Integration Manual

for S32K3 FLS Driver

Document Number: IM34FLSASR4.4 Rev0000R3.0.0 P01 Rev. 1.0

Chapter 1

Revision History

Revision	Date	Author	Description
1.0	31.03.2023	NXP RTD Team	S32K3 Real-Time Drivers AUTOSAR 4.4 & R21-11 Version 3.0.0

Chapter 2

Introduction

This integration manual describes the integration requirements for FLS Driver for S32K3XX microcontrollers.

AUTOSAR Flash Driver configuration parameters description can be found in the configuration_parameters section. Deviations from the specification are described in the additional_requirements section.

AUTOSAR Flash driver requirements and APIs are described in the Flash Driver Software Specification Document (version 4.4.0) [1] and in the api_reference section.

2.1 Supported Derivatives

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

- s32k310_mqfp100
- s32k310_lqfp48
- s32k311_mqfp100 / MWCT2015S_mqfp100
- s32k311_lqfp48
- s32k312_mqfp100 / MWCT2016S_mqfp100
- s32k312_mqfp172 / MWCT2016S_mqfp172
- s32k314_mqfp172
- s32k314_mapbga257

- s32k322_mqfp100 / MWCT2D16S_mqfp100
- s32k322_mqfp172 / MWCT2D16S_mqfp172
- s32k324_mqfp172 / MWCT2D17S_mqfp172
- s32k324_mapbga257
- s32k341_mqfp100
- s32k341_mqfp172
- s32k342_mqfp100
- s32k342_mqfp172
- s32k344_mqfp172
- s32k344_mapbga257
- s32k394_mapbga289
- s32k396_mapbga289
- s32k358_mqfp172
- s32k358_mapbga289
- s32k328_mqfp172
- s32k328_mapbga289
- s32k338_mqfp172
- s32k338_mapbga289
- s32k348_mqfp172
- s32k348_mapbga289
- s32m274_lqfp64
- s32m276_lqfp64

All of the above microcontroller devices are collectively named as S32K3.

Note: MWCT part numbers contain NXP confidential IP for Qi Wireless Power.

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	S32K3 Real-Time Drivers AUTOSAR Release 4.4.0
2	Reference Manual	S32K3xx Reference Manual, Rev.6, Draft B, 01/2023
		S32K39 and S32K37 Reference Manual, Rev. 2 Draft A, 11/2022
		S32M27x Reference Manual, Rev.2, Draft A, 02/2023
3	Datasheet	S32K3xx Data Sheet, Rev. 6, 11/2022
		S32K396 Data Sheet, Rev. 1.1 — 08/2022
		S32M2xx Data Sheet, Rev. 2 RC — 12/2022
4	Errata	S32K358_0P14E Mask Set Errata — Rev. 28, 9/2022
		S32K396_0P40E Mask Set Errata, Rev. DEC2022, 12/2022
		S32K311_0P98C Mask Set Errata, Rev. 6/March/2023, 3/2023
		S32K312: Mask Set Errata for Mask 0P09C, Rev. 25/April/2022
		S32K342: Mask Set Errata for Mask 0P97C, Rev. 10, 11/2022
		S32K3x4: Mask Set Errata for Mask 0P55A/1P55A, Rev. 14/Oct/2022

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
- DIAB Compiler/Assembler/Linker Options
- GHS Compiler/Assembler/Linker Options
- IAR Compiler/Assembler/Linker Options

The RTD driver files are compiled using:

- NXP GCC 10.2.0 20200723 (Build 1728 Revision g5963bc8)
- Wind River Diab Compiler 7.0.4
- Compiler Versions: Green Hills Multi 7.1.6d / Compiler 2021.1.4
- Compiler Versions: IAR ANSI C/C++ Compiler V8.50.10 (safety version)

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

The TS_T40D34M30I0R0 part of the plugin name is composed as follows:

- T = Target_Id (e.g. T40 identifies Cortex-M architecture)
- D = Derivative_Id (e.g. D34 identifies S32K3 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-m7	Targeted ARM processor for which GCC should tune the performance of the code
-mthumb	Generates code that executes in Thumb state
-mlittle-endian	Generate code for a processor running in little-endian mode
-mfpu=fpv5-sp-d16	Specifies the floating-point hardware available on the target
-mfloat-abi=hard	Specifies the floating-point ABI to use. "hard" allows generation of floating-point instructions and uses FPU-specific calling conventions
-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 aforementioend -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.
-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

Compiler Option	Description
-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	This option is only used to build test for generation Ram/↩ Stack size report. Makes the compiler output stack usage information for the program, on a per-function basis
-fdump-ipa-all	This option is only used to build test for generation Ram/↩ Stack size report. Enables all inter-procedural analysis dumps
-c	Stop after assembly and produce an object file for each source file
-DS32K3XX	Predefine S32K3XX as a macro, with definition 1
-D $ (DERIVATIVE)	Predefine S32K3's derivative as a macro, with definition 1. For example: Predefine for S32K344 will be -DS32K344.
-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
-DD_CACHE_ENABLE	Predefine D_CACHE_ENABLE as a macro, with definition 1. Enables data cache initalization in source file system.↩ c under the Platform driver
-DI_CACHE_ENABLE	Predefine I_CACHE_ENABLE as a macro, with definition 1. Enables instruction cache initalization in source file system.c under the Platform driver
-DENABLE_FPU	Predefine ENABLE_FPU as a macro, with definition 1. Enables FPU initalization in source file system.c under the Platform driver
-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.
–sysroot=	Specifies the path to the sysroot, for Cortex-M7 it is /arm-none-eabi/newlib
-specs=nano.specs	Use Newlib nano specs
-specs=nosys.specs	Do not use printf/scanf

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=cortexm7	Targeted ARM processor for which GCC should tune the performance of the code
-mfpu=fpv5-sp-d16	Specifies the floating-point hardware available on the target
-mfloat-abi=hard	Specifies the floating-point ABI to use. "hard" allows generation of floating-point instructions and uses FPU-specific calling conventions
-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=cortexm7	Targeted ARM processor for which GCC should tune the performance of the code
-mthumb	Generates code that executes in Thumb state
-mfpu=fpv5-sp-d16	Specifies the floating-point hardware available on the target
-mfloat-abi=hard	Specifies the floating-point ABI to use. "hard" allows generation of floating-point instructions and uses FPU-specific calling conventions
-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
-specs=nano.specs	Use Newlib nano specs
-specs=nosys.specs	Do not use printf/scanf

3.1.2 DIAB Compiler/Assembler/Linker Options

3.1.2.1 DIAB Compiler Options

Compiler Option	Description
-tARMCORTEXM7MG:simple	Selects target processor (hardware single-precision, software double-precision floating-point)
-mthumb	Selects generating code that executes in Thumb state
-std=c99	Follows the C99 standard for C
-Oz	Like -O2 with further optimizations to reduce code size
-g	Generates DWARF 4.0 debug information
-fstandalone-debug	Emits full debug info for all types used by the program
-Wstrict-prototypes	Warn if a function is declared or defined without specifying the argument types
-Wsign-compare	Produce warnings when comparing signed type with unsigned type
-Wdouble-promotion	Give a warning when a value of type float is implicitly promoted to double
-Wunknown-pragmas	Issues a warning for unknown pragmas
-Wundef	Warns if an undefined identifier is evaluated in an #if directive. Such identifiers are replaced with zero

Compiler Option	Description
-Wextra	Enables some extra warning flags that are not enabled by '-Wall'
-Wall	Enables all of the most useful warnings (for historical reasons this option does not literally enable all warnings)
-pedantic	Emits a warning whenever the standard specified by the -std option requires a diagnostic
-Werror=implicit-function-declaration	Generates an error whenever a function is used before being declared
-fno-common	Compile common globals like normal definitions
-fno-signed-char	Char is unsigned
-fno-trigraphs	Do not process trigraph sequences
-V	Displays the current version number of the tool suite
-c	Stop after assembly and produce an object file for each source file
-DS32K3XX	Predefine S32K3XX as a macro, with definition 1
-D $ (DERIVATIVE)	Predefine S32K3's derivative as a macro, with definition 1
-DDIAB	Predefine DIAB 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
-DD_CACHE_ENABLE	Predefine D_CACHE_ENABLE as a macro, with definition 1. Enables data cache initalization in source file system.↩ c under the Platform driver
-DI_CACHE_ENABLE	Predefine I_CACHE_ENABLE as a macro, with definition 1. Enables instruction cache initalization in source file system.c under the Platform driver
-DENABLE_FPU	Predefine ENABLE_FPU as a macro, with definition 1. Enables FPU initalization in source file system.c under the Platform driver
-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 DIAB Assembler Options

Assembler Option	Description
-mthumb	Selects generating code that executes in Thumb state
-Xpreprocess-assembly	Invokes C preprocessor on assembly files before running the assembler
-Xassembly-listing	Produces an .lst assembly listing file
-c	Stop after assembly and produce an object file for each source file
-tARMCORTEXM7MG:simple	Selects target processor (hardware single-precision, software double-precision floating-point)

3.1.2.3 DIAB 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
linker_script_file.dld	Use linker_script_file.dld as the linker script. This script replaces the default linker script (rather than adding to it)
-m30	m2 + m4 + m8 + m16
-Xstack-usage	Gathers and display stack usage at link time
-Xpreprocess-lecl	Perform pre-processing on linker scripts
-Llibrary_path	Points to the libraries location for ARMV7EMMG to be used for linking
-lc	Links with the standard C library
-lm	Links with the math library
-tARMCORTEXM7MG:simple	Selects target processor (hardware single-precision, software double-precision floating-point)

3.1.3 GHS Compiler/Assembler/Linker Options

3.1.3.1 GHS Compiler Options

Compiler Option	Description
-cpu=cortexm7	Selects target processor: Arm Cortex M7
-thumb	Selects generating code that executes in Thumb state
-fpu=vfpv5_d16	Specifies hardware floating-point using the v5 version of the VFP instruction set, with 16 double-precision floating-point registers
-fsingle	Use hardware single-precision, software double-precision FP instructions
-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	Bitfelds declared with an integer type are unsigned

Compiler Option	Description
–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 andgenerate 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
-DS32K3XX	Predefine S32K3XX as a macro, with definition 1
-D $ (DERIVATIVE)	Predefine S32K3's derivative as a macro, with definition 1. For example: Predefine for S32K344 will be -DS32K344.
-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
-DD_CACHE_ENABLE	Predefine D_CACHE_ENABLE as a macro, with definition 1. Enables data cache initalization in source file system.↩ c under the Platform driver
-DI_CACHE_ENABLE	Predefine I_CACHE_ENABLE as a macro, with definition 1. Enables instruction cache initalization in source file system.c under the Platform driver
-DENABLE_FPU	Predefine ENABLE_FPU as a macro, with definition 1. Enables FPU initalization in source file system.c under the Platform driver
-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 GHS Assembler Options

Assembler Option	Description
-cpu=cortexm7	Selects target processor: Arm Cortex M7
-fpu=vfpv5_d16	Specifies hardware floating-point using the v5 version of the VFP instruction set, with 16 double-precision floating-point registers
-fsingle	Use hardware single-precision, software double-precision FP instructions
-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

3.1.3.3 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 themodules 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
-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
-nostartfiles	Controls the start files to be linked into the executable

3.1.4 IAR Compiler/Assembler/Linker Options

3.1.4.1 IAR Compiler Options

Compiler Option	Description
–cpu Cortex-M7	Targeted ARM processor for which IAR should tune the performance of the code
–cpu_mode thumb	Generates code that executes in Thumb state
–endian little	Generate code for a processor running in little-endian mode
–fpu VFPv5-SP	Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). Single-precision variant.
-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

Compiler Option	Description
–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
–do_explicit_zero_opt_in_named_sections	Disable the exception for variables in user-named sections, and thus treat explicit initializations to zero as zero initializations, not copy 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
-DS32K3XX	Predefine S32K3XX as a macro, with definition 1
-D $ (DERIVATIVE)	Predefine S32K3's derivative as a macro, with definition 1. For example: Predefine for S32K344 will be -DS32K344.
-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.
-DD_CACHE_ENABLE	Predefine D_CACHE_ENABLE as a macro, with definition 1. Enables data cache initalization in source file system.↩ c under the Platform driver
-DI_CACHE_ENABLE	Predefine I_CACHE_ENABLE as a macro, with definition 1. Enables instruction cache initalization in source file system.c under the Platform driver
-DENABLE_FPU	Predefine ENABLE_FPU as a macro, with definition 1. Enables FPU initalization in source file system.c under the Platform driver
-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.

3.1.4.2 IAR Assembler Options

Assembler Option	Description
–cpu Cortex-M7	Targeted ARM processor for which IAR should generate the instruction set
–fpu VFPv5-SP	Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). Single-precision variant.
–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

3.1.4.3 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-M7	Selects the ARM processor variant to link the application for
–fpu VFPv5-SP	Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). Single-precision variant.
–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 S32K3XX 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

- Fls_TS_T40D34M30I0R0\include\C40_Ip.h
- Fls_TS_T40D34M30I0R0\include\C40_Ip_Types.h
- Fls_TS_T40D34M30I0R0\include\C40_Ip_Ac.h
- Fls_TS_T40D34M30I0R0\include\Fls.h
- Fls_TS_T40D34M30I0R0\include\Fls_Api.h
- Fls_TS_T40D34M30I0R0\include\Fls_IPW.h
- Fls_TS_T40D34M30I0R0\include\Fls_Types.h
- Fls_TS_T40D34M30I0R0\include\Qspi_Ip.h
- Fls_TS_T40D34M30I0R0\include\Qspi_Ip_Types.h
- Fls_TS_T40D34M30I0R0\include\Qspi_Ip_Common.h
- Fls_TS_T40D34M30I0R0\include\Qspi_Ip_Controller.h
- Fls_TS_T40D34M30I0R0\include\Qspi_Ip_HwAccess.h
- Fls_TS_T40D34M30I0R0\src\C40_Ip.c
- Fls_TS_T40D34M30I0R0\src\C40_Ip_Ac.c

- Fls_TS_T40D34M30I0R0\src\Fls.c
- Fls_TS_T40D34M30I0R0\src\Fls_IPW.c
- Fls_TS_T40D34M30I0R0\src\Qspi_Ip.c
- Fls_TS_T40D34M30I0R0\src\Qspi_Ip_Controller.c
- Fls_TS_T40D34M30I0R0\src\Qspi_Ip_Sfdp.c

Note

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

- Fls_Cfg_Defines.h
- Fls_Cfg.h
- Fls_Cfg.c
- Fls_PBcfg.h
- Fls_PBcfg.c
- Qspi_Ip_Cfg.h
- Qspi_Ip_PBcfg.h
- Qspi_Ip_Features.h
- Qspi_Ip_CfgDefines.h
- Qspi_Ip_PBCfg.c
- C40_Ip_Cfg.h
- C40_Ip_PBcfg.c

3.2.2 Files from MemIf folder

- MemIf_TS_T40D34M30I0R0\include\MemIf_Types.h

3.2.3 Files from Base common folder

- Base_TS_T40D34M30I0R0\include\Compiler.h
- Base_TS_T40D34M30I0R0\include\Compiler_Cfg.h
- Base_TS_T40D34M30I0R0\include\ComStack_Types.h
- Base_TS_T40D34M30I0R0\include\Fls_MemMap.h
- Base_TS_T40D34M30I0R0\include\Mcal.h
- Base_TS_T40D34M30I0R0\include\Platform_Types.h
- Base_TS_T40D34M30I0R0\include\Std_Types.h
- Base_TS_T40D34M30I0R0\include\Reg_eSys.h
- Base_TS_T40D34M30I0R0\include\Soc_Ips.h
- Base_TS_T40D34M30I0R0\include\Reg_Macros.h
- Base_TS_T40D34M30I0R0\include\SilRegMacros.h
- Base_TS_T40D34M30I0R0\header\S32K344.h

3.2.4 Files from Det folder

- Det_TS_T40D34M30I0R0\include\Det.h

3.2.5 Files from Rte folder

- Rte_TS_T40D34M30I0R0\include\SchM_Fls.h

3.2.6 Files from Mcl folder

- Mcl_TS_T40D34M30I0R0\include\CDD_Mcl.h

3.3 Setting up the plugins

The Flash Driver was designed to be configured by using the EB Tresos Studio (version 29.0.0 b220329-0119 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_T40D34M30I0R0\config\Fls.xdm
- VSMD (Vendor Specific Module Definition) file(s) in AUTOSAR compliant EPD format:
  - Fls_TS_T40D34M30I0R0\autosar\Fls_<subderivative_name>.epd
- Code Generation Templates for variant aware parameters:
  - Fls_TS_T40D34M30I0R0\generate_PB\include\Fls_PBcfg.h
  - Fls_TS_T40D34M30I0R0\generate_PB\src\C40_Ip_PBcfg.c
  - Fls_TS_T40D34M30I0R0\generate_PB\src\Fls_PBcfg.c
  - Fls_TS_T40D34M30I0R0\generate_PB\src\Qspi_Ip_PBcfg.c
  - Fls_TS_T40D34M30I0R0\generate_PB\include\Qspi_Ip_PBcfg.h
- Code Generation Templates for parameters without variation points:
  - Fls_TS_T40D34M30I0R0\generate_PC\include\C40_Ip_Cfg.h
  - Fls_TS_T40D34M30I0R0\generate_PC\include\Fls_Cfg.h
  - Fls_TS_T40D34M30I0R0\generate_PC\include\Qspi_Ip_Cfg.h
  - Fls_TS_T40D34M30I0R0\generate_PC\include\Qspi_Ip_Features.h
  - Fls_TS_T40D34M30I0R0\generate_PC\include\Qspi_Ip_CfgDefines.h
  - Fls_TS_T40D34M30I0R0\generate_PC\src\Fls_Cfg.c

3.3.2 Steps to generate the configuration:

1. Copy the module folders:
   - Base_TS_T40D34M30I0R0
   - Det_TS_T40D34M30I0R0
   - EcuC_TS_T40D34M30I0R0
   - Fls_TS_T40D34M30I0R0
   - MemIf_TS_T40D34M30I0R0
   - Platform_TS_T40D34M30I0R0
   - Resource_TS_T40D34M30I0R0
   - Rte_TS_T40D34M30I0R0
   - Mcl_TS_T40D34M30I0R0 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().

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

- 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

None.

5.3 Peripheral Hardware Requirements

The FLS driver uses the "Flash Memory" MCU peripheral (C40) 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.

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 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

5.7.1 For memory data

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 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 only apply to internal flash operations.

5.7.2 For access code function

S32K3xx has independent cache lines for Data and Instruction (D-CACHE and I-CACHE). This might results in inconsistency when using the feature load the flash access code to RAM (**FlsAcLoadOnJobStart** is enabled). In case the access code function (AC) is written only to the D-CACHE, not propagate to the RAM memory. When invoking the AC function, a hardfault exception will be thrown because of there is no valid data in the I-CACHE or the RAM memory.

This cache coherence problem can be solved by two methods:

1. Configure the RAM momory region upon which the AC function (**C40_Ip_AccessCode**) will be loaded to, as non-cacheable
2. Enable the **FlsCleanCacheAfterLoadAc** feature

FlsSynchronizeCache parameter allows to call Mcl cache API to clean caches after copying the AC function to RAM to ensure the synchronization between cache and RAM memory.

Note

It is highly recommended to define the start addresses to be aligned to the cache line size boundary to avoid unexpected behavior:

- The start address of the data buffers
- The start address on RAM that the AC function will be loaded to

5.8 User Mode support

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 C40_Ip_SetUserAccess↩ Allowed(void)	For seting the user access allowed for C40 registers protected by REG_↩ PROT	C40_Ip_Trusted↩ Functions.h
void C40_Ip_Block4PipeSelect(void)	Setup the active pipe for flash memory block 4 access	C40_Ip_Trusted↩ Functions.h
void C40_Ip_DataError↩ Suppression(void)	Setup the ECC error handling on data flash block	C40_Ip_Trusted↩ Functions.h

Function syntax	Description	Available via
uint32 C40_Ip_GetPflashDataError↩ SuppressionStatus(void)	Read DERR_SUP bit to get the data error suppression status	C40_Ip_Trusted↩ Functions.h
void C40_Ip_SetLockProtect(C40_Ip_VirtualSectorsType Virtual↩ Sector)	Set lock bit for flash sectors	C40_Ip_Trusted↩ Functions.h
uint32 C40_Ip_GetLockProtect(C40_Ip_VirtualSectorsType Virtual↩ Sector)	Read lock bit status of flash sectors	C40_Ip_Trusted↩ Functions.h
void C40_Ip_ClearLockProtect(C40_Ip_VirtualSectorsType Virtual↩ Sector)	Clear lock bit for flash sectors	C40_Ip_Trusted↩ Functions.h
void C40_Ip_MainInterfaceWrite↩ LogicalAddress(uint32 Address)	Write the program erase address using logical address registers located in the Platform Flash Controller	C40_Ip_Trusted↩ Functions.h
void C40_Ip_CheckLockDomainID↩ _CheckRegister(C40_Ip_VirtualSectorsType Virtual↩ Sector, uint32 ∗CheckRegister, uint32 ∗TempLockMasterRegister)	Read and check the lock domain ID for flash sectors	C40_Ip_Trusted↩ Functions.h
void Qspi_Ip_Sfp_Configure_Privileged(QuadSPI_Type ∗ baseAddr, Qspi_Ip_ControllerConfigType const ∗ userConfigPtr)	Configure the SFP registers	Qspi_Ip_Trusted↩ Functions.h
void Qspi_Ip_Sfp_ClearLatchedErrors_↩ Privileged(QuadSPI_Type ∗ BaseAddr)	Clear the errors latched in SFP registers	Qspi_Ip_Trusted↩ Functions.h
void Qspi_Ip_ResetPrivilegedRegisters_↩ Privileged(QuadSPI_Type ∗ BaseAddr)	Reset the registers the require privilege access for programming	Qspi_Ip_Trusted↩ Functions.h
uint16 Qspi_Ip_WriteLuts_Privileged(uint32 Instance, uint8 StartLutRegister, const uint32 ∗Data, uint8 Size)	Configure pairs of LUT commands from the specified LUT register	Qspi_Ip_Trusted↩ Functions.h
void Qspi_Ip_SetAhbSeqId_Privileged(uint32 instance, uint8 seqID)	Sets sequence ID for AHB operations	Qspi_Ip_Trusted↩ Functions.h

5.8.2 User Mode configuration in AutosarOS

When User mode is enabled, the driver may has 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 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 unmarshalling 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.

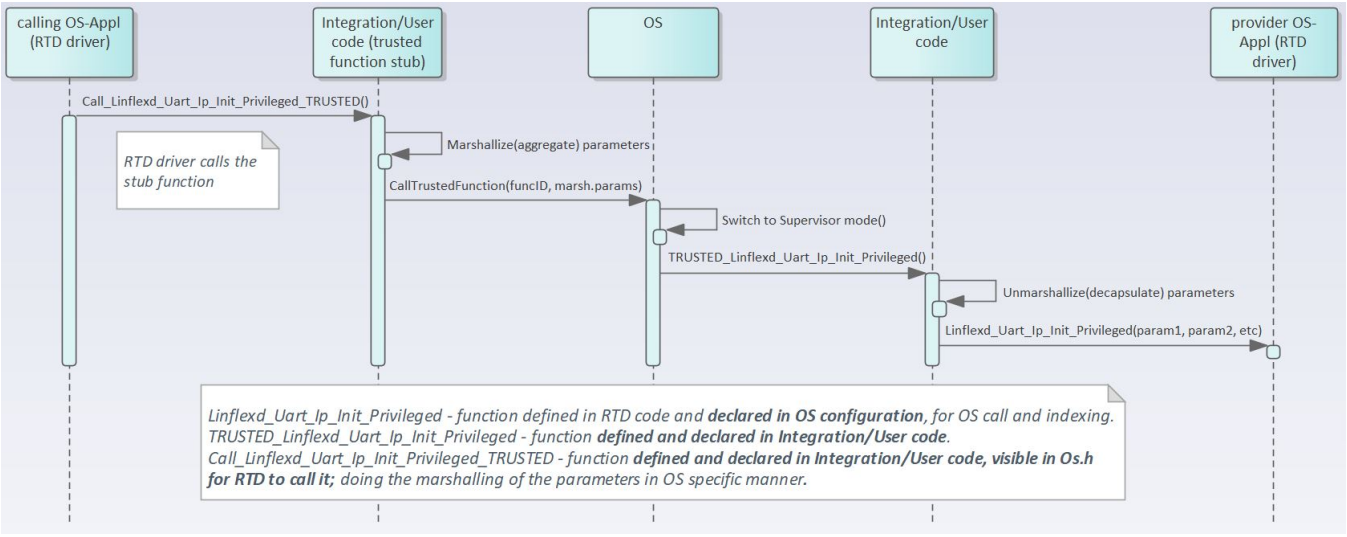


Figure 5.1 Example sequence chart for calling `Linflexd_Uart_Ip_Init_Privileged` as trusted function

5.9 Multicore support

- The Fls implements "Autosar 4.4 MCAL Multi-Core Distribution" according to Type I. For more details, please refer to AUTOSAR_EXP_BSWDistributioGuide.
- The FLS driver supports multicore synchronization if "FlsMCoreEnable" is enabled. Vendor-specific Sema4 multi-core support
- When the synchronization is enabled, the Multicore related parameters (semaphore addresses, timeout values, core number) have to be configured accordingly on both cores
- The synchronization feature arbitrates the access to the flash resource, using the configured hardware semaphores
  - The core which wins the arbitration will access the flash resource and start the flash job in hardware
  - The core which looses the arbitration will mark it's current job as pending until the controller is idle or until timeout occurs

Warning

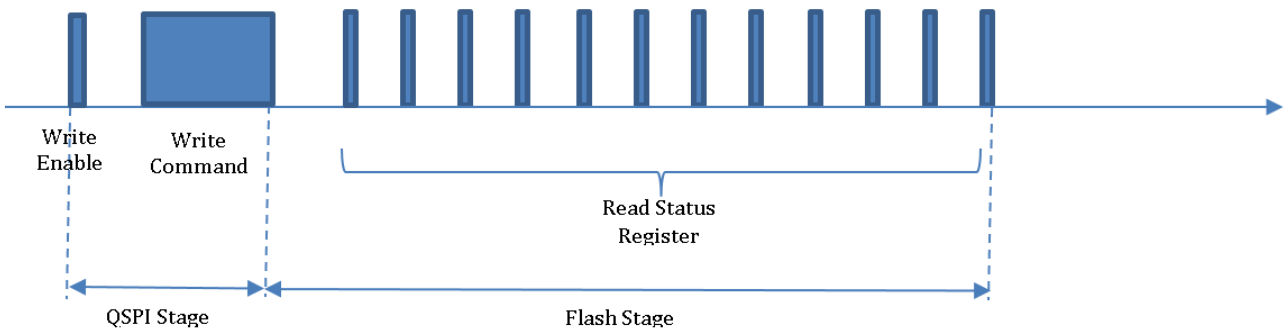
Any synchronization issues at initialization will be avoided if both cores perform the init before starting any other jobs

- External flash driver in multicore context

5.9.1 External flash driver in multicore context

5.9.1.1 Background

- External flash operations consist of two stages. In stage 1 (QSPI stage) the command is sent to the external flash via QSPI controller. In stage 2 (Flash Stage) the actual operation takes place in the external flash and the MCU must poll the status of the operation to find when it ends. Flash stage only applies to Write and Erase operations.



- In multicore context the general restriction is that no two operations (read, write, erase) may be performed in parallel. The QSPI stages of different operations may not overlap because the QSPI module can only do one transmission at a time. For write and erase operations this restriction usually includes the flash stage too, because most serial flashes do not offer read-while-write features. So both stages depicted above must be performed as one atomic operation (not interruptible by other cores).

5.9.1.2 FLS Driver Multicore approach

- QSPI / external flash can be protected against concurrent accesses by an XRDC semaphore [hereinafter only named semaphore]. Each core that wants to make any flash operations (read, write or erase) through QSPI must first take the semaphore. This is done internally by the FLS driver. The semaphore is released at the end of the operation
- To further guard against interference, XRDC must be set to allow write access to QSPI registers only if the semaphore is owned by the respective domain. This ensures the current flash operation cannot be impacted by writes to QSPI registers performed by another (faulty) core
- For write and erase operations the semaphore must be held for the entire duration of the operation, including the flash stage. Starting another operation during the flash stage would result in the command being ignored by the flash device (and dummy data provided in case of read). Read operations also need to own the semaphore, to avoid QSPI contentions or RWW restriction violations in the flash device
- For large operations, which are internally split by the driver in several smaller operations, the semaphore is held for the duration of the entire operation.
- The semaphore is taken on first come – first served basis, there is no arbitration between cores. This can potentially lead to denial of service for an indeterminate amount of time for a core, if other cores continuously perform flash operations and grab the semaphore first

5.9.1.3 AHB Reads

- It is not recommended to perform AHB reads directly from the application, as they may interfere with normal FLS driver functionality. QSPI can only perform one operation at a time, so attempting AHB reads while the module is busy may cause the current operation to fail. It is recommended to do all reads using the FLS driver's Fls_Read() function. This must also be taken into account while debuggers are attached which may also read from flash
- If the application must perform AHB reads, the same XRDC semaphore used by the FLS driver must be used to guard the AHB reads. This can not be enforced by XRDC, as the semaphore only controls write permissions. So in this case it is the software's responsibility to ensure the semaphore is owned before AHB reads, failing to do so could potentially cause flash operations to fail
- If a core only needs to perform read operations during the runtime stage, it can use AHB reads and benefit from the security features of the XRDC, which can ensure the core only has access to the allocated memory range and also can forbid the core's access to the QSPI registers, thus ensuring no interference with other cores' operations. The requirement to use the semaphore still applies

5.9.1.4 Read while write

- Read While Write is not supported, as AUTOSAR only allows one flash operation at a time. This cannot be changed without changing the AUTOSAR API, since the function Fls_GetJobResult() and the job end notifications have no parameters so it is not possible to distinguish between multiple parallel jobs

5.9.1.5 Error recovery

- If one of the cores crashes while owning the semaphores it is possible to recover the semaphore by forcefully resetting it from outside the Fls driver. This mechanism allows either individual or all gates to be reset by a separate domain. If the application has no other way of knowing that another core has crashed, a timeout can be used. The timeout value depends on the application (one core may experience delays caused by multiple operations started from the other cores, as there is no arbitration when taking the semaphore)
- After the semaphore reset any core can initiate flash operations
- If the flash device was left in a busy state after semaphore reset, the driver will automatically issue a software reset command before the next flash operation

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 a flash read, program access
FlsFinishedFlashAccessNotif	Mark the end of a flash read, program access
FlsMCoreTimeoutNotification	Notify the multi core timeout in case FlsMCoreEnable node is enabled

Notification	Usage
FlsQspiInitCallout	Perform additional checks and configurations on the external memory at the end of FLS initialization
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 the internal 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.
- For example: a hard-fault exception might occurr after systick interrupt handler was called during the write (or erase) access code from RAM. Because of the systick interrupt handler is stored in FLASH and leads to a read-while-write error.

Note

- **FlsFinishedFlashAccessNotif** is only called after flash memory read accesses (in read, compare, verify write, verify erase jobs) and after a synchronous write or erase operation. It means that, with asynchronous write or erase operations, this notification will not be invoked.
- 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.

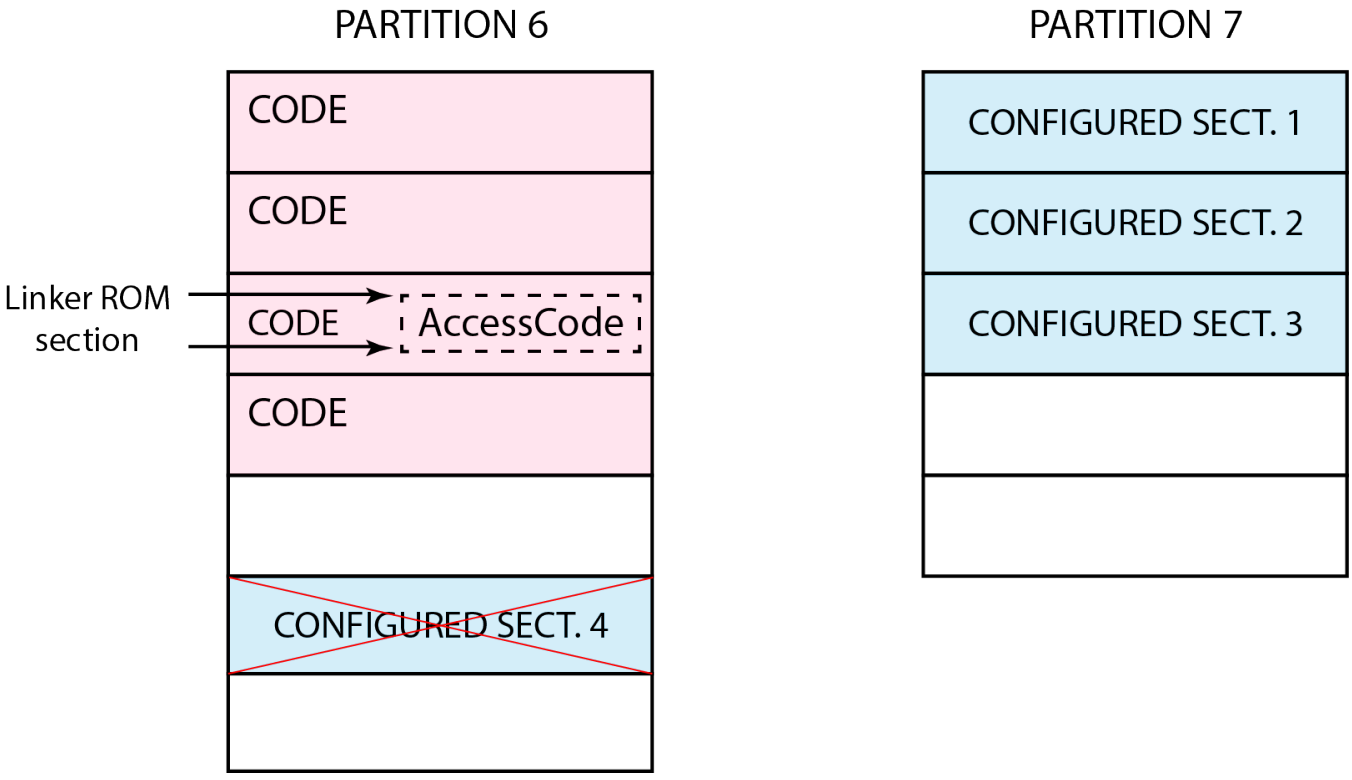
Index	Section name	Type of section	Description
	FLS_STOP_SEC_CONST_↩ UNSPECIFIED		End of above section.
8	FLS_START_SEC_VAR_INIT_8	Variables	Start of memory Section for Variable declaration that have to be aligned to 8 bit.
	FLS_STOP_SEC_VAR_INIT_8		End of above section.
9	FLS_START_SEC_VAR_INIT_↩ UNSPECIFIED	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_↩ UNSPECIFIED		End of above section.
10	FLS_START_SEC_VAR_↩ CLEARED_8	Variables	Start of Memory Section for Variable 8 bits. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_↩ CLEARED_8		End of above section.
11	FLS_START_SEC_VAR_↩ CLEARED_16	Variables	Start of Memory Section for Variable 16 bits. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_↩ CLEARED_16		End of above section.
12	FLS_START_SEC_VAR_↩ CLEARED_32	Variables	Start of Memory Section for Variable 32 bits. These variables are cleared to zero by start-up code.
	FLS_STOP_SEC_VAR_↩ CLEARED_32		End of above section.
13	FLS_START_SEC_VAR_↩ CLEARED_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_↩ CLEARED_BOOLEAN		End of above section.
14	FLS_START_SEC_VAR_↩ CLEARED_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_↩ CLEARED_UNSPECIFIED		End of above section.
15	FLS_START_SEC_VAR_INIT_↩ BOOLEAN	Variables	Start of memory Section for Variables with type boolean.
	FLS_STOP_SEC_VAR_INIT_↩ BOOLEAN		End of above section.

Linker command file Access Code section

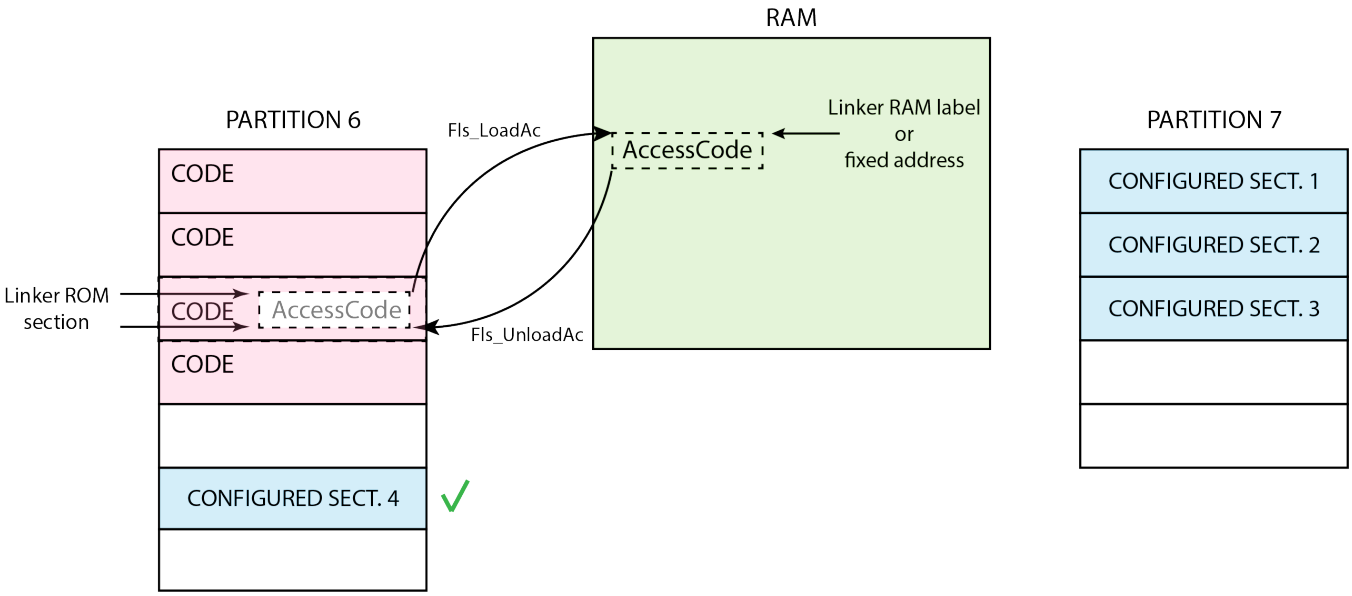
The "C40_Ip_AccessCode" function executes the actual hardware write/erase operations.

The "C40_Ip_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 "C40_Ip_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 "C40_Ip_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.



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

- Tips for FLS integration

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

8.1 Tips for FLS integration

8.1.1 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.

Note

- This configuration is the result of the requirement number CPR_RTD_00515.fls: "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."

- ECC Management on Flash

8.1.2 ECC Management on Flash

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

- For Data Flash:
  - If DERR_SUP on PFLASH_PFCR3 register is set to 1 (by enabling node FlsDataErrorSuppression)
  - Reading the ECC-affected data from the Data flash segments (named FLS_DATA_...) doesn't lead to an exception being raised (unlike reading the ECCs from the Code flash).
  - This mechanism has been integrated into the existing exception management so that by using the same configuration parameter "FlsECCCheck" it is possible to enable the ECC handling for both Code and Data Flash.

- ECC Management on Internal Flash
- ECC Management on Qspi Flash

8.1.2.1 ECC Management on Internal Flash

8.1.2.1.1 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);

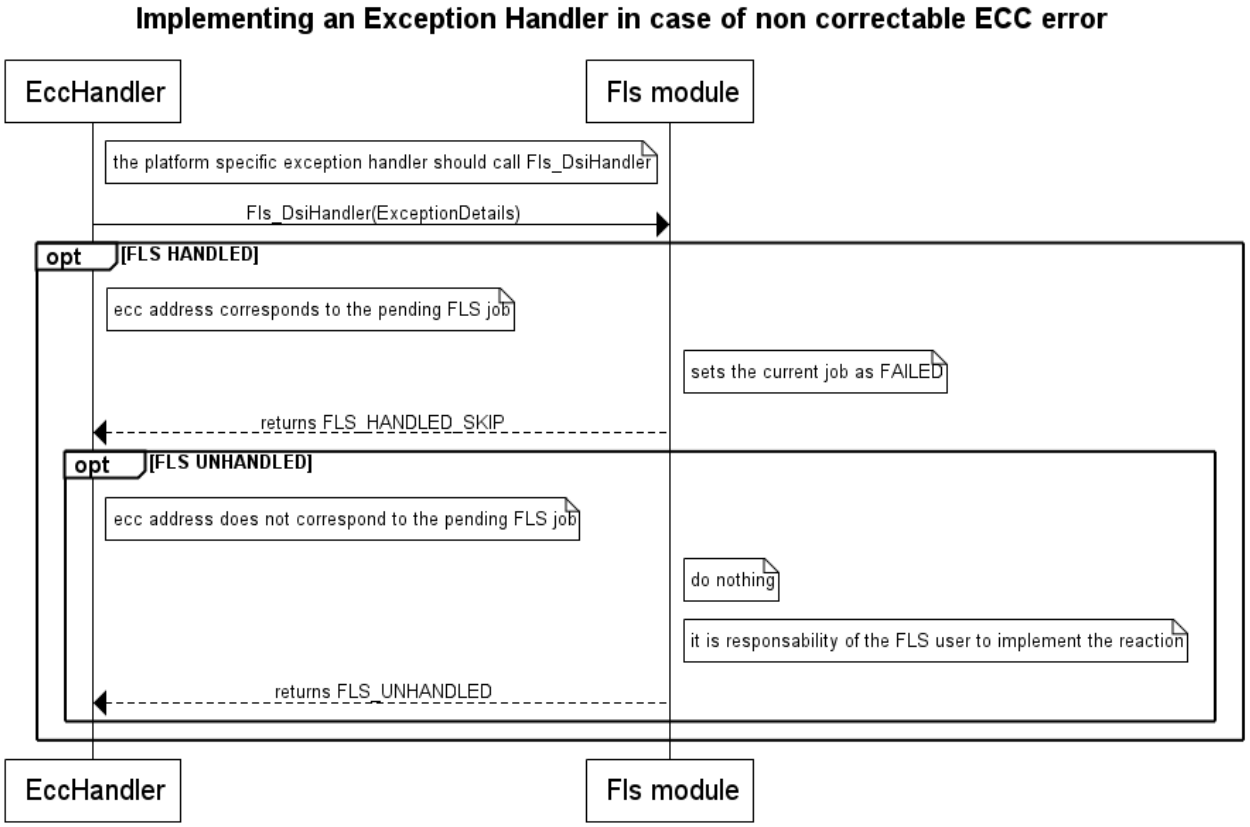
  Note

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

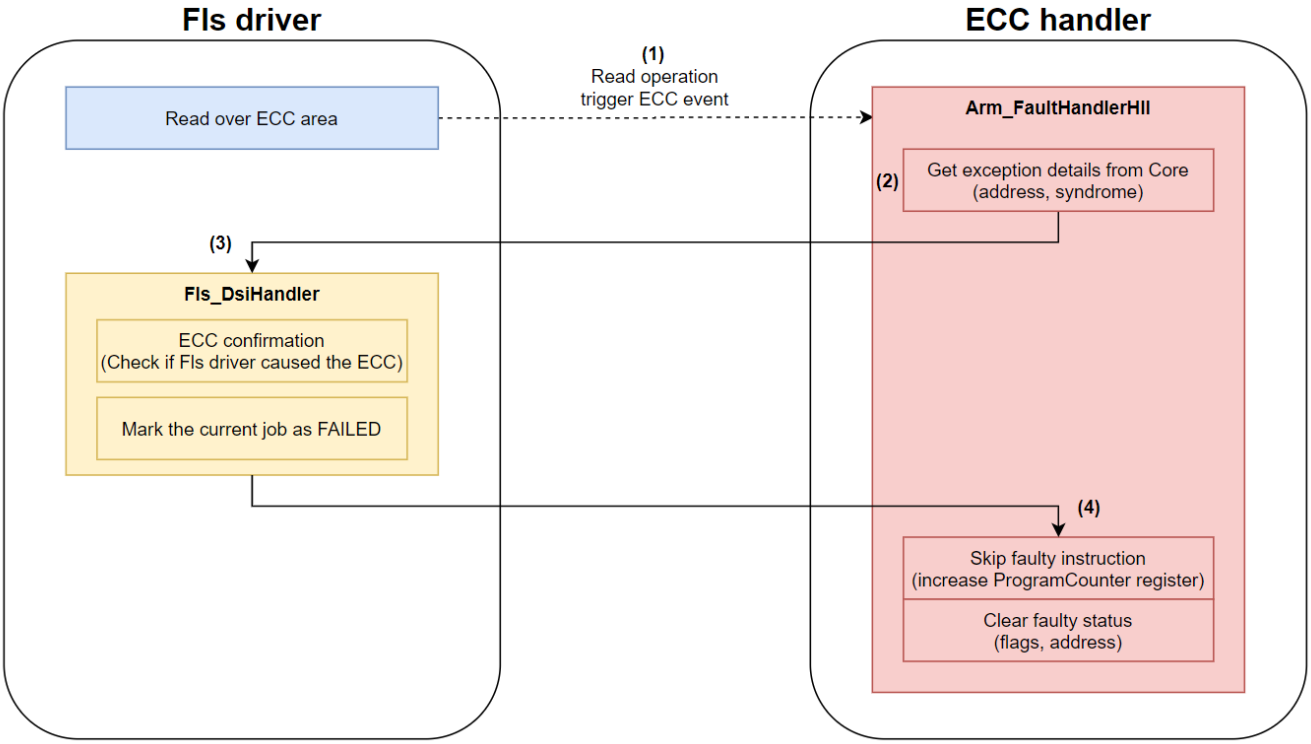
- The **Fls_ExceptionDetailsType** data structure contains some information about the details of Exception and in particular:
  - The pointer to the instruction that generated the ECC (Fls_InstructionAddressType)
  - The data address that caused the ECC error (Fls_DataAddressType)
  - Details on the 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
  - 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**)
- In addition to this information, a basic flow and an implementation idea for the fault handler is depicted below.



- Below is the actual flow of ECC handling on the ARM cortex M4 and M7:



1. When Fls driver read/compare over an ECC erase, an ECC exception occurs, the **HardFaultHandler** (**Ecc↩ Handler**) will be raised
2. **HardFaultHandler** gets some information about the exception:
   - The instruction that generated ECC from **PC** register
   - The data address that caused ECC from **BFAR** register
   - The exception syndrome from **CFSR** register
   - Then passes them to the function **Fls_DsiHandler**
3. **Fls_DsiHandler** examines the job executing by the driver and the exception information
   - If the exception caused by the current job of Fls driver, it will mark the job as FAILED and return **FLS_HANDLED_SKIP**
   - If not, it will do nothing and return **FLS_UNHANDLED**
4. Based evaluation result, the following recovery strategies may be implemented by the **EccHandler**
   - Skip the instruction that caused the ECC
   - Perform a controlled shutdown of current activity
   - Do nothing (infinite loop), etc

- File: Vector_core.s

```
.section  ".intc_vector","ax"
.align 2
.thumb
.globl undefined_handler
```

```
.globl undefined_handler
.globl VTABLE
.globl ___Stack_start__c0          /* Top of Stack for Initial Stack Pointer */
.globl Reset_Handler               /* Reset Handler */
.globl NMI_Handler                 /* NMI Handler */
.globl Arm_FaultHandlerThumb       /* Hard Fault Handler */
.globl MemManage_Handler           /* Reserved */
.globl Arm_FaultHandlerThumb       /* Bus Fault Handler */
.globl UsageFault_Handler          /* Usage Fault Handler */
.globl SVC_Handler                 /* SVCall Handler */
.globl DebugMon_Handler            /* Debug Monitor Handler */
.globl PendSV_Handler              /* PendSV Handler */
.globl SysTick_Handler             /* SysTick Handler */ /* 15*/
```

- File: Arm_FaultHandlerThumb.s

```
.globl  Arm_FaultHandlerThumb
/*
Step 1: Detect if application is using MSP or PSP
Step 2: Pointer in r0 is provided as an parameter (Arm_HardFaultHandlerHll)to the HLL function
*/
Arm_FaultHandlerThumb:
mov     r0,r14              /* r0 = EXC_RETURN; */
mov     r1,#0x4             /* r1 = 0x4; */
and     r0,r1               /* EXC_RETURN & 0x4; */
beq     label_msp_stack     /* if (EXC_RETURN & 0x4) r0=PSP else r0=MSP; */
mrs     r0,PSP              /* r0 = PSP; (PSP stack used) */
b       label_end_stack
label_msp_stack:
mrs     r0,MSP              /* r0 = MSP; (MSP stack used) */
label_end_stack:
/* Pointer in r0 is provided as an parameter to the HLL function */
add r0,#0x18
/* NOTE: HLL function is called by pure branch (b) without link (bl).
This will cause that, upon HLL exiting, the execution will continue directly
from the location pointed by the address provided in r0. */
LDR   R3,=Arm_HardFaultHandlerHll
bx    R3
```

- File: Arm_FaultHandlerHll.c

```
#define BFAR_ADDR 0xE000ED38
#define CFSR_ADDR 0xE000ED28
void Arm_HardFaultHandlerHll(Fls_InstructionAddressType * instr_pt2pt)
```

```
{
Fls_ExceptionDetailsType excDetails;
Fls_CompHandlerReturnType specificHandlerResult;
/* The instruction opcode(or the first 16 bits) value, stored in memory,
for the instruction which caused the fault
*/
uint16 instrOpcode;
/* Size of the instruction opcode stored in memory, 2 or 4 bytes */
uint8 thumbInstrSize;
Fls_InstructionAddressType instr_pt = * instr_pt2pt;
Fls_DataAddressType data_pt = (void const *)(*((uint32*)BFAR_ADDR));
uint32 syndrome = *((uint32*)CFSR_ADDR);
/* Compute the instruction opcode size for the instruction which caused the hardfault.
The value will be used to compute the address of the following instruction
*/
instrOpcode = *((uint16*)(*instr_pt2pt));
/* Compute the size of the instruction which caused the fault */
if (((instrOpcode & 0xE800) == 0xE800) ||   /* 0b11101x... */
((instrOpcode & 0xF000) == 0xF000) ||   /* 0b11110x... */
((instrOpcode & 0xF800) == 0xF800))     /* 0b11111x... */
{
/* Instruction size is 32 bits, 4 bytes */
thumbInstrSize = 4;
}
else
{
/* Instruction size is 16 bits, 2 bytes */
thumbInstrSize = 2;
}
excDetails.instruction_pt = instr_pt;
excDetails.data_pt        = data_pt;
excDetails.syndrome_u32   = syndrome;
specificHandlerResult = Fls_DsiHandler( &excDetails );
switch(specificHandlerResult)
{
case FLS_HANDLED_SKIP:
{
/* exception was handled by one of the functions called above,
continue execution, skipping the causing instruction
In the test code we assume that the exception was caused by 16-bit/32-bit
load Thumb instruction => increment return address by the size of the instruction */
```

```
instr_pt2pt = instr_pt + thumbInstrSize;
/* clear the flags and address register */
*((volatile uint32*)CFSR_ADDR) = *((volatile uint32*)CFSR_ADDR);
*((uint32*)BFAR_ADDR) = 0x0;
}
break;
case FLS_HANDLED_RETRY:
/* exception was handled by one of the functions called above,
Continue exceution, retrying the causing instruction
Thus, we don't need to modify instr_pt
*/
break;
case FLS_UNHANDLED:
/* special handling: try to store some info that nobody handled this exception
Then, try to shut-down in a controlled way. For this purpose we just fall through
*/
case FLS_HANDLED_STOP:
/* Try to shut-down in a controlled way */
/* If there's no chance to shut down in a controlled way, just fall through... */
default:
/* unexpected return - we end in an endless loop */
for (;;);
}
}
```

8.1.2.1.2 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.

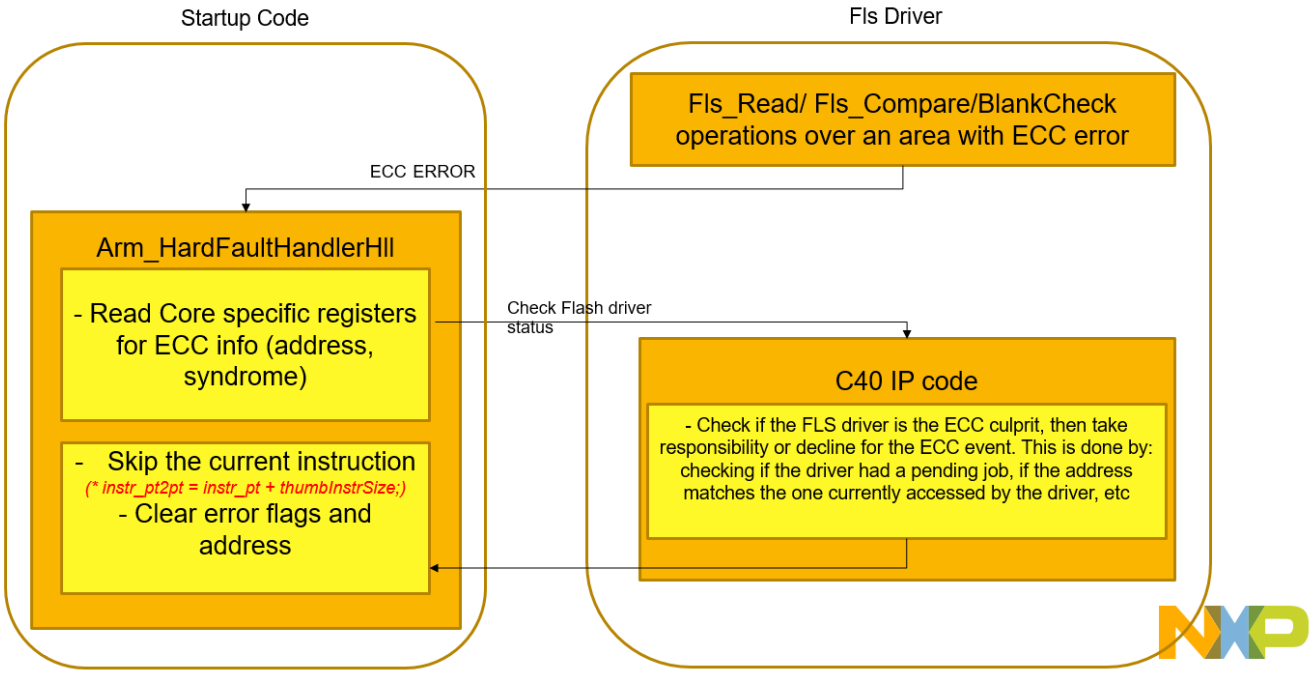
- The solution proposed here is to delegate the part of the operation that might cause an exception to a small task that can be terminated. The main task detects the termination due to exception and performs the recovery actions.

- How it works:

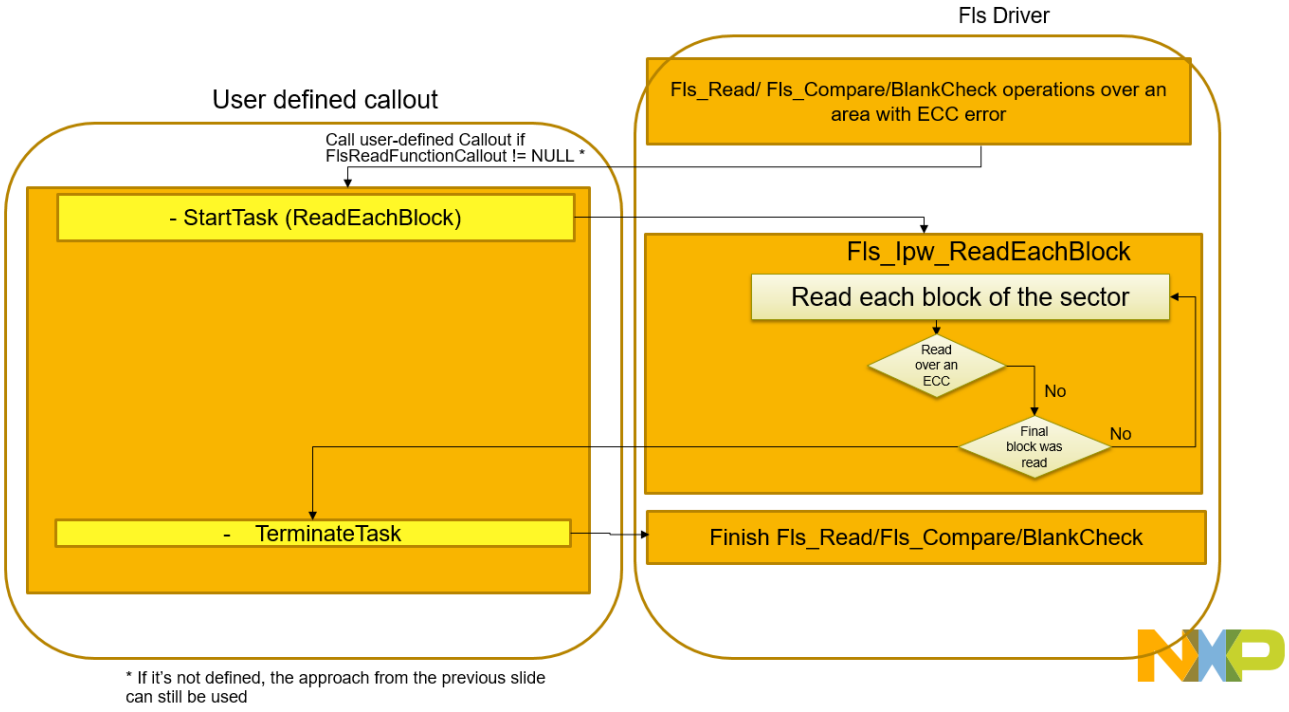
  – CalloutReadFunction (defined by the user): Where user can schedule a task which performs the copy operation (the caller) Care is taken to record the unit that is about to be copied, before the copying starts. If an exception occurs, the task can be terminated and the driver task can attempt to recover.

  – 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)

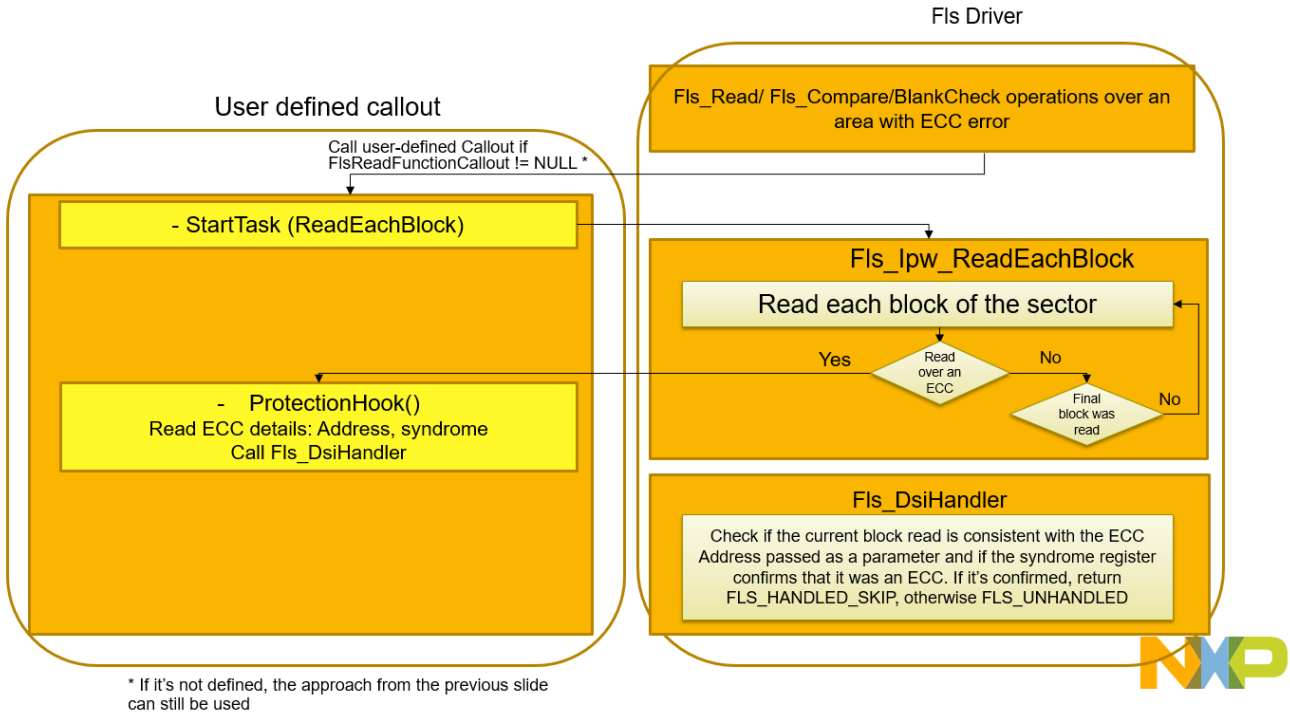
- Previous ECC approach:



- New ECC handling approach (No ECC case):



- New ECC handling approach (ECC case):



8.1.2.2 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.

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 FLS 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 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.
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
EA_RTD_00113	When RTD drivers are integrated with AutosarOS and User mode support is enabled, the integrator shall assure that the definition and declaration of all RTD functions needed to be called as trusted functions follow the naming convention Call<Function_Name>TRUSTED(parameter1,parameter2,...) in Integration/User code. They need to visible in Os.h for the driver to call them. They will call RTD <Function_Name>() as trusted functions in OS specific manner.

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

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support