Integration Manual

for S32R27X-37X FLS Driver

Document Number: IM47FLSASR4.2 Rev002R2.0.1
Rev. 1.0

Contents

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

Chapter 6
Main API Requirements

Chapter 7
Memory Allocation

Chapter 8
Configuration parameters considerations

Chapter 9
Integration Steps

Chapter 10
External Assumptions for FLS driver

Chapter 11
ISR Reference

Chapter 12
Symbolic Names DISCLAIMER

Chapter 1
Revision History

Table 1-1. Revision History

Revision	Date	Author	Description
1.0	26/01/2018	NXP MCAL Team	Updated version for ASR 4.2.2S32R27X-37X2.0.1 Release

Chapter 2
Introduction

This integration manual describes the integration requirements for FLS Driver for S32R27X-37X microcontrollers.

2.1 Supported Derivatives

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

Table 2-1. S32R27X-37X Derivatives

NXP Semiconductor	s32r274_mapbga257, s32r372_mapbga257, s32r372_mapbga141,yardxxx_mapbga257

All of the above microcontroller devices are collectively named as S32R27X-37X .

2.2 Overview

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

AUTOSAR

- paves the way for innovative electronic systems that further improve performance, safety and environmental friendliness.
- is a strong global partnership that creates one common standard: "Cooperate on standards, compete on implementation".

- is a key enabling technology to manage the growing electrics/electronics complexity. It aims to be prepared for the upcoming technologies and to improve cost-efficiency without making any compromise with respect to quality.
- facilitates the exchange and update of software and hardware over the service life of the vehicle.

2.3 About this Manual

This Technical Reference employs the following typographical conventions:

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

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

Notes and warnings are shown as below:

Note

This is a note.

2.4 Acronyms and Definitions

Table 2-2. Acronyms and Definitions

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

2.5 Reference List

Table 2-3. Reference List

#	Title	Version
1	Specification of FLS Driver	AUTOSAR Release 4.2.2
2	S32R274 Reference Manual	Rev 3 04/2017
3	S32R372 Reference Manual	Rev 2 09/2017
4	S32R274_1N58R Mask Set Errata (1N58R)	Rev.2
5	S32R274_2N58R Mask Set Errata (2N58R)	Rev.1
6	S32R372_0N36U Mask Set Errata (0N36U)	Rev. 1

Chapter 3
Building the Driver

This section describes the source files and various compilers, linker options used for building the Autosar FLS driver for NXP Semiconductor S32R27X-37X . It also explains the EB Tresos Studio plugin setup procedure.

3.1 Build Options

The FLS driver files are compiled using

- Green Hills Multi 6.1.6 / Compiler 2015.1.6
- Windriver DIAB DIAB_5_9_6_2-FCS_20161109_185532

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

Note

The TS_T2D47M20I1R0 plugin name is composed as follow:

TS_T = Target_Id

D = Derivative_Id

M = SW_Version_Major

I = SW_Version_Minor

R = Revision

(i.e. Target_Id = 2 identifies PA architecture and Derivative_Id = 47 identifies the S32R274)

3.1.1 GHS Compiler/Linker/Assembler Options

Table 3-1. Compiler Options

Option	Description
-cpu=ppc5748gz4204	Selects target processor: ppc5748gz4204
-cpu=ppc5748gz210	Selects target processor: ppc5748gz210
-ansi	Specifies ANSI C with extensions. This mode extends the ANSI X3.159-1989 standard with certain useful and compatible constructs.
-noSPE	Disables the use of SPE and vector floating point instructions by the compiler.
-Ospace	Optimize for size.
-sda=0	Enables the Small Data Area optimization with a threshold of 0.
-vle	Enables VLE code generation
-dual_debug	Enables the generation of DWARF, COFF, or BSD debugging information in the object file
-G	Generates source level debugging information and allows procedure call from debugger's command line.
--no_exceptions	Disables support for exception handling
-Wundef	Generates warnings for undefined symbols in preprocessor expressions
-Wimplicit-int	Issues a warning if the return type of a function is not declared before it is called
-Wshadow	Issues a warning if the declaration of a local variable shadows the declaration of a variable of the same name declared at the global scope, or at an outer scope
-Wtrigraphs	Issues a warning for any use of trigraphs
--prototype_errors	Generates errors when functions referenced or called have no prototype
--incorrect_pragma_warnings	Valid #pragma directives with wrong syntax are treated as warnings
-noslashcomment	C++ like comments will generate a compilation error
-preprocess_assembly_files	Preprocesses assembly files
-nostartfile	Do not use Start files
-c	Produces an object file (called input-file.o) for each source file.
--diag_error 223	Sets the specified compiler diagnostic messages to the level of error
--short_enum	Store enumerations in the smallest possible type
-DAUTOSAR_OS_NOT_USED	-D defines a preprocessor symbol and optionally can set it to a value. AUTOSAR_OS_NOT_USED: By default in the package, the drivers are compiled to be used without Autosar OS. If the drivers are used with Autosar OS, the compiler option '-DAUTOSAR_OS_NOT_USED' must be removed from project options
-DGHS	-D defines a preprocessor symbol and optionally can set it to a value. This one defines the GHS preprocessor symbol.
-DEU_DISABLE_ANSILIB_CALLS	-D defines a preprocessor symbol and optionally can set it to a value. This one defines the EU_DISABLE_ANSILIB_CALLS preprocessor symbol.

Table 3-2. Assembler Options

Option	Description
-cpu=ppc5748gz4204	Selects target processor: ppc5748gz4204
-cpu=ppc5748gz210	Selects target processor: ppc5748gz210

Table continues on the next page...

Table 3-2. Assembler Options (continued)

Option	Description
-G	Generates source level debugging information and allows procedure call from debugger's command line.
-list	Creates a listing by using the name of the object file with the .lst extension

Table 3-3. Linker Options

Option	Description
-cpu=ppc5748gz4204	Selects target processor: ppc5748gz4204
-cpu=ppc5748gz210	Selects target processor: ppc5748gz210
-nostartfiles	Do not use Start files.
-vle	Enables VLE code generation
--nocpp	Do not Generate Constructors/Destructors
-Mn	sort numerically the MAP file
-delete	The -delete option instructs the linker to remove functions that are not referenced in the final executable.
-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.
-keepmap	keeps the MAP file in case of link error

3.1.2 DIAB Compiler/Linker/Assembler Options

Table 3-4. Compiler Options

Option	Description
-tPPCE200Z4204N3VEN:simple	Sets target processor to PPCE200Z4204N3V, generates ELF using EABI conventions, No floating point support (minimizes the required runtime), selects simple environment settings for Startup Module and Libraries.
-tPPCE200Z210N3VEN:simple	Sets target processor to PPCE200Z210N3, generates ELF using EABI conventions, No floating point support (minimizes the required runtime), selects simple environment settings for Startup Module and Libraries.
-Xdialect-ansi	Follow the ANSI C standard with some additions
-XO	Enables extra optimizations to produce highly optimized code
-g3	Generate symbolic debugger information and do all optimizations.
-Xsize-opt	Optimize for size rather than speed when there is a choice
-Xsmall-data=0	Set Size Limit for 'small data' Variables to zero.
-Xsmall-const=0	Set Size Limit for 'small const' Variables to zero.
-Xaddr-sconst=0x11	Specify addressing for constant static and global variables with size less than or equal to -Xsmall-const to far-absolute.
-Xaddr-sdata=0x11	Specify addressing for non-constant static and global variables with size less than or equal to -Xsmall-data in size to far-absolute.

Table continues on the next page...

Table 3-4. Compiler Options (continued)

Option	Description
-Xno-common	Disable use of the 'COMMON' feature so that the compiler or assembler will allocate each uninitialized public variable in the .bss section for the module defining it, and the linker will require exactly one definition of each public variable
-Xnested-interrupts	Allow nested interrupts
-Xdebug-dwarf2	Generate symbolic debug information in dwarf2 format
-Xdebug-local-all	Force generation of type information for all local variables
-Xdebug-local-cie	Create common information entry per module
-Xdebug-struct-all	Force generation of type information for all typedefs, struct, union and class types
-Xforce-declarations	Generates warnings if a function is used without a previous declaration
-ee1481	Generate an error when the function was used before it has been declared
-Xmacro-undefined-warn	Generates a warning when an undefined macro name occurs in a #if preprocessor directive
-Xlink-time-lint	Enable the checking of object and function declarations across compilation units, as well as the consistency of compiler options used to compile source files
-W:as:,-l	Pass the option '-l' (lower case letter L) to the assembler to get an assembler listing file
-Wa,-Xisa-vle	Instruct the assembler to expect and assemble VLE (Variable Length Encoding) instructions rather than BookE instructions.
-c	Produces an object file (called input-file.o) for each source file.
-DAUTOSAR_OS_NOT_USED	-D defines a preprocessor symbol and optionally can set it to a value. AUTOSAR_OS_NOT_USED: By default in the package, the drivers are compiled to be used without Autosar OS. If the drivers are used with Autosar OS, the compiler option '-DAUTOSAR_OS_NOT_USED' must be removed from project options
-DDIAB	-D defines a preprocessor symbol and optionally can set it to a value. This one defines the DIAB preprocessor symbol.
-DEU_DISABLE_ANSILIB_CALLS	-D defines a preprocessor symbol and optionally can set it to a value. This one defines the EU_DISABLE_ANSILIB_CALLS preprocessor symbol.

Table 3-5. Assembler Options

Option	Description
-tPPCE200Z4204N3VEN:simple	Sets target processor to PPCE200Z4204N3V, generates ELF using EABI conventions, No floating point support (minimizes the required runtime), selects simple environment settings for Startup Module and Libraries.
-tPPCE200Z210N3VEN:simple	Sets target processor to PPCE200Z210N3, generates ELF using EABI conventions, No floating point support (minimizes the required runtime), selects simple environment settings for Startup Module and Libraries.
-g	Dump the symbols in the global symbol table in each archive file.
-Xisa-vle	Expect and assemble VLE (Variable Length Encoding) instructions rather than Book E instructions. The default code section is named .text_vle instead of .text, and the default code section fill "character" is set to 0x44444444 instead of 0. The .text_vle code section will have ELF section header flags marking it as VLE code, not Book E code.
-Xasm-debug-on	Generate debug line and file information
-Xdebug-dwarf2	Generate symbolic debug information in dwarf2 format
-Xsemi-is-newline	Treat the semicolon (;) as a statement separator instead of a comment character.

Table 3-6. Linker Options

Option	Description
-tPPCE200Z4204N3VEN:simple	Sets target processor to PPCE200Z4204N3V, generates ELF using EABI conventions, No floating point support (minimizes the required runtime), selects simple environment settings for Startup Module and Libraries.
-tPPCE200Z210N3VEN:simple	Sets target processor to PPCE200Z210N3, generates ELF using EABI conventions, No floating point support (minimizes the required runtime), selects simple environment settings for Startup Module and Libraries.
-Xelf	Generates ELF object format for output file
-m6	Generates a detailed link map and cross reference table
-Xlink-time-lint	Enable the checking of object and function declarations across compilation units, as well as the consistency of compiler options used to compile source files

3.2 Files required for Compilation

This section describes the include files required to compile, assemble (if assembler code) and link the FLS driver for S32R27X-37X microcontrollers.

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

FLS Files

- ..\Fls_TS_T2D47M20I1R0\src\Fls.c
- ..\Fls_TS_T2D47M20I1R0\src\Fls_Ac.c
- ..\Fls_TS_T2D47M20I1R0\include\Fls.h
- ..\Fls_TS_T2D47M20I1R0\include\Fls_Api.h
- ..\Fls_TS_T2D47M20I1R0\include\Fls_InternalTypes.h
- ..\Fls_TS_T2D47M20I1R0\include\Fls_Types.h
- ..\Fls_TS_T2D47M20I1R0\include\Fls_Version.h
- ..\Fls_TS_T2D47M20I1R0\include\Reg_eSys_FLASHC.h

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

Other includes files:

Files from MemIf folder:

- ..\MemIf_TS_T2D47M20I1R0\include\MemIf_Types.h

Files from Base common folder

- ..\Base_TS_T2D47M20I1R0\include\Compiler.h
- ..\Base_TS_T2D47M20I1R0\include\Compiler_Cfg.h
- ..\Base_TS_T2D47M20I1R0\include\ComStack_Types.h
- ..\Base_TS_T2D47M20I1R0\include\MemMap.h
- ..\Base_TS_T2D47M20I1R0\include\Mcal.h
- ..\Base_TS_T2D47M20I1R0\include\Platform_Types.h
- ..\Base_TS_T2D47M20I1R0\include\Std_Types.h
- ..\Base_TS_T2D47M20I1R0\include\Reg_eSys.h
- ..\Base_TS_T2D47M20I1R0\include\Soc_Ips.h
- ..\Base_TS_T2D47M20I1R0\include\RegLockMacros.h
- ..\Base_TS_T2D47M20I1R0\include\SilRegMacros.h

Files from Dem folder:

- ..\Dem_TS_T2D47M20I1R0\include\Dem.h

Files from Det folder:

- ..\Det_TS_T2D47M20I1R0\include\Det.h

Files from RTE folder:

- ..\Rte_TS_T2D47M20I1R0\include\SchM_Fls.h

3.3 Setting up the Plug-ins

The FLS driver was designed to be configured by using the EB Tresos Studio (version EB tresos Studio 21.0.0 b160607-0933 or later.)

- VSMD (Vendor Specific Module Definition) file in EB tresos Studio XDM format: .. \Fls_TS_T2D47M20I1R0\config\Fls.xdm
- VSMD (Vendor Specific Module Definition) file in AUTOSAR compliant EPD format: ..\Fls_TS_T2D47M20I1R0\autosar\(one EPD file for each supported sub-derivative)
- Code Generation Templates for Post-Build time configuration parameters:
  - ..\Fls_TS_T2D47M20I1R0\generate_PB\src\Fls_PBcfg.c
- Code Generation Templates for Pre-Compile time configuration parameters:
  - ..\Fls_TS_T2D47M20I1R0\generate_PC\include\Fls_Cfg.h
  - ..\Fls_TS_T2D47M20I1R0\generate_PC\src\Fls_Cfg.c

Steps to generate the configuration:

1. Copy the module folders Fls_TS_T2D47M20I1R0 , Base_TS_T2D47M20I1R0 and Resource_ TS_T2D47M20I1R0 and Dem_ TS_T2D47M20I1R0 and Det_ TS_T2D47M20I1R0 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

None.

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 unlocked for a successful operation. It is also handled by Fls driver, but it should be configured using Fls_PhysicalSectorUnlock parameter for all configured blocks.

Note: if the unlock is not handled by Fls driver must be setup on an application level. Example of unlock code can be found in Fls driver UM document.

4.2 Function Calls during Shutdown

None.

4.3 Function Calls during Wake-up

None.

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

When reading a FLASH location with a non correctable ECC error an IVOR exception is thrown.

On Z0, Z3 and Z6 core based platforms, a different IVOR is thrown according to the value of the bits ME and EE in the MSR (Machine Check Register) as show in the table below:

Table 4-1. IVOR selection on Z0, Z3 and Z6 core based platform

MSR[ME]	MSR[EE]	data/instruction	IVOR	Function to call
0	0	X	N/A	No IVOR thrown: Machine Checkstop state is entered (TO BE AVOIDED)
1	1	flash data read	IVOR2 (DSI)	Fls_DsiHandler
1	0	flash data read	IVOR1 (MCI)	Fls_MciHandler

On Z4 and Z7 core based platform, IVOR1 is thrown independently from MSR bits state as show in the table below.

Table 4-2. IVOR selection on Z4 and Z7 core based platform

MSR[ME]	MSR[EE]	data/instruction	IVOR	Function to call
0	0	X	N/A	Machine check stop not implemented (can cause infinite loop, TO BE AVOIDED)
1	0, 1	flash data read	IVOR1	Fls_DsiHandler

The Flash driver provide two API which can be called inside the user ECC error recovery handler (IVOR handler driver functions):

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

For Z0, Z3 and Z6 both functions should be used as explained on above tables, while for Z4 and Z7 core only **Fls_DsiHandler()**must be used.

Those API's are available only if the configuration parameter **FlsDsiHandlerApi = true**;

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

- a pointer to the statement that generated the ECC (**Fls_InstructionAddressType**);
- an address of the data that caused the error in ECC (**Fls_DataAddressType**);
- details on the type of exception (**uint32**).

IVOR Hander Driver functions examine 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 DSI (or MCI) reason;
- Data address which caused the exception matches the address currently accessed by pending flash read or flash compare job or it is in the same cache line.

If these conditions are verified the IVOR handler driver functions return **FLS_HANDLED_SKIP** and sets 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**);

On PowerPC platforms, if data cache is enabled and even if the flash region is configured as cache-inhibited in the memory protection unit, a cache data line fill to the flash memory may be triggered by the driver`s load instruction. The read data will not be marked as valid in the cache, but the cache fetch might trigger an ECC exception if there is an ECC affected location inside the same cache line as the location read by the driver. For example: if the flash driver accesses for read location 0x10000000, and there is an ECC affected location at address 0x10000008, an ECC event might be triggered by a cache line fill. To cover this behavior, the Fls_DsiHandler will return FLS_HANDLED_SKIP and report a job as failed if there is any ECC event in the same cache line as the currently accessed memory location. **Note:**To maintain the robustness level in the FEE driver, the "Fee_VirtualPageSize" should be configured according to the cache line size (32 bytes).

If the ECC exception is influenced by a cache fetch, depending on the position of the location read by driver and the position of the ECC affected location inside the cache line, the triggered exception might be synchronous or asynchronous. If the exception is synchronous (LD bit of the syndrome is set) the value reported in MCSRR0 represents the address of the instruction which caused the exception, and should be skipped if return to driver code is needed. If the exception is asynchronous (LD bit of the syndrome is cleared) the value reported in MCSRR0 represents the address of the instruction which would have executed next, had the exception not appeared, thus it would have to be executed.

IVOR handler driver functions are not available for ISI (Instruction Storage Interrupt).

In addition to this information, a basic flow is depicted in the following figure:

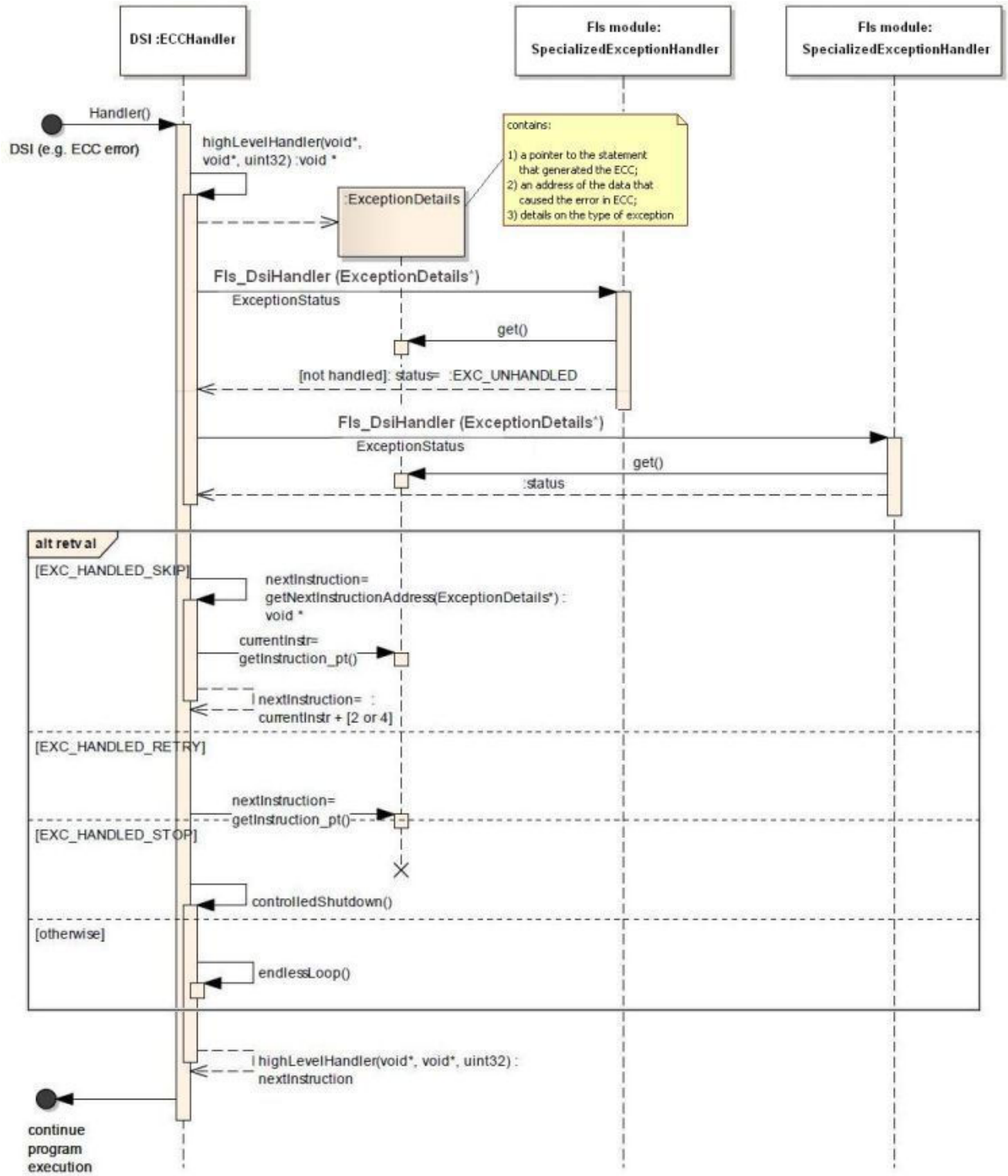

Figure 4-1. Example of ECC Handling

4.5 ECC Management on Data Flash

Since reading the ECC-affected data from the Data flash segments (named FLS_DATA_...) never leads to the exception (unlike to the reading the ECCs from the Code flash), the dedicated mechanism was implemented to handle the ECCs in the Data flash. This mechanism has been integrated into the existing exception management so that by using the same configuration parameter "FlsDsiHandlerApi" it is possible to enable the ECC handling for both Code and Data Flash.

If the uncorrectable ECC error is detected during the read from the Data Flash, the EER bit of the MCR register and the ADR register are updated to notify about the ECC event (if it is present in the Code Flash additionally an IVOR exception is triggered). Read and Compare FLS jobs handle such a situation in the following way:

Any read from the flash memory is preceded by the entering the critical section (FLS_EXCLUSIVE_AREA_00) and clearing the EER bit and succeeded by the EER reading and exiting the critical section. If the EER bit is set an ECC error occurred during any flash read performed within the critical section. In such a case the read operation is reported as failed to the upper SW layers regardless of the address connected with the ECC error (i.e. address provided by the ADR register). It means that the read operation might fail also in the case when the EER bit was set due to another activity (i.e. DMA transfer, activity of another core, instruction fetch, etc.) - such a read cannot be trusted since the information about the potential ECC failure was lost. Since the read from the flash and reading the EER bit is always done within one single critical section, it is a user responsibility to prevent the reads which might affect the EER bit (if possible).

Note: The embedded flash memory is addressable by page (256 bits - 4 double words) for read operation. In other words, even if the ECC error in the Data flash is present in 1 double word only, it will be reported for any address inside this page (4 double word area). To maintain the robustness level in the FEE driver, the "Fee_VirtualPageSize" should be configured according to the page size (32 bytes).

Chapter 5
Module requirements

5.1 Exclusive areas to be defined in BSW scheduler

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

FLS driver has five exclusive areas (EA) FLS_EXCLUSIVE_AREA_10, FLS_EXCLUSIVE_AREA_11, FLS_EXCLUSIVE_AREA_12, FLS_EXCLUSIVE_AREA_13 and FLS_EXCLUSIVE_AREA_14. The purpose of these exclusive areas is to make the functions Fls_Erase, Fls_Write, Fls_Read, Fls_Compare and Fls_BlankCheck thread safe and thus protect FLS internal job variables.

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_10 Used in Fls_Erase.

FLS_EXCLUSIVE_AREA_11 Used in Fls_Write.

FLS_EXCLUSIVE_AREA_12 Used in Fls_Read.

FLS_EXCLUSIVE_AREA_13 Used in Fls_Compare.

FLS_EXCLUSIVE_AREA_14 Used in Fls_BlankCheck.

Table 5-1. Exclusive Areas

	FLS_EXCLUSIVE_AREA_10	FLS_EXCLUSIVE_AREA_11	FLS_EXCLUSIVE_AREA_12	FLS_EXCLUSIVE_AREA_13	FLS_EXCLUSIVE_AREA_14
FLS_EXCLUSIVE_AREA_10		x	x	x	x
FLS_EXCLUSIVE_AREA_11	x		X	X	X
FLS_EXCLUSIVE_AREA_12	x	x		X	X
FLS_EXCLUSIVE_AREA_13	x	x	X		X
FLS_EXCLUSIVE_AREA_14	x	x	X	X	

5.1.2 Flash access notifications

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

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

Note: These notifications are the result of unclear Fls SWS document requirement number FLS215.

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

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

area or if there is no need to exclusively link an ECC error to the flash driver read, then the implementation could be ‘void’ also. In all other cases you have to block the execution of the code (task-s) which would access this affected Flash area (sector-s).

5.2 Peripheral Hardware Requirements

The FLS driver uses/controls the "Flash Memory" MCU peripheral. For more details about peripheral and its structure refers to MCU reference manual.

5.3 ISR to configure within OS – dependencies

None.

5.4 ISR Macro

None.

5.5 Other AUTOSAR modules - dependencies

- **Base** This module provides basic data types and auxiliary macros or functions commonly used by other AUTOSAR modules.
- **Dem:** This module is necessary for enabling reporting of production relevant error status. The API function used is Dem_ReportErrorStatus().
- **Det** This module is necessary for enabling Development error detection. The API function used is Det_ReportError(). The activation/deactivation of Development error detection is configurable using 'FlsDevErrorDetect' configuration parameter.
- **Rte:** Exclusive areas implementations.
- **MemIf:** Memory Interface
- **Resource:** Sub-Derivative model is selected from Resource configuration.
- **MCI(Machine Check Interrupt):** (only if FlsDsiHandlerApi=true)

5.6 User Mode Support

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

Fls module will set the UAA bit in REG_PROT_GCR register for C55FMC (IPV_FLASHV2).

The following functions of Fls must be called as trusted function :

- Fls_Flash_SetUserAccessAllowed()

5.7 Data Cache Restriction

The data cache has to be inhibited over all flash sectors handled by the Flash driver.

Chapter 6
Main API Requirements

6.1 Main functions calls within Rte module

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

6.2 API Requirements

None

6.3 Calls to Notification Functions, Callbacks, Callouts

The FLS driver provides notifications that are user configurable:

- FlsAcCallback (usually routed to Wdg module)
- FlsJobEndNotification (usually routed to Fee module)
- FlsJobErrorNotification (usually routed to Fee module)
- FlsStartFlashAccessNotif (used, if needed, to mark the start of a flash read,program access)
- FlsFinishedFlashAccessNotif (used, if needed, to mark the end of a flash read,program access)

6.4 Tips for FLS integration

Synchronous vs. Asynchronous write 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.

Example1:

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

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

Example2:

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

Possible values after interrupted HW write

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

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

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

3. Always wrong value (stable ECC error).

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

Note: One FLS HW write operation writes “FlsPageSize” Bytes from higher to lower addresses. It is possible to interrupt (e.g. by reset) this job after each byte.

Example: value 00 00 00 00 shall be written to the FLS. We can reset the write operation to obtain values: FF FF FF 00, FF FF 00 00 or FF 00 00 00 (of course not all of them can be seen because of ECC error. This is just an example to explain the write operation process).

Chapter 7
Memory Allocation

7.1 Sections to be defined in MemMap.h

For Post Build data:

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



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

For Code:

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

For Variables:

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



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



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



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

For Constant data:

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

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

For Ram Code:

For position-independent Access Code:

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

7.2 Linker command file

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

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

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

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

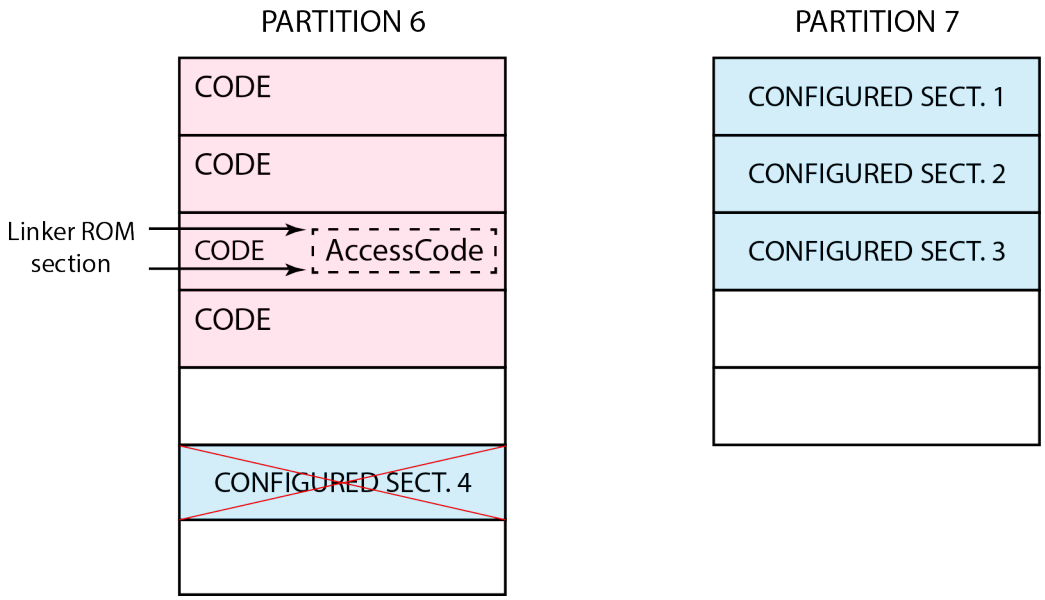

Figure 7-1. "Fls_Flash_AccessCode" function used from Flash

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

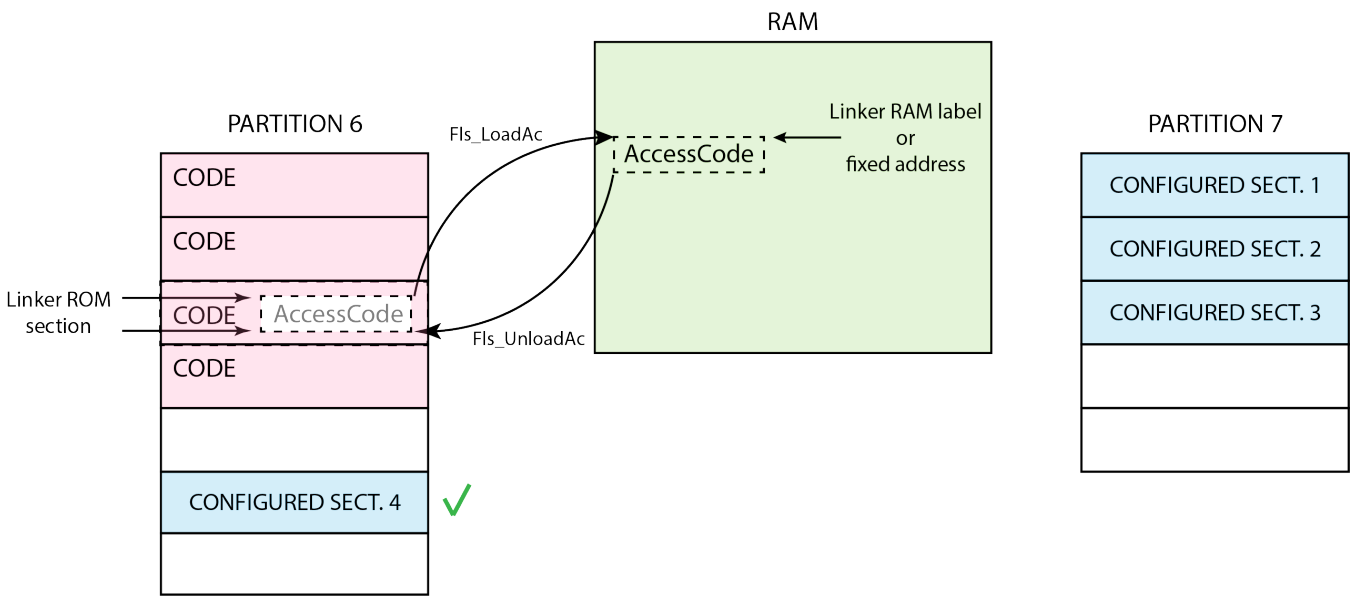

Figure 7-2. "Fls_Flash_AccessCode" function used from RAM

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

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

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

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

- FlsAcErase of space corresponding to Fls_ACEraseSize (see above)

  FlsAcWrite of space corresponding to Fls_ACWriteSize (see above)

Alternatively, using the following configuration parameters

- FlsAcErasePointer
- FlsAcWritePointer

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

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

DIAB linker command file example:

```
    ....

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

            /* ... */
    }>flash_memory

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

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

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

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

GHS linker command file example:

Linker command file

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

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

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

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

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

Chapter 8
Configuration parameters considerations

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

8.1 Configuration Parameters

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

Table 8-1. Configuration Parameters

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

Table continues on the next page...

Table 8-1. Configuration Parameters (continued)

Configuration Container	Configuration Parameters	Configuration Variant	Current Implementation
	FlsAsyncWriteTimeout	VariantPostBuild	PostBuild
	FlsAsyncEraseTimeout	VariantPostBuild	PostBuild
	FlsSyncWriteTimeout	VariantPostBuild	PostBuild
	FlsSyncEraseTimeout	VariantPostBuild	PostBuild
	FlsAbortTimeout	VariantPostBuild	PostBuild
FlsConfigSet			
	FlsAcErase	VariantPostBuild	PostBuild
	FlsAcWrite	VariantPostBuild	PostBuild
	FlsAcErasePointer	VariantPostBuild	PostBuild
	FlsAcWritePointer	VariantPostBuild	PostBuild
	FlsCallCycle	VariantPostBuild	PostBuild
	FlsDefaultMode	VariantPostBuild	PostBuild
	FlsAcCallback	VariantPostBuild	PostBuild
	FlsJobEndNotification	VariantPostBuild	PostBuild
	FlsJobErrorNotification	VariantPostBuild	PostBuild
	FlsStartFlashAccessNotif	VariantPostBuild	PostBuild
	FlsFinishedFlashAccessNotif	VariantPostBuild	PostBuild
	FlsMaxReadFastMode	VariantPostBuild	PostBuild
	FlsMaxReadNormalMode	VariantPostBuild	PostBuild
	FlsMaxWriteFastMode	VariantPostBuild	PostBuild
	FlsMaxWriteNormalMode	VariantPostBuild	PostBuild
	FlsProtection	VariantPostBuild	PostBuild
FlsSector			
	FlsSectorIndex	VariantPostBuild	PostBuild
	FlsPhysicalSectorUnlock	VariantPostBuild	PostBuild
	FlsPhysicalSector	VariantPostBuild	PostBuild
	FlsNumberOfSectors	VariantPostBuild	PreCompile
	FlsPageSize	VariantPostBuild	PreCompile
	FlsSectorSize	VariantPostBuild	PreCompile
	FlsSectorStartaddress	VariantPostBuild	PreCompile
	FlsProgrammingSize	VariantPostBuild	PostBuild
	FlsSectorEraseAsynch	VariantPostBuild	PostBuild
	FlsPageWriteAsynch	VariantPostBuild	PostBuild

Chapter 9
Integration Steps

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

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

Chapter 10
External Assumptions for FLS driver

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

[SMCAL_CPR_EXT165]

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

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

application timing. >>

[SMCAL_CPR_EXT175]

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

[SWS_Fls_00214]

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

NOTE

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

[SWS_Fls_00240]

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

NOTE

Not a FLS module requirement

[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

Not a FLS module requirement

[SWS_Fls_00260]

<< API function Description

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

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

NOTE

Not a FLS module requirement

[SWS_Fls_00261]

<< API function Description

Det_ReportError Service to report development errors. >>

NOTE

Not a FLS module requirement

[SWS_Fls_00262]

<< Service name: Fee_JobEndNotification

Syntax: void Fee_JobEndNotification(

void

)

Sync/Async: Synchronous

Reentrancy: Don't care

Parameters (in): None

Parameters (inout): None

Parameters (out): None

Return value: None

Description: This callback function is called when a job has been completed with a positive result. >>

NOTE

Configurable interface (Fee_JobEndNotification() callback).

[SWS_Fls_00263]

<< Service name: Fee_JobErrorNotification

Syntax: void Fee_JobErrorNotification(

void

)

Sync/Async: Synchronous

Reentrancy: Don't care

Parameters (in): None

Parameters (inout): None

Parameters (out): None

Return value: None

Description: This callback function is called when a job has been canceled or finished with negative result. >>

NOTE

Configurable interface (Fee_JobErrorNotification() callback).

Chapter 11
ISR Reference

ISR functions exported by the FLS driver.

11.1 Software specification

The following sections contains driver software specifications.

11.1.1 Define Reference

Constants supported by the driver are as per AUTOSAR FLS Driver software specification Version 4.2 Rev0002 .

11.1.2 Enum Reference

Enumeration of all constants supported by the driver are as per AUTOSAR FLS Driver software specification Version 4.2 Rev0002 .

11.1.3 Function Reference

Functions of all functions supported by the driver are as per AUTOSAR FLS Driver software specification Version 4.2 Rev0002 .

11.1.4 Structs Reference

Data structures supported by the driver are as per AUTOSAR FLS Driver software specification Version 4.2 Rev0002 .

11.1.5 Types Reference

Types supported by the driver are as per AUTOSAR FLS Driver software specification Version 4.2 Rev0002 .

11.1.6 Variables Reference

Variables supported by the driver are as per AUTOSAR FLS Driver software specification Version 4.2 Rev0002 .

Chapter 12
Symbolic Names DISCLAIMER

All containers having the symbolic name tag set as true in the Autosar schema will generate defines like

#define <Container_Short_Name> <Container_ID>

For this reason it is forbidden to duplicate the name of such containers across the MCAL configuration, or to use names that may trigger other compile issues (e.g. match existing #ifdefs arguments).

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number IM47FLSASR4.2 Rev002R2.0.1
Revision 1.0

AUTOSAR