48
- s32k118_lqfp64
- s32k142_lqfp48
- s32k142_lqfp64
- s32k142_lqfp100
- s32k142w_lqfp48
- s32k142w_lqfp64
- s32k144_lqfp48
- s32k144_lqfp64
- s32k144_lqfp100

- s32k144_mapbga100
- s32k144w_lqfp48
- s32k144w_lqfp64
- s32k146_lqfp64
- s32k146_lqfp100
- s32k146_mapbga100
- s32k146_lqfp144
- s32k148_lqfp100
- s32k148_mapbga100
- s32k148_lqfp144
- s32k148_lqfp176

All of the above microcontroller devices are collectively named as S32K1.

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** style: Used for important terms, notes and warnings.
- *Italic* style: 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.

Warning

This is a warning

2.4 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 Available
MCU	Microcontroller Unit
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
FEE	Flash EEPROM Emulation
FLS	Flash
RTD	Real Time Drivers
XML	Extensible Markup Language

2.5 Reference List

#	Title	Version
1	Specification of Fls Driver	AUTOSAR Release 4.4.0
2	Reference Manual	S32K1xx Series Reference Manual, Rev. 14, 09/2021
3	Datasheet	S32K1xx Data Sheet, Rev. 14, 08/2021
4	Errata	S32K116_0N96V Rev. 22/OCT/2021
		S32K118_0N97V Rev. 22/OCT/2021
		S32K142_0N33V Rev. 22/OCT/2021
		S32K144_0N57U Rev. 22/OCT/2021
		S32K144W_0P64A Rev. 22/OCT/2021
		S32K146_0N73V Rev. 22/OCT/2021
		S32K148_0N20V Rev. 22/OCT/2021

Chapter 3

Building the driver

- [Build Options](#)
- [Files required for compilation](#)
- [Setting up the plugins](#)

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

3.1 Build Options

- [GCC Compiler/Assembler/Linker Options](#)
- [GHS Compiler/Assembler/Linker Options](#)
- [IAR Compiler/Assembler/Linker Options](#)

The RTD driver files are compiled using:

- NXP GCC 9.2.0 20190812 (Build 1649 Revision gaf57174)
- IAR ANSI C/C++ Compiler V8.40.3.228/W32 for ARM Functional Safety
- Green Hills Multi 7.1.6d / Compiler 2020.1.4

The compiler, assembler, and linker flags used for building the driver are explained below.

The TS_T40D2M10I1R0 part of the plugin name is composed as follows:

- T = Target_Id (e.g. T40 identifies Cortex-M architecture)
- D = Derivative_Id (e.g. D2 identifies S32K1 platform)
- M = SW_Version_Major and SW_Version_Minor
- I = SW_Version_Patch
- R = Reserved

3.1.1 GCC Compiler/Assembler/Linker Options

3.1.1.1 GCC Compiler Options

Compiler Option	Description
-mcpu=cortex-m4	Targeted ARM processor for which GCC should tune the performance of the code (for S32K14x devices)
-mcpu=cortex-m0plus	Targeted ARM processor for which GCC should tune the performance of the code (for S32K11x devices)
-mthumb	Generates code that executes in Thumb state
-mlittle-endian	Generate code for a processor running in little-endian mode
-mfpv4-sp-d16	Specifies the floating-point hardware available on the target (for S32K14x devices)
-mfloat-abi=hard	Specifies the floating-point ABI to use. "hard" allows generation of floating-point instructions and uses FPU-specific calling conventions (for S32K14x devices)
-mfpv4-sp-d16	Specifies the floating-point hardware available on the target (for S32K11x devices)
-mfloat-abi=soft	Specifies the floating-point ABI to use. Specifying "soft" causes GCC to generate output containing library calls for floating-point operations (for S32K11x devices)
-std=c99	Specifies the ISO C99 base standard
-Os	Optimize for size. Enables all -O2 optimizations except those that often increase code size
-ggdb3	Produce debugging information for use by GDB using the most expressive format available, including GDB extensions if at all possible. Level 3 includes extra information, such as all the macro definitions present in the program
-Wall	Enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros
-Wextra	This enables some extra warning flags that are not enabled by -Wall
-pedantic	Issue all the warnings demanded by strict ISO C. Reject all programs that use forbidden extensions. Follows the version of the ISO C standard specified by the aforementioned -std option
-Wstrict-prototypes	Warn if a function is declared or defined without specifying the argument types
-Wundef	Warn if an undefined identifier is evaluated in an #if directive. Such identifiers are replaced with zero
-Wunused	Warn whenever a function, variable, label, value, macro is unused
-Werror=implicit-function-declaration	Make the specified warning into an error. This option throws an error when a function is used before being declared
-Wsign-compare	Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.
-Wdouble-promotion	Give a warning when a value of type float is implicitly promoted to double
-fno-short-enums	Specifies that the size of an enumeration type is at least 32 bits regardless of the size of the enumerator values.

Compiler Option	Description
-funsigned-char	Let the type char be unsigned by default, when the declaration does not use either signed or unsigned
-funsigned-bitfields	Let a bit-field be unsigned by default, when the declaration does not use either signed or unsigned
-fomit-frame-pointer	Omit the frame pointer in functions that don't need one. This avoids the instructions to save, set up and restore the frame pointer; on many targets it also makes an extra register available.
-fno-common	Makes the compiler place uninitialized global variables in the BSS section of the object file. This inhibits the merging of tentative definitions by the linker so you get a multiple-definition error if the same variable is accidentally defined in more than one compilation unit
-fstack-usage	Makes the compiler output stack usage information for the program, on a per-function basis
-fdump-ipa-all	Enables all inter-procedural analysis dumps
-c	Stop after assembly and produce an object file for each source file
-DS32K1XX	Predefine S32K1XX as a macro, with definition 1
-DS32K148	Predefine S32K148 as a macro, with definition 1
-DGCC	Predefine GCC as a macro, with definition 1
-DUSE_SW_VECTOR_MODE	Predefine USE_SW_VECTOR_MODE as a macro, with definition 1. By default, the drivers are compiled to handle interrupts in Software Vector Mode
-DI_CACHE_ENABLE	Predefine I_CACHE_ENABLE as a macro, with definition 1. Enables instruction cache initialization in source file system.c under the Platform driver (for S32K14x devices)
-DENABLE_FPU	Predefine ENABLE_FPU as a macro, with definition 1. Enables FPU initialization in source file system.c under the Platform driver (for S32K14x devices)
-DMCAL_ENABLE_USER_MODE_SUPPORT	Predefine MCAL_ENABLE_USER_MODE_SUPPORT as a macro, with definition 1. Allows drivers to be configured in user mode.

3.1.1.2 GCC Assembler Options

Assembler Option	Description
-xassembler-with-cpp	Specifies the language for the following input files (rather than letting the compiler choose a default based on the file name suffix)
-mcpu=cortex-m4	Targeted ARM processor for which GCC should tune the performance of the code (for S32K14x devices)
-mcpu=cortex-m0plus	Targeted ARM processor for which GCC should tune the performance of the code (for S32K11x devices)
-mthumb	Generates code that executes in Thumb state
-c	Stop after assembly and produce an object file for each source file

3.1.1.3 GCC Linker Options

Linker Option	Description
-Wl,-Map,filename	Produces a map file
-T linkerfile	Use linkerfile as the linker script. This script replaces the default linker script (rather than adding to it)
-entry=Reset_Handler	Specifies that the program entry point is Reset_Handler
-nostartfiles	Do not use the standard system startup files when linking
-mcpu=cortex-m4	Targeted ARM processor for which GCC should tune the performance of the code (for S32K14x devices)
-mcpu=cortex-m0plus	Targeted ARM processor for which GCC should tune the performance of the code (for S32K11x devices)
-mthumb	Generates code that executes in Thumb state
-mfpu=fpv4-sp-d16	Specifies the floating-point hardware available on the target (for S32K14x devices)
-mfloat-abi=hard	Specifies the floating-point ABI to use. "hard" allows generation of floating-point instructions and uses FPU-specific calling conventions (for S32K14x devices)
-mfpu=auto	Specifies the floating-point hardware available on the target (for S32K11x devices)
-mfloat-abi=soft	Specifies the floating-point ABI to use. Specifying "soft" causes GCC to generate output containing library calls for floating-point operations (for S32K11x devices)
-mlittle-endian	Generate code for a processor running in little-endian mode
-ggdb3	Produce debugging information for use by GDB using the most expressive format available, including GDB extensions if at all possible. Level 3 includes extra information, such as all the macro definitions present in the program
-lc	Link with the C library
-lm	Link with the Math library
-lgcc	Link with the GCC library
-n	Turn off page alignment of sections, and disable linking against shared libraries

3.1.2 GHS Compiler/Assembler/Linker Options

3.1.2.1 GHS Compiler Options

Compiler Option	Description
-cpu=cortexm4	Selects target processor: Arm Cortex M4 (for S32K14x devices)
-cpu=cortexm0plus	Selects target processor: Arm Cortex M0+ (for S32K11x devices)
-thumb	Selects generating code that executes in Thumb state
-fpu=vfpv4_d16	Specifies hardware floating-point using the v4 version of the VFP instruction set, with 16 double-precision floating-point registers (for S32K14x devices)
-fsingle	Use hardware single-precision, software double-precision FP instructions (for S32K14x devices)

Compiler Option	Description
-fsoft	Specifies software floating-point (SFP) mode. This setting causes your target to use integer registers to hold floating-point data and use library subroutine calls to emulate floating-point operations (for S32K11x devices)
-C99	Use (strict ISO) C99 standard (without extensions)
-ghstd=last	Use the most recent version of Green Hills Standard mode (which enables warnings and errors that enforce a stricter coding standard than regular C and C++)
-Osize	Optimize for size
-gnu_asm	Enables GNU extended asm syntax support
-dual_debug	Generate DWARF 2.0 debug information
-G	Generate debug information
-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
-Wimplicit-int	Produce warnings if functions are assumed to return int
-Wshadow	Produce warnings if variables are shadowed
-Wtrigraphs	Produce warnings if trigraphs are detected
-Wundef	Produce a warning if undefined identifiers are used in #if preprocessor statements
-unsigned_chars	Let the type char be unsigned, like unsigned char
-unsigned_fields	Bitfields declared with an integer type are unsigned
-no_commons	Allocates uninitialized global variables to a section and initializes them to zero at program startup
-no_exceptions	Disables C++ support for exception handling
-no_slash_comment	C++ style // comments are not accepted and generate errors
-prototype_errors	Controls the treatment of functions referenced or called when no prototype has been provided
-incorrect_pragma_warnings	Controls the treatment of valid #pragma directives that use the wrong syntax
-c	Stop after assembly and produce an object file for each source file
-DS32K1XX	Predefine S32K1XX as a macro, with definition 1
-DS32K148	Predefine S32K148 as a macro, with definition 1
-DGHS	Predefine GHS as a macro, with definition 1
-DUSE_SW_VECTOR_MODE	Predefine USE_SW_VECTOR_MODE as a macro, with definition 1. By default, the drivers are compiled to handle interrupts in Software Vector Mode
-DI_CACHE_ENABLE	Predefine I_CACHE_ENABLE as a macro, with definition 1. Enables instruction cache initialization in source file system.c under the Platform driver (for S32K14x devices)
-DENABLE_FPU	Predefine ENABLE_FPU as a macro, with definition 1. Enables FPU initialization in source file system.c under the Platform driver (for S32K14x devices)

Compiler Option	Description
-DMCAL_ENABLE_USER_MODE_SUPPORT	Predefine MCAL_ENABLE_USER_MODE_SUPPORT as a macro, with definition 1. Allows drivers to be configured in user mode

3.1.2.2 GHS Assembler Options

Assembler Option	Description
-cpu=cortexm4	Selects target processor: Arm Cortex M4 (for S32K14x devices)
-cpu=cortexm0plus	Selects target processor: Arm Cortex M0+ (for S32K11x devices)
-preprocess_assembly_files	Controls whether assembly files with standard extensions such as .s and .asm are preprocessed
-list	Creates a listing by using the name and directory of the object file with the .lst extension
-c	Stop after assembly and produce an object file for each source file

3.1.2.3 GHS Linker Options

Linker Option	Description
-e Reset_Handler	Make the symbol Reset_Handler be treated as a root symbol and the start label of the application
-T linker_script_file.ld	Use linker_script_file.ld as the linker script. This script replaces the default linker script (rather than adding to it)
-map	Produce a map file
-keepmap	Controls the retention of the map file in the event of a link error
-Mn	Generates a listing of symbols sorted alphabetically/numerically by address
-delete	Instructs the linker to remove functions that are not referenced in the final executable. The linker iterates to find functions that do not have relocations pointing to them and eliminates them
-ignore_debug_references	Ignores relocations from DWARF debug sections when using -delete. DWARF debug information will contain references to deleted functions that may break some third-party debuggers
-Llibrary_path	Points to library_path (the libraries location) for thumb2 to be used for linking
-larch	Link architecture specific library
-lstartup	Link run-time environment startup routines. The source code for the modules in this library is provided in the src/libstartup directory
-lind_sd	Link language-independent library, containing support routines for features such as software floating point, run-time error checking, C99 complex numbers, and some general purpose routines of the ANSI C library (for S32K14x devices)
-lind_sf	Link language-independent library, containing support routines for features such as software floating point, run-time error checking, C99 complex numbers, and some general purpose routines of the ANSI C library (for S32K11x devices)
-v	Prints verbose information about the activities of the linker, including the libraries it searches to resolve undefined symbols
-keep=C40_Ip_AccessCode	Avoid linker remove function C40_Ip_AccessCode from FLS module because it is not referenced explicitly

Linker Option	Description
-nostartfiles	Controls the start files to be linked into the executable

3.1.3 IAR Compiler/Assembler/Linker Options

3.1.3.1 IAR Compiler Options

Compiler Option	Description
-cpu=Cortex-M4	Targeted ARM processor for which IAR should tune the performance of the code (for S32K14x devices)
-cpu=Cortex-M0+	Targeted ARM processor for which IAR should tune the performance of the code (for S32K11x devices)
-cpu_mode=thumb	Generates code that executes in Thumb state
-endian=little	Generate code for a processor running in little-endian mode
-fpu=FPv4-SP	Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). Single-precision variant. (for S32K14x devices)
-fpu=none	Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). No FPU. (for S32K11x devices)
-e	Enables all IAR C language extensions
-Ohz	Optimize for size. the compiler will emit AEABI attributes indicating the requested optimization goal. This information can be used by the linker to select smaller or faster variants of DLIB library functions
-debug	Makes the compiler include debugging information in the object modules. Including debug information will make the object files larger
-no_clustering	Disables static clustering optimizations. Static and global variables defined within the same module will not be arranged so that variables that are accessed in the same function are close to each other
-no_mem_idioms	Makes the compiler not optimize certain memory access patterns
-no_explicit_zero_opt	Do not treat explicit initializations to zero of static variables as zero initializations
-require_prototypes	Force the compiler to verify that all functions have proper prototypes. Generates an error otherwise
-no_wrap_diagnostics	Does not wrap long lines in diagnostic messages
-diag_suppress=Pa050	Suppresses diagnostic message Pa050
-DS32K1XX	Predefine S32K1XX as a macro, with definition 1
-DS32K148	Predefine S32K148 as a macro, with definition 1
-DIAR	Predefine IAR as a macro, with definition 1
-DUSE_SW_VECTOR_MODE	Predefine USE_SW_VECTOR_MODE as a macro, with definition 1. By default, the drivers are compiled to handle interrupts in Software Vector Mode.

Compiler Option	Description
-DI_CACHE_ENABLE	Predefine I_CACHE_ENABLE as a macro, with definition 1. Enables instruction cache initialization in source file system.c under the Platform driver (for S32K14x devices)
-DENABLE_FPU	Predefine ENABLE_FPU as a macro, with definition 1. Enables FPU initialization in source file system.c under the Platform driver (for S32K14x devices)
-DMCAL_ENABLE_USER_MODE_SUPPORT	Predefine MCAL_ENABLE_USER_MODE_SUPPORT as a macro, with definition 1. Allows drivers to be configured in user mode.

3.1.3.2 IAR Assembler Options

Assembler Option	Description
-cpu=Cortex-M4	Targeted ARM processor for which IAR should tune the performance of the code (for S32K14x devices)
-cpu=Cortex-M0+	Targeted ARM processor for which IAR should tune the performance of the code (for S32K11x devices)
-cpu_mode thumb	Selects the thumb mode for the assembler directive CODE
-g	Disables the automatic search for system include files
-r	Generates debug information

3.1.3.3 IAR Linker Options

Linker Option	Description
-map filename	Produces a map file
-config linkerfile	Use linkerfile as the linker script. This script replaces the default linker script (rather than adding to it)
-cpu=Cortex-M4	Targeted ARM processor for which IAR should tune the performance of the code (for S32K14x devices)
-cpu=Cortex-M0+	Targeted ARM processor for which IAR should tune the performance of the code (for S32K11x devices)
-fpu=FPv4-SP	Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). Single-precision variant. (for S32K14x devices)
-fpu=none	Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). No FPU. (for S32K11x devices)
-entry _start	Treats _start as a root symbol and start label
-enable_stack_usage	Enables stack usage analysis. If a linker map file is produced, a stack usage chapter is included in the map file
-skip_dynamic_initialization	Dynamic initialization (typically initialization of C++ objects with static storage duration) will not be performed automatically during application startup
-no_wrap_diagnostics	Does not wrap long lines in diagnostic messages

3.2 Files required for compilation

- This section describes the include files required to compile, assemble and link the AUTOSAR Flash Driver for S32K1 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.

3.2.1 Fls Files

- Include file:
 - Fls_TS_T40D2M10I1R0\include\Fls.h
 - Fls_TS_T40D2M10I1R0\include\Fls_Api.h
 - Fls_TS_T40D2M10I1R0\include\Fls_Types.h
 - Fls_TS_T40D2M10I1R0\include\Fls_IPW.h
 - Fls_TS_T40D2M10I1R0\include\Ftfc_Fls_Ip_Ac.h
 - Fls_TS_T40D2M10I1R0\include\Ftfc_Fls_Ip_Types.h
 - Fls_TS_T40D2M10I1R0\include\Ftfc_Fls_Ip.h
 - Fls_TS_T40D2M10I1R0\include\Qspi_IP_Types.h
 - Fls_TS_T40D2M10I1R0\include\Qspi_IP.h
 - Fls_TS_T40D2M10I1R0\include\Qspi_Ip_Common.h
 - Fls_TS_T40D2M10I1R0\include\Qspi_Ip_Controller.h
 - Fls_TS_T40D2M10I1R0\include\Qspi_Ip_HwAccess.h
- Source files:
 - Fls_TS_T40D2M10I1R0\src\Fls.c
 - Fls_TS_T40D2M10I1R0\src\Fls_IPW.c
 - Fls_TS_T40D2M10I1R0\src\Ftfc_Fls_Ip_Ac.c
 - Fls_TS_T40D2M10I1R0\src\Ftfc_Fls_Ip.c
 - Fls_TS_T40D2M10I1R0\src\Qspi_Ip_Controller.c
 - Fls_TS_T40D2M10I1R0\src\Qspi_Ip_Sfdp.c
 - Fls_TS_T40D2M10I1R0\src\Qspi_Ip.c

Note

These files should be generated by the user using a configuration/generation tool

- Fls_Cfg.h
- Fls_Cfg.c
- Fls_PBcfg.h
- Fls_PBcfg.c
- Ftfc_Fls_Ip_Cfg.h
- Ftfc_Fls_Ip_PBcfg.c
- Qspi_Ip_Features.h
- Qspi_Ip_Cfg.h
- Qspi_Ip_PBcfg.h
- Qspi_Ip_PBcfg.c

3.2.2 Other includes files

3.2.2.1 Files from MemIf folder

- MemIf_TS_T40D2M10I1R0\include\MemIf_Types.h

3.2.2.2 Files from Base common folder

- Base_TS_T40D2M10I1R0\include\Compiler.h
- Base_TS_T40D2M10I1R0\include\Compiler_Cfg.h
- Base_TS_T40D2M10I1R0\include\ComStack_Types.h
- Base_TS_T40D2M10I1R0\include\Fls_MemMap.h
- Base_TS_T40D2M10I1R0\include\Mcal.h
- Base_TS_T40D2M10I1R0\include\Platform_Types.h
- Base_TS_T40D2M10I1R0\include\Std_Types.h
- Base_TS_T40D2M10I1R0\include\Reg_eSys.h
- Base_TS_T40D2M10I1R0\include\Soc_Ips.h
- Base_TS_T40D2M10I1R0\include\Reg_Macros.h
- Base_TS_T40D2M10I1R0\include\SilRegMacros.h
- Base_TS_T40D2M10I1R0\header\S32K148.h

3.2.2.3 Files from Det folder

- Det_TS_T40D2M10I1R0\include\Det.h

3.2.2.4 Files from Platform folder:

- Platform_TS_T40D2M10I1R0\include\Platform.h

3.2.2.5 Files from Rte folder

- Rte_TS_T40D2M10I1R0\include\SchM_Fls.h

3.2.2.6 Files from Os folder

- Os_TS_T40D2M10I1R0\include\Os.h

3.3 Setting up the plugins

The Flash Driver was designed to be configured by using the EB Tresos Studio (version 27.1.0 b200625-0900 or later)

3.3.1 Location of various files inside the FLS module folder

- VSMD (Vendor Specific Module Definition) file in EB Tresos Studio XDM format:
 - Fls_TS_T40D2M10I1R0\config\Fls.xdm
- VSMD (Vendor Specific Module Definition) file(s) in AUTOSAR compliant EPD format:
 - Fls_TS_T40D2M10I1R0\autosar\Fls_<subderivative_name>.epd
- Code Generation Templates for variant aware parameters:
 - Fls_TS_T40D2M10I1R0\generate_PB\include\Fls_PBcfg.h
 - Fls_TS_T40D2M10I1R0\generate_PB\src\Fls_PBcfg.c
 - Fls_TS_T40D2M10I1R0\generate_PB\src\Ftfc_Fls_Ip_PBcfg.c
 - Fls_TS_T40D2M10I1R0\generate_PB\src\Qspi_Ip_PBcfg.c
 - Fls_TS_T40D2M10I1R0\generate_PB\include\Qspi_Ip_PBcfg.h
- Code Generation Templates for parameters without variation points:
 - Fls_TS_T40D2M10I1R0\generate_PC\include\Fls_Cfg.h
 - Fls_TS_T40D2M10I1R0\generate_PC\include\Ftfc_Fls_Ip_Cfg.h
 - Fls_TS_T40D2M10I1R0\generate_PC\include\Qspi_Ip_Cfg.h
 - Fls_TS_T40D2M10I1R0\generate_PC\include\Qspi_Ip_Features.h
 - Fls_TS_T40D2M10I1R0\generate_PC\src\Fls_Cfg.c

3.3.2 Steps to generate the configuration

1. Copy the module folders:
 - Base_TS_T40D2M10I1R0
 - Det_TS_T40D2M10I1R0
 - EcuC_TS_T40D2M10I1R0
 - Fls_TS_T40D2M10I1R0
 - MemIf_TS_T40D2M10I1R0
 - Os_TS_T40D2M10I1R0
 - Platform_TS_T40D2M10I1R0
 - Resource_TS_T40D2M10I1R0
 - Rte_TS_T40D2M10I1R0 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

- [Function Calls during Start-up](#)
- [Function Calls during Shutdown](#)
- [Function Calls during Wake-up](#)

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:

- The driver does not touch the FlexNVM block partition. Users need to configure the depart appropriately in order to use the Data flash area.
- 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. There are two options to aid the configuration of the external memory
- Option 1: Configure the protection settings value in the 'Status register settings' in 'MemCfg' array. This configuration will be written into the status register of the external flash device at the end of the initialization.
- Option 2: 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.

Option 1: Configuring the protection bit field in Status register settings

MemCfg

Name MemCfg_0

Fls External Flash **initConfiguration** FlsLUT

Name statusConfig

Size in bytes of status register (1 -> 4)	1
Position of busy bit (0 -> 31)	0
busy bit active value (0 -> 1)	1
Position of Write Enable bit (0 -> 31)	1
Offset of block protection bits (0 -> 31)	2
Width of block protection bitfield (0 -> 32)	4
Value of block protection bitfield (0 -> 15)	0

- Option 2: Using FlsQspiInitCallout

Fls

Name Fls

General **ControllerCfg** MemCfg FlsController FlsMem FlsSector Published Information

Fls Start Flash Access Notification	Fls_StartFlashAccessNotif
Fls Finished Flash Access Notifications	Fls_FinishedFlashAccessNotif
Fls Read Function Callout	FlsReadFunctionCallout
Fls Qspi Init Callout	FlsQspiInitCallout0
Fls Qspi Reset Callout	FlsQspiResetCallout0
Fls Qspi Program and Erase Error Check Callout	FlsQspiErrorCheckCallout0
Fls Qspi Ecc Check Callout	FlsQspiEccCheckCallout0

4.2 Function Calls during Shutdown

None.

4.3 Function Calls during Wake-up

None.

Chapter 5

Module requirements

- Exclusive areas to be defined in BSW scheduler
- Exclusive areas not available on this platform
- Peripheral Hardware Requirements
- ISR to configure within AutosarOS - dependencies
- ISR Macro
- Other AUTOSAR modules - dependencies
- Data Cache Restrictions
- User Mode support
- multicore_support

5.1 Exclusive areas to be defined in BSW scheduler

In the current implementation, Fls is using the services of Schedule Manager (SchM) for entering and exiting the exclusive areas. The following critical regions are used in the Fls driver:

FLS_EXCLUSIVE_AREA_10 is used in function Fls_Erase to protect the updates for:

- Fls_eJobResult
- Fls_u32JobSectorIt
- Fls_u32JobSectorEnd
- Fls_eJob
- Fls_u8JobStart
- Fls_eJobResult

FLS_EXCLUSIVE_AREA_11 is used in function Fls_Write to protect the updates for:

- Fls_eJobResult
- Fls_u32JobSectorIt
- Fls_u32JobSectorEnd
- Fls_u32JobAddrEnd
- Fls_u32JobAddrIt
- Fls_pJobDataSrcPtr
- Fls_eJob
- Fls_u8JobStart
- Fls_eJobResult

FLS_EXCLUSIVE_AREA_12 is used in function Fls_Read to protect the updates for:

- Fls_eJobResult
- Fls_u32JobSectorIt
- Fls_u32JobSectorEnd
- Fls_u32JobAddrIt
- Fls_u32JobAddrEnd
- Fls_pJobDataDestPtr
- Fls_u8JobStart
- Fls_eJobResult

FLS_EXCLUSIVE_AREA_13 is used in function Fls_Compare to protect the updates for:

- Fls_eJobResult
- Fls_u32JobSectorIt
- Fls_u32JobSectorEnd
- Fls_u32JobAddrIt
- Fls_u32JobAddrEnd
- Fls_pJobDataSrcPtr
- Fls_eJob
- Fls_u8JobStart
- Fls_eJobResult

FLS_EXCLUSIVE_AREA_14 is used in function Fls_BlankCheck to protect the updates for:

- Fls_eJobResult

Module requirements

- Fls_u32JobSectorIt
- Fls_u32JobAddrIt
- Fls_u32JobAddrEnd
- Fls_eJob
- Fls_u8JobStart
- Fls_eJobResult

The critical regions from interrupts are grouped in “Interrupt Service Routines Critical Regions (composed diagram)”. If an exclusive area is “exclusive” with the composed “Interrupt Service Routines Critical Regions (composed diagram)” group, it means that it is exclusive with each one of the ISR critical regions.

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	AREA_11	AREA_12	AREA_10	AREA_13	AREA_14
AREA_11		X	X	X	X
AREA_12	X		X	X	X
AREA_10	X	X		X	X
AREA_13	X	X	X		X
AREA_14	X	X	X	X	

5.2 Exclusive areas not available on this platform

List of exclusive areas which are not available on this platform (or blank if they're all available).

5.3 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 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.

5.4 ISR to configure within AutosarOS - dependencies

None.

5.5 ISR Macro

RTD 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.

5.5.1 Without an Operating System The macro `USING_OS_AUTOSAROS` must not be defined.

5.5.1.1 Using Software Vector Mode

The macro `USE_SW_VECTOR_MODE` must be defined and the ISR macro is defined as:

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

In this case, the drivers' interrupt handlers are normal C functions and their prologue/epilogue will handle the context save and restore.

5.5.1.2 Using Hardware Vector Mode

The macro `USE_SW_VECTOR_MODE` must not be defined and the ISR macro is defined as:

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

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

5.5.2 With an Operating System Please refer to your OS documentation for description of the ISR macro.

5.6 Other AUTOSAR modules - dependencies

- **Base:** The BASE module contains the common files/definitions needed by all RTD modules.
- **Det:** The DET module is used for enabling Development error detection. The API function used are `Det_ReportError()` or `Det_ReportRuntimeError()` or `Det_ReportTransientFault()`. 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. RTD modules need ECUC to retrieve the variant information.
- **Mcl:** This module provides service for Cache operation.
- **Os:** The OS module is used for OS configuration. RTD modules need OS to retrieve the application information.

5.7 Data Cache Restrictions

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

1. Disable data cache
2. Configure the flash region upon which the driver operates, as non-cacheable
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 A↔PI 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.
- 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. The FLS driver will check and only invalidate cache for P-Flash sectors, the D-Flash ones will not be touched.
- Additionally, `PCCMR[R2]` should be programmed as 00b (NonCacheable) as S32K148 FlexNVMs region is not cacheable. The device may show unexpected behavior when cache tries to access FlexNVM region. An example can be found in the examples of S32K148. For more information, please refer to the LMEM chapter in the Reference manual

5.8 User Mode support

- [User Mode configuration in the module](#)
- [User Mode configuration in AutosarOS](#)

5.8.1 User Mode configuration in the module

The Fls can be run in user mode if the following steps are performed:

- Enable **FlsEnableUserModeSupport** from the configuration
- Call the following functions as trusted functions:

Function syntax	Description	Available via
void <code>Ftfc_Fls_Ip_Invalid↔PrefetchBuff_Ram(void)</code>	Invalidate prefetch buffer before reading to make sure that the driver always reads the new data from flash	<code>Ftfc_Fls_Ip_Trusted↔Functions.h</code>

5.8.2 User Mode configuration in AutosarOS

When User mode is enabled, the driver may have the functions that need to be called as trusted functions in AutosarOS context. Those functions are already defined in driver and declared in the header `<IpName>_Ip_TrustedFunctions.h`. This header also included all headers files that contains all types definition used by parameters or return types of those functions. Refer the chapter [User Mode configuration in the module](#) for more detail about those functions and the name of header files they are declared inside. Those functions will be called indirectly with the naming convention below in order to AutosarOS can call them as trusted functions.

```
Call_<Function_Name>_TRUSTED(parameter1,parameter2,...)
```

That is the result of macro expansion `OsIf_Trusted_Call` in driver code:

```
#define OsIf_Trusted_Call[1-6params](name,param1,...,param6) Call_##name##_TRUSTED(param1,...,param6)
```

So, the following steps need to be done in AutosarOS:

- Ensure `MCAL_ENABLE_USER_MODE_SUPPORT` macro is defined in the build system or somewhere global.
- Define and declare all functions that need to call as trusted functions follow the naming convention above in Integration/User code. They need to be visible in `Os.h` for the driver to call them. They will do the marshalling of the parameters and call `CallTrustedFunction()` in OS specific manner.
- `CallTrustedFunction()` will switch to privileged mode and call `TRUSTED_<Function_Name>()`.
- `TRUSTED_<Function_Name>()` function is also defined and declared in Integration/User code. It will un-marshalling of the parameters to call `<Function_Name>()` of driver. The `<Function_Name>()` functions are already defined in driver and declared in `<IpName>_Ip_TrustedFunctions.h`. This header should be included in OS for OS call and indexing these functions.

See the sequence chart below for an example calling `Linflexd_Uart_Ip_Init_Privileged()` as a trusted function.

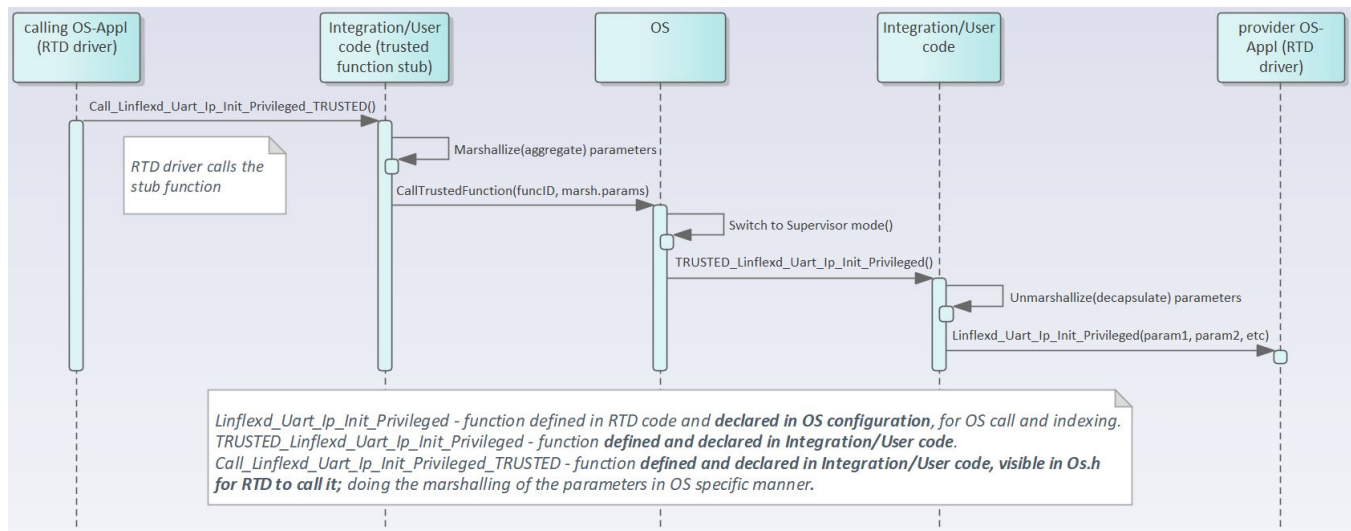


Figure 5.1 Example sequence chart for calling `Linflexd_Uart_Ip_Init_Privileged` as trusted function

Chapter 6

Main API Requirements

- [Main function calls within BSW scheduler](#)
- [API Requirements](#)
- [Calls to Notification Functions, Callbacks, Callouts](#)

6.1 Main function 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:

Notification	Usage
FlsAcCallback	Usually routed to Wdg module
FlsJobEndNotification	Usually routed to Fee module
FlsJobErrorNotification	Usually routed to Fee module
FlsStartFlashAccessNotif	Mark the start of an internal flash read, program access
FlsFinishedFlashAccessNotif	Mark the end of an internal flash read, program access
FlsQspiInitCallout	Perform additional checks and configurations on the external memory at the end of FLS initialization

Main API Requirements

Notification	Usage
FlsQspiResetCallout	Perform reset or cancel on external memory, depending on the available reset mechanisms
FlsQspiErrorCheckCallout	Check any errors occurred in external memories during erase or program operations
FlsQspiEccCheckCallout	Interrogate the ECC status of the memory after each read operation
FlsReadFunctionCallout	Make FLS ECC exception handling can be integrated with Autosar OS. In this callout, user can implement the reading from Flash to Ram buffer, call OS synchronization/task killing, etc. Please see the chapter "ECC Management on Flash" for more information

- [Internal flash access notifications](#)

6.3.1 Internal flash access notifications

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

- FlsStartFlashAccessNotif
- 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

- FlsFinishedFlashAccessNotif is only called after flash memory read accesses (in read, compare, verify write, verify erase jobs)
- 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).

Chapter 7

Memory allocation

- [Sections to be defined in Fls_MemMap.h](#)
- [Linker command file](#)

7.1 Sections to be defined in Fls_MemMap.h

Index	Section name	Type of section	Description
1	FLS_START_SEC_CODE	Code	Start of memory Section for Code.
	FLS_STOP_SEC_CODE		End of above section.
2	FLS_START_SEC_CODE_AC	Code	Start of memory section for Code placed in a specific linker section.
	FLS_STOP_SEC_CODE_AC		End of above section.
3	FLS_START_SEC_RAMCODE	Code	Start of memory section for code placed and executed from RAM.
	FLS_STOP_SEC_RAMCODE		End of above section.
4	FLS_START_SEC_CONFIG_DATA_8	Configuration Data	Start of memory Section for Configuration Data that have to be aligned to 8 bit.
	FLS_STOP_SEC_CONFIG_DATA_8		End of above section.
5	FLS_START_SEC_CONFIG_DATA_UNSPECIFIED	Configuration Data	Start of Memory Section for Config Data. Used for variables, constants, structure, array and unions when SIZE (alignment) does not fit the criteria of 8,16 or 32 bit. For instance used for variables of unknown size
	FLS_STOP_SEC_CONFIG_DATA_UNSPECIFIED		End of above section.
6	FLS_START_SEC_CONST_32	Constant	Data Used for constants that have to be aligned to 32 bit.
	FLS_STOP_SEC_CONST_32		End of above section.
7	FLS_START_SEC_CONST_UNSPECIFIED	Constant Data	The parameters that are not variant aware shall be stored in memory section for constants.

Memory allocation

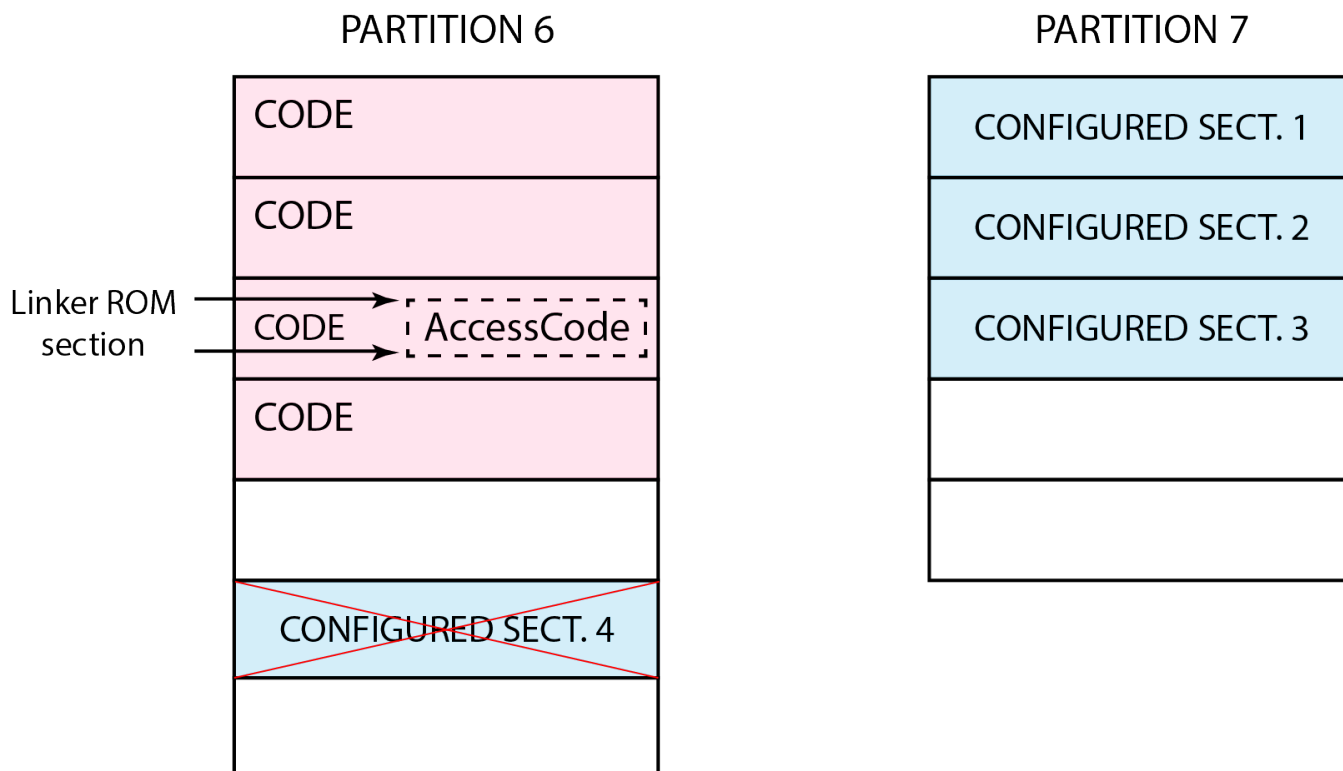
Index	Section name	Type of section	Description
	FLS_STOP_SEC_CONST_UNSP← ECIFIED		End of above section.
8	FLS_START_SEC_VAR_INIT_8	Variables	Start of memory Section for Variable declaration that have to be aligned to 8 bit.
	FLS_STOP_SEC_VAR_INIT_8		End of above section.
9	FLS_START_SEC_VAR_INIT_← UNSPECIFIED	Variables	Used for variables, structures, arrays when the SIZE (alignment) does not fit the criteria of 8,16 or 32 bit. These variables are never cleared and never initialized by start-up code
	FLS_STOP_SEC_VAR_INIT_U← NSPECIFIED		End of above section.
10	FLS_START_SEC_VAR_CLEA← RED_8	Variables	Start of Memory Section for Variable 8 bits. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_CLEAR← ED_8		End of above section.
11	FLS_START_SEC_VAR_CLEA← RED_16	Variables	Start of Memory Section for Variable 16 bits. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_CLEAR← ED_16		End of above section.
12	FLS_START_SEC_VAR_CLEA← RED_32	Variables	Start of Memory Section for Variable 32 bits. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_CLEAR← ED_32		End of above section.
13	FLS_START_SEC_VAR_CLEA← RED_BOOLEAN	Variables	Start of memory Section for Variables with type boolean. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_CLEAR← ED_BOOLEAN		End of above section.
14	FLS_START_SEC_VAR_CLEA← RED_UNSPECIFIED	Variables	Start of memory Section for Variables. Used for variables, constants, structure, array and unions when SIZE (alignment) does not fit the criteria of 8, 16 or 32 bit. For instance used for variables of unknown size. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_CLEAR← ED_UNSPECIFIED		End of above section.

Linker command file Access Code section

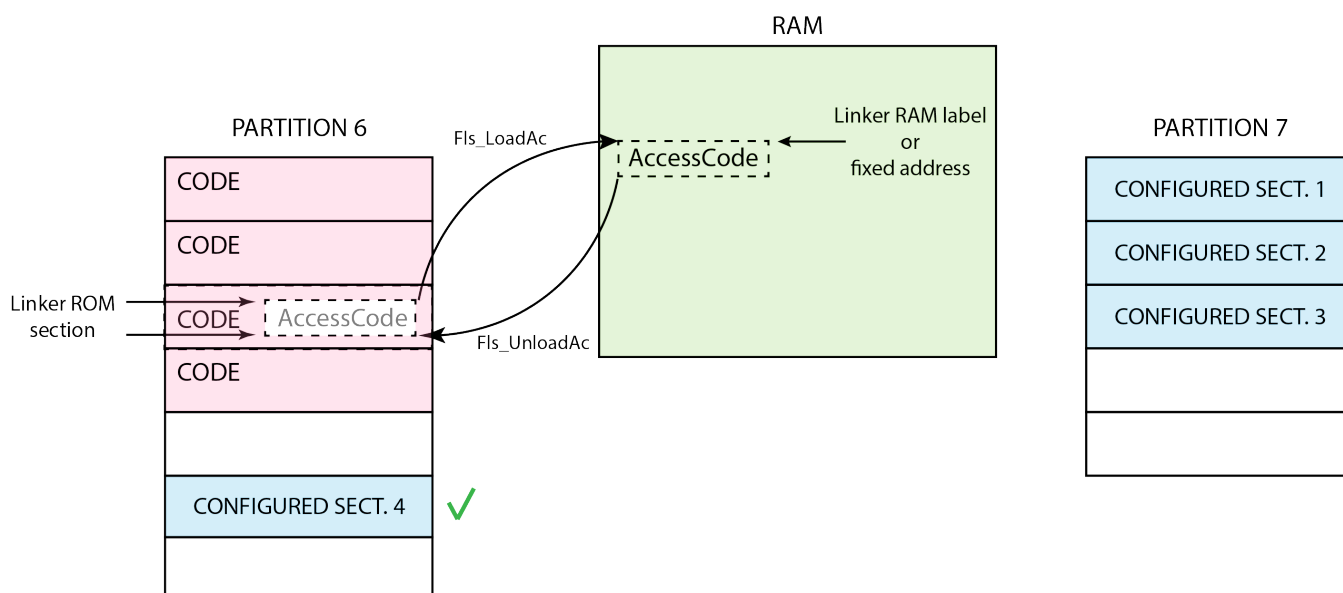
The "Fls_Flash_AccessCode" function executes the actual hardware write/erase operations.

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, "3.6.2 Avoiding RWW problem" chapter.



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.



Memory allocation

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:

GCC linker command file example:

```
....

SECTIONS
{
    .flash :
    {
        acfls_code_rom_start = .;
        . = ALIGN(0x4);
        (.acfls_code_rom)
        acfls_code_rom_end = .;

    } > int_flash

    .acfls_code_ram :
    {
        . += (acfls_code_rom_end - acfls_code_rom_start );
    } > int_sram

    ;Fls module access code support
    Fls_ACERaseRomStart      = acfls_code_rom_start;
    Fls_ACERaseRomEnd        = acfls_code_rom_end;
    Fls_ACERaseSize          = acfls_code_rom_end - acfls_code_rom_start;

    Fls_ACWriteRomStart      = acfls_code_rom_start;
    Fls_ACWriteRomEnd        = acfls_code_rom_end;
    Fls_ACWriteSize          = acfls_code_rom_end - acfls_code_rom_start;

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

7.2 Linker command file

Memory shall be allocated for every section defined in the driver's "<Module>_MemMap.h.



Chapter 8

Integration Steps

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

1. Generate the required module configuration(s). For more details refer to section [Files Required for Compilation](#)
2. Allocate the proper memory sections in the driver's memory map header file ("`<Module>_MemMap.h`") and linker command file. For more details refer to section [Sections to be defined in `<Module>_MemMap.h`](#)
3. Compile & build the module with all the dependent modules. For more details refer to section [Building the Driver](#)

Chapter 9

External assumptions for driver

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

External Assumption Req ID	External Assumption Text
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. This requiremnt only apply for internal flash.
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 - Header File - Description - Det_ReportRuntimeError - Det.h - Service to report runtime errors. If a callout has been configured then this callout shall be called. - Note: Out of scope sMcal
SWS_Fls_00261	API function - Header File - Description - Det_ReportError - Det.h - Service to report development errors. - Note: Out of scope sMcal
SWS_Fls_00262	Service name: - Fee_JobEndNotification - Syntax: - void Fee_JobEnd←Notification(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. - Available via: - Fee.h - Note: Configurable interface (Fee_JobEndNotification() callback).
SWS_Fls_00263	Service name: - Fee_JobErrorNotification - Syntax: - void Fee_JobError←Notification(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. - Available via: - Fee.h - Note: Configurable interface (Fee_JobError←Notification() callback).
EA_RTD_00071	If interrupts are locked, a centralized function pair to lock and unlock interrupts shall be used.
EA_RTD_00072	The option "Fls Hardware Timeout Handling" shall be enabled in the F←LS driver configuration and the timeout value shall be configured to fit the application timing.

External Assumption Req ID	External Assumption Text
EA_RTD_00080	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) .
EA_RTD_00082	When caches are enabled and data buffers are allocated in cacheable memory regions the buffers involved in DMA transfer shall be aligned with both start and end to cache line size. Note: Rationale: This ensures that no other buffers/variables compete for the same cache lines.
EA_RTD_00087	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.
EA_RTD_00088	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.
EA_RTD_00090	The Init job resets the driver's state(Example: clear SEMA4s, abort H↔W 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.
EA_RTD_00091	If the FLS synchronization feature is enabled, the FLS driver instances are not allowed to share any common flash sectors in their configuration.
EA_RTD_00104	In order to handle Flash ECC exceptions through ProtectionHook(), the user shall call the Flash_ReadEachBlock function inside a task that is started from the configurable callback guarded by FlsECCHandlingProtectionHook. Rationale: The task can be can be forcibly terminated by ProtectionHook() in case of an ECC exception.
EA_RTD_00105	If the Fls_DsiHandler() confirms that the ECC exception is caused by FLS driver read (function returns FLS_HANDLED_SKIP) then the protection hook shall return PRO_TERMINATETASKISR, in order for the Operating System to forcibly terminate the faulty task.
EA_RTD_00106	Standalone IP configuration and HL configuration of the same driver shall be done in the same project
EA_RTD_00107	The integrator shall use the IP interface only for hardware resources that were configured for standalone IP usage. Note: The integrator shall not directly use the IP interface for hardware resources that were allocated to be used in HL context.
EA_RTD_00108	The integrator shall use the IP interface to a build a CDD, therefore the BSWMD will not contain reference to the IP interface

9.1 Application's tasks

- It is responsibility of integrator/application to ensure that MCU-wide parameters like voltage supply etc. are according to and in limits specified in MCU documentation.
- Integrator/application is responsible to implement additional functionality that cancel any on-going erase/write Fls jobs if MCU conditions are not in such limits.

9.2 External memory

Flash driver shall support all Flash external memories that can be used via the QuadSPI Controller. These memories should:

- Fit to the LUT programming model
- Be electrically compatible with the QuadSPI IP
- The driver shall support initialization of such memories as well as configuring the command sequences for basic functionality (Flash Writes, Reads, Erase, etc)
- Rationale: In order to allow a wide array of Flash external memories that might be on customer boards, we allow the configuration of any memory that can be configured through the QuadSpi Controller
- To support various Flash external memories, the driver has callouts to allow configuring device specific features, for example the **FlsQspiInitCallout** field allows configuration of memory specific features during Init

Chapter 10

Tips for FLS integration

10.1 Synchronous vs. Asynchronous mode

- 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.

10.2 Speculative prefetch buffer of internal flash

- Regarding the reference manual, no read/erase/execute/etc operations from the Flash memory should be made while writing to the OCMDRn register to disable the prefetch buffers
- Therefore, the function "Ftfc_Fls_Ip_InvalidPrefetchBuff_Ram" should be placed and executed from RAM, because it accesses the flash prefetch buffers, which should not be modified while reading code instructions
- In practice, this function still works normally from FLASH because it might be loaded to the I_CACHE (Cache Instruction) and executed from there
- However, on platforms that do not support cache or cache is disabled, this functions must be placed on RAM
- The configuration node "FlsInvalidPrefetchBuffFromRam" on the GUI allows users to choose where the function "Ftfc_Fls_Ip_InvalidPrefetchBuff_Ram" will be allocated (RAM or FLASH)

Note

- This configuration is the result of the requirement number CPR_RTD_00515.fl: "Driver shall ensure that all the functionalities that are executable from RAM, can be disabled. All limitations for execution from RAM shall be stated in the Integration manual. Rationale: On some platforms, the RAM may not be executable due to security restrictions."

- [Flash Configuration Field](#)
- [ECC Management](#)

10.3 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.

10.3.1 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 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:

- 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 unprotected code can be found in Fls driver UM document.

10.3.2 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.

10.4 ECC Management

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

- [ECC Management on Internal Flash](#)
- [ECC Management on Qspi Flash](#)

10.4.1 ECC Management on Internal Flash

10.4.1.1 Solution 1: Recover from the exception by manually incrementing the program counter (PC register)

- While reading from the Flash, if an ECC exception occurs, a HardFaultHandler will be raised, where current instruction is skipped and the ECC exception will be confirmed.
- The current instruction is skipped by the user in the HardFaultHandler, while the ECC confirmation is done by calling an API provided by the Flash driver:
- `Fls_CompHandlerReturnType Fls_DsiHandler(const Fls_ExceptionDetailsType * pExceptionDetailsPtr);`

Note

This API is available only if the configuration parameter `FlsECCCheck = true`

- The `Fls_ExceptionDetailsType` data structure contains some information about the details of Exception and in particular:
 - The pointer to the instruction that generated the ECC (`Fls_InstructionAddressType`)
 - The data address that caused the ECC error (`Fls_DataAddressType`)
 - Details on the type of exception (`uint32`)
- The `Fls_DsiHandler` 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

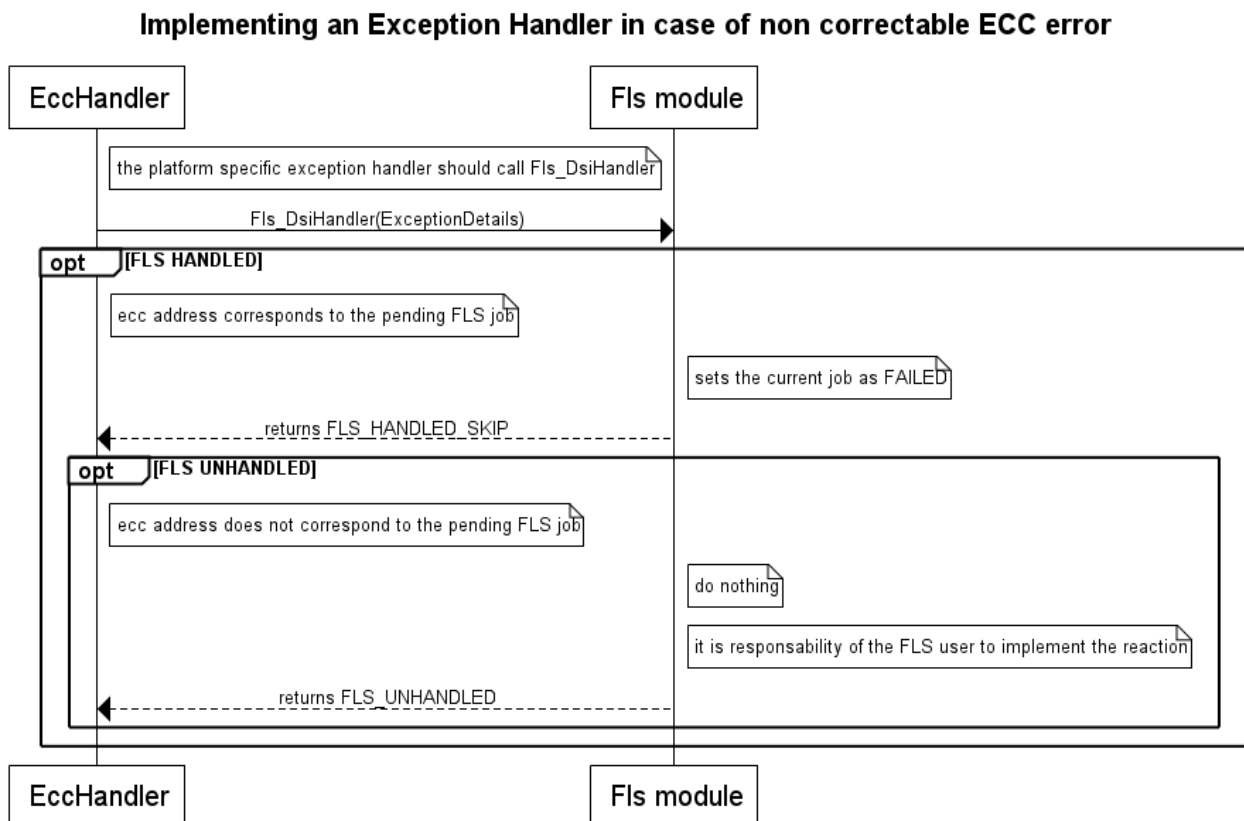
Tips for FLS integration

- 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 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)

Note

S32K118 derivative does not check syndrome and data address because it uses the M0++ core missing some registers related to read syndrome (CFSR) and data address (BFAR)

- In addition to this information, a basic flow and an implementation idea for the fault handler is depicted below.



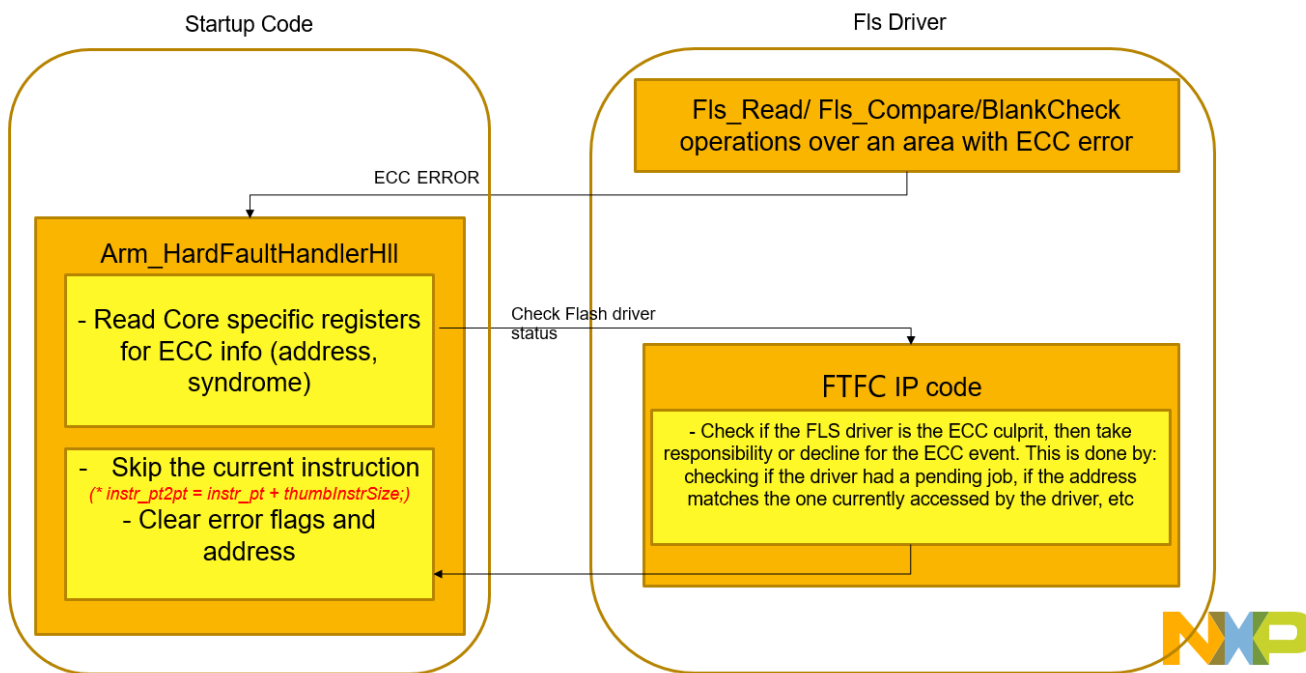
10.4.1.2 Solution 2: Using a callout read function to adapt with Autosar Os

- Background: The AUTOSAR OS specification (AUTOSAR_SWS_OS) states that the OS is responsible for handling exceptions. When an exception occurs, the operating system calls ProtectionHook(), and uses the return value to determine to recover from the exception. The minimum reaction defined by AUTOSAR is to terminate the task or ISR in which the exception occurred. PRO_IGNORE does not apply here because the reaction is only specified for certain timing protection faults.

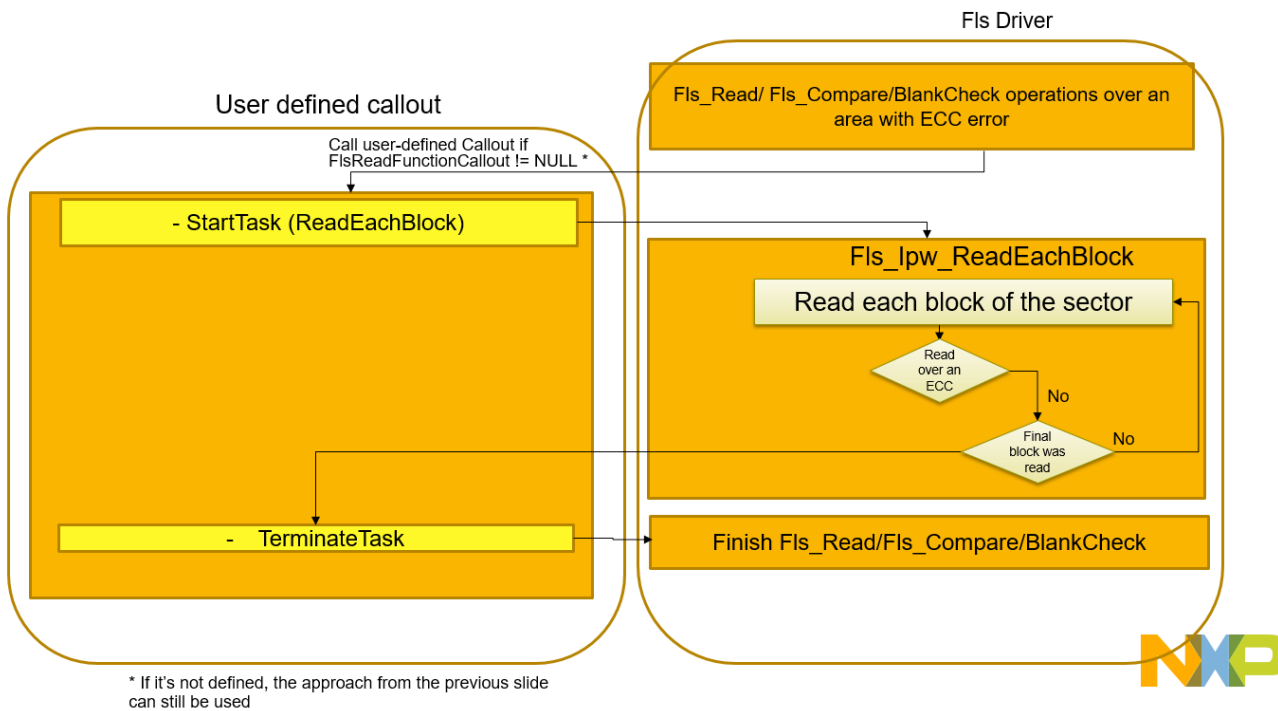
Operating system vendors may provide addition reactions, such as attempting to return to the faulty task. However, such an action can only work reliably if the ProtectionHook() can eliminate the cause of the exception.

- One proposed way of recovering from an exception has been to retrun to the task after making a small modification to the task's program counter. This is problematic for several reasons:
 1. It requires adaptations in the operating system that are specific to the hardware vendor. In a standard product, such adaptations come with a constant test and maintenance overhead.
 2. The adaptations usually require a callout function in the early exception handling. Such software makes it impossible to verify and validate the operating system as a unit. This is especially a problem when the operating system is intended to be used to provide freedom from interference as a safety element out of context.
 3. The "small modification" is usually to advance the program counter to the next instruction. This might not be valid; it depends on the compiler and compiler options.
 4. Modifying the task's program counter interferes with the control flow of the task would be a direct violation of one of the assumed safety requirements. To be fully AUTOSAR compatible, it is therefore desirable to use the features defined by AUTOSAR. That means that the task in which the exception occurs must be terminated. If that task's only function is to manage the hardware that might cause an exception, it should be straightforward to adapt the task. However, that is rarely the case.
- The solution proposed here is to delegate the part of the operation that might cause an exception to a small task that can be terminated. The main task detects the termination due to exception and performs the recovery actions.
- How it works:
 - CalloutReadFunction (defined by the user): Where user can schedule a task which performs the copy operation (the caller) Care is taken to record the unit that is about to be copied, before the copying starts. If an exception occurs, the task can be terminated and the driver task can attempt to recover.
 - ReadEachBlock (called inside the CalloutReadFunction): This task performs the actual copy operation. The amount of data to copy can be arbitrarily long. If the copy fails because of the expected exception, the task can be forcibly terminated (for example: by the ProtectionHook)
- Previous ECC approach:

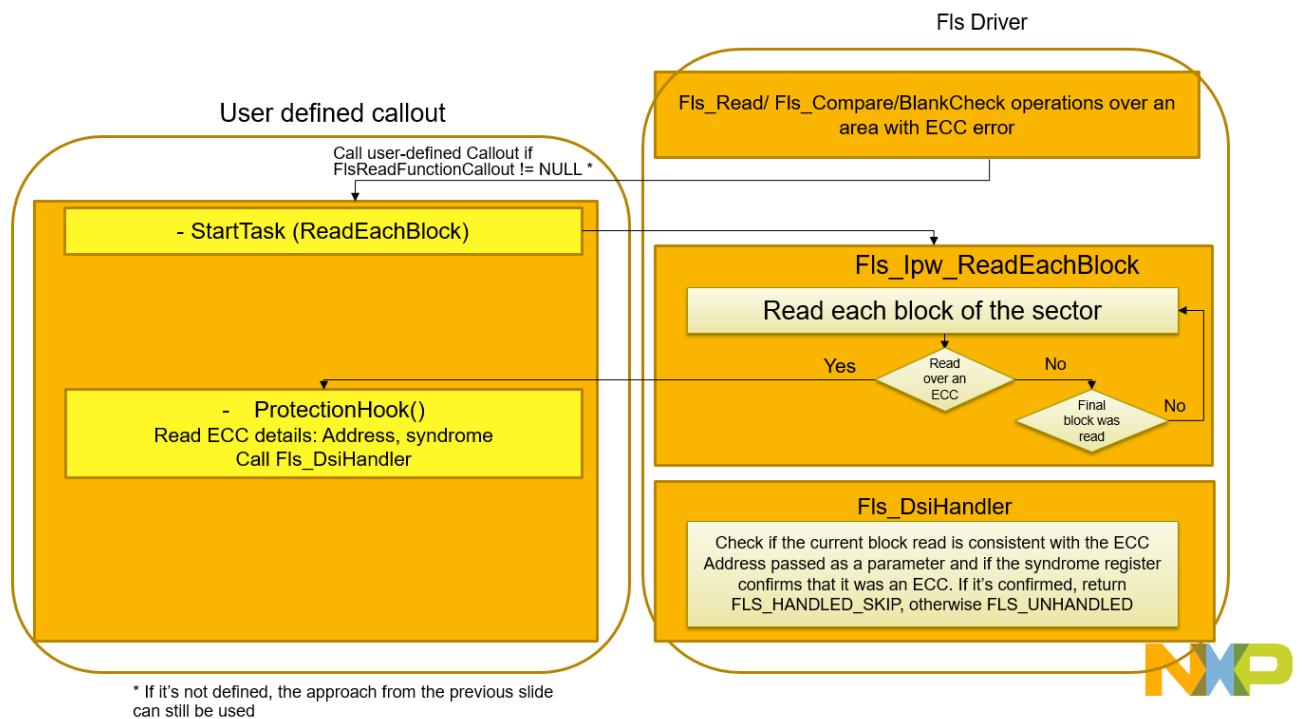
Tips for FLS integration



- New ECC handling approach (No ECC case):



- New ECC handling approach (ECC case):



10.4.2 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.

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.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM, AMBA, ARM Powered, Artisan, Cortex, Jazelle, Keil, SecurCore, Thumb, TrustZone, and Vision are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. ARM7, ARM9, ARM11, big.LITTLE, CoreLink, CoreSight, DesignStart, Mali, mbed, NEON, POP, Sensinode, Socrates, ULINK and Versatile are trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2022 NXP B.V.

