Flash driver user guide

TRAVEO™ T2G family

About this document

Scope and purpose

This user guide describes the architecture, configuration, and usage of the flash driver. It helps you to understand the functionality of the driver and provides a reference for the driver's API.

The installation, build process, and general information about the use of the EB tresos Studio are not within the scope of this document. See the EB tresos Studio for ACG8 user's guide [7] for detailed information of these topics.

Intended audience

This document is intended for anyone who uses the flash driver of the TRAVEO™ T2G family.

Document structure

Chapter 1 General overview gives a brief introduction to the flash driver, explains the embedding in the AUTOSAR environment, and describes the supported hardware and development environment.

Chapter 2 Using the flash driver details the steps required to use the flash driver in your application.

Chapter 3 Structure and dependencies describes the file structure and the dependencies for the flash driver.

Chapter 4 EB tresos Studio configuration interface describes the configuration of the flash driver.

Chapter 5 Functional description gives a functional description of all services offered by the flash driver.

Chapter 6 Hardware resources gives a description of all hardware resources used.

The Appendix A and Appendix B provides a complete API reference and access register table.

Abbreviations and definitions

Table 1 Abbreviation

Abbreviation	Description
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
Basic Software (BSW)	Standardized part of software which does not fulfill a vehicle functional job.
CM0+	Arm® Cortex®-M0+ CPU core
CM4	Arm® Cortex®-M4 CPU core
CM7_0	Arm® Cortex®-M7 CPU first core
CM7_1	Arm® Cortex®-M7 CPU second core
CM7_2	Arm® Cortex®-M7 CPU third core

Abbreviation	Description
CM7_3	Arm® Cortex®-M7 CPU fourth core
Data buffer	A RAM area in memory which flash driver APIs access
DEM	Diagnostic Event Manager
DET	Default Error Tracer
DMA	Direct Memory Access
EB tresos ECU AUTOSAR Suite	A collection of AUTOSAR Basic Software modules and a Runtime Environment integrated in a common configuration and build environment.
EB tresos Studio	Elektrobit Automotive configuration framework
ECC	Error Checking Code
EEPROM	Electrically erasable programmable ROM
FLS	Flash driver module
FEE	Flash EEPROM Emulation
Flash sector	A flash sector is the smallest amount of flash memory that can be erased in one pass. The size of the flash sector depends upon the flash technology and is therefore hardware dependent.
Flash page	A flash page is the smallest amount of flash memory that can be programmed in one pass. The size of the flash page depends upon the flash technology and is therefore hardware dependent.
GHS	Green Hills Software
HSM	Hardware Security Module
HW	Hardware
IPC	Inter Processor Communication
ISR	Interrupt Service Routine
µC	Microcontroller
MCAL	Microcontroller Abstraction Layer
MCU	Microcontroller Unit
MPU	Memory Protection Unit
Non-blocking mode	Mode that does not block CM0+ while flash memory operation is running.
OS	Operating System
SchM	BSW Scheduler
S-LLD	Security Low Level Driver
Work Flash	Application flash (Flash memory for storing user's data by such as FEE)
Work flash block#0	First flash area in two separate work flash. Refer to data sheet to know the mounted devices and address mappings.
Work flash block#1	Second flash area in two separate work flash. Refer to data sheet to know the mounted devices and address mappings.

Related documents

AUTOSAR requirements and specifications

Bibliography

[1] General specification of basic software modules, AUTOSAR release 4.2.2.

[2] Specification of flash driver, AUTOSAR release 4.2.2.

[3] Specification of flash EEPROM emulation, AUTOSAR release 4.2.2.

[4] Specification of default error tracer, AUTOSAR release 4.2.2.

[5] Specification of RTE, AUTOSAR release 4.2.2.

[6] Specification of ECU configuration parameters, AUTOSAR release 4.2.2.

Elektrobit automotive documentation

Bibliography

[7] EB tresos Studio for ACG8 user's guide.

Hardware documentation

The hardware documents are listed in the delivery notes.

Related standards and norms

Bibliography

[8] Layered software architecture, AUTOSAR release 4.2.2.

Table of contents

About this document **1**
Table of contents **4**
1 General overview **8**
1.1 Introduction to the AUTOSAR flash driver 8
1.2 User profile 8
1.3 Embedding in the AUTOSAR environment 8
1.4 Supported hardware 9
1.5 Development environment 9
1.6 Character set and encoding 9
1.7 HSM support 9
2 Using the flash driver **10**
2.1 Installation and prerequisites 10
2.2 Configuring the flash driver 10
2.2.1 Architecture details 11
2.3 Adapting your application 12
2.4 Starting the build process 13
2.5 Measuring stack consumption 14
2.6 Memory mapping 14
2.6.1 Memory allocation keyword 14
3 Structure and dependencies **16**
3.1 Static files 16
3.2 Configuration files 16
3.3 Generated files 16
3.4 Dependencies 17
3.4.1 Flash EEPROM emulation (FEE) 17
3.4.2 DET 17
3.4.3 BSW scheduler 17
3.4.4 Error callout handler 17
4 EB tresos Studio configuration interface **18**
4.1 General configuration 18
4.2 Vendor specific configuration 18
4.2.1 Parameter constraints 18
4.2.1.1 Container FlsGeneral 18
4.2.1.2 Container FlsConfigSet 22
4.2.1.3 Container FlsDemEventParameterRefs 25
4.2.1.4 Container FlsExternalDriver 25
4.2.1.5 Container FlsSector 25
4.2.1.6 Container FlsPublishedInformation 26
4.2.2 Vendor and driver specific parameters 28
4.2.2.1 Container FlsGeneral 28
4.2.2.2 Container FlsConfigSet 37
4.2.2.3 Container FlsSector 39
4.2.3 Other modules 39
4.2.3.1 Flash EEPROM emulation 39
4.2.3.2 DET 39
4.2.3.3 BSW scheduler 39
5 Functional description **40**

5.1 Function of the flash driver 40
5.1.1 Flash driver state machine 40
5.1.1.1 State MEMIF_UNINIT 40
5.1.1.2 State MEMIF_IDLE 40
5.1.1.3 State MEMIF_BUSY 41
5.1.2 Flash driver job result state 41
5.1.2.1 MEMIF_JOB_OK 41
5.1.2.2 MEMIF_JOB_PENDING 41
5.1.2.3 MEMIF_JOB_CANCELED 41
5.1.2.4 MEMIF_JOB_FAILED 41
5.1.2.5 MEMIF_BLOCK_INCONSISTENT 41
5.1.3 Initialization 42
5.1.4 Reading data from the flash memory 42
5.1.5 Writing data to the flash memory 43
5.1.6 Erasing data from the flash memory 45
5.1.7 Comparing data from the flash memory 46
5.1.8 Checking blank for the flash memory 47
5.1.9 Canceling a job prior to maturity 48
5.1.10 Retrieving the status information 48
5.1.11 Setting the driver operation mode 48
5.1.12 Suspending a job 49
5.1.13 Resuming a suspended job 49
5.1.14 Timeout supervision 50
5.1.15 eCT flash safety mechanism 50
5.1.15.1 Related configurations 50
5.1.15.2 IPC lock acquisition and release 51
5.1.15.3 Arbitration sequences 52
5.1.15.4 Assumptions of use 57
5.1.15.5 Limitations 57
5.2 Virtual flash memory layout 57
5.3 Parallel flash operations for separate work flash memories 57
5.4 Default error detection 58
5.5 Runtime error detection 59
5.6 Reentrancy 59
5.7 Debugging support 59
6 Hardware resources 60
6.1 Registers 60
6.2 Interrupts 60
6.3 Fault 61
6.4 IPC 62
6.5 System call 63
6.6 Memory protection unit (MPU) 64
6.7 DMA 66
7 Appendix A – API reference 67
7.1 Data types 67
7.1.1 Flash driver data types 67
7.1.1.1 Fls_AddressType 67
7.1.1.2 Fls_LengthType 67
7.1.1.3 Fls_ConfigType 67

7.1.1.4 External data types 67
7.1.1.5 Std_ReturnType 67
7.1.1.6 Std_VersionInfoType 67
7.1.1.7 MemIf_ModeType 68
7.1.1.8 MemIf_StatusType 68
7.1.1.9 MemIf_JobResultType 68
7.2 Macros 68
7.2.1 Error codes 68
7.2.2 Version information 69
7.2.3 Module information 69
7.2.4 API service IDs 69
7.3 Functions 70
7.3.1 Fls_Init 70
7.3.2 Fls_Erase 71
7.3.3 Fls_Write 72
7.3.4 Fls_Cancel 73
7.3.5 Fls_GetStatus 74
7.3.6 Fls_GetJobResult 75
7.3.7 Fls_Read 76
7.3.8 Fls_Compare 77
7.3.9 Fls_SetMode 78
7.3.10 Fls_GetVersionInfo 79
7.3.11 Fls_BlankCheck 79
7.3.12 Fls_ReadImmediate 80
7.3.13 Fls_Suspend 81
7.3.14 Fls_Resume 82
7.3.15 Fls_SetCycleMode 83
7.4 Scheduled functions 84
7.4.1 Fls_MainFunction 84
7.5 Expected interfaces 85
7.5.1 Mandatory interface 85
7.5.2 Optional interfaces 85
7.5.2.1 Det_ReportError 85
7.5.2.2 Det_ReportRuntimeError 86
7.5.3 Configurable interfaces 86
7.5.3.1 Fee_JobEndNotification 87
7.5.3.2 Fee_JobErrorNotification 87
7.5.3.3 Fee_DedErrorNotification 88
7.5.3.4 Fee_SedErrorNotification 88
7.5.3.5 Systemcall callout function 89
7.5.3.6 Erase callout API 89
7.6 Required callback functions 90
7.6.1 Callout functions 90
7.6.2 Error callout API 90
8 Appendix B – Access register table 91
8.1 FLASHC 91
8.2 FLASHC_FM_CTL_ECT 93
8.3 FLASHC1 94
8.4 FLASHC1_FM_CTL_ECT 96
8.5 FAULT 99

8.6 IPC 101
8.7 CPUSS 103
8.8 M-DMA (DMAC) 103
8.9 DMAC_CH 104
Revision history 106
Disclaimer 112

1 General overview

1.1 Introduction to the AUTOSAR flash driver

The flash driver abstracts the hardware internal flash controller of the TRAVEO™ T2G microcontroller and provides API functions for writing, erasing, reading, and comparing data from or to the flash memory.

1.2 User profile

This guide presumes the reader has a basic knowledge of the following:

- Flash memory
- Embedded systems
- The AUTOSAR terminology
- The C programming language

1.3 Embedding in the AUTOSAR environment

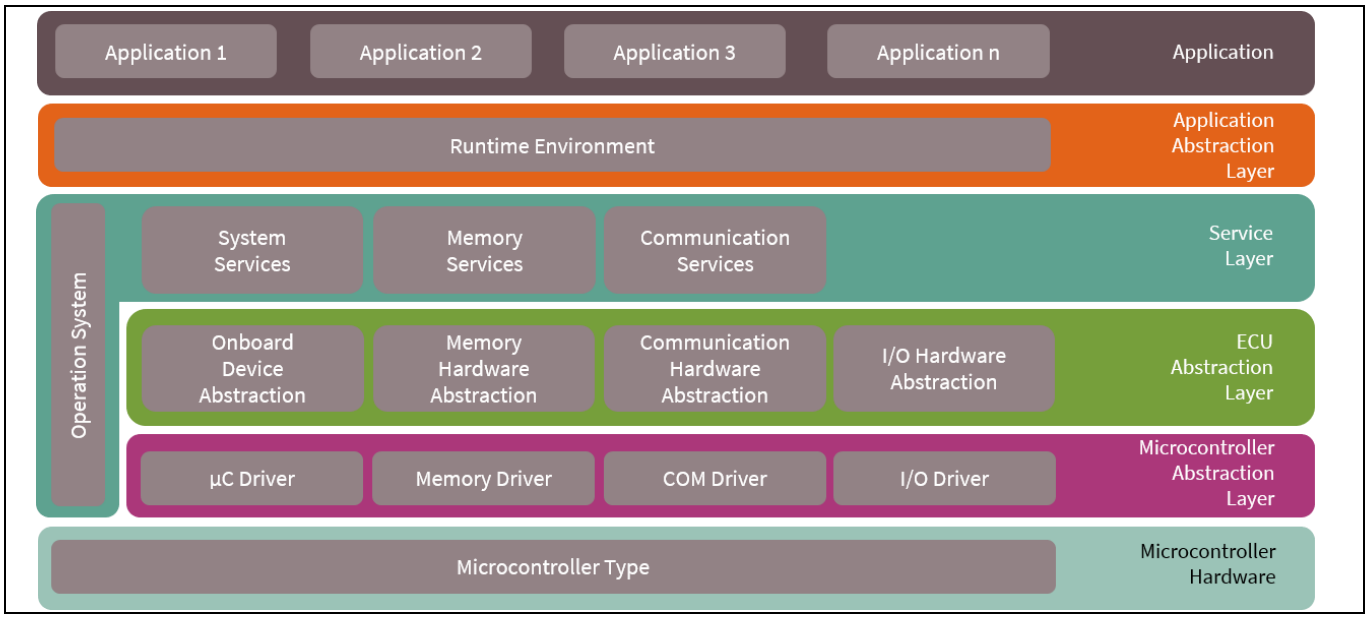

Figure 1 Overview of AUTOSAR software layers

Figure 1 shows the layered AUTOSAR software architecture. The FLS driver (Figure 2) is one of the memory drivers in the microcontroller abstraction layer (see *Layered software architecture* [8]).

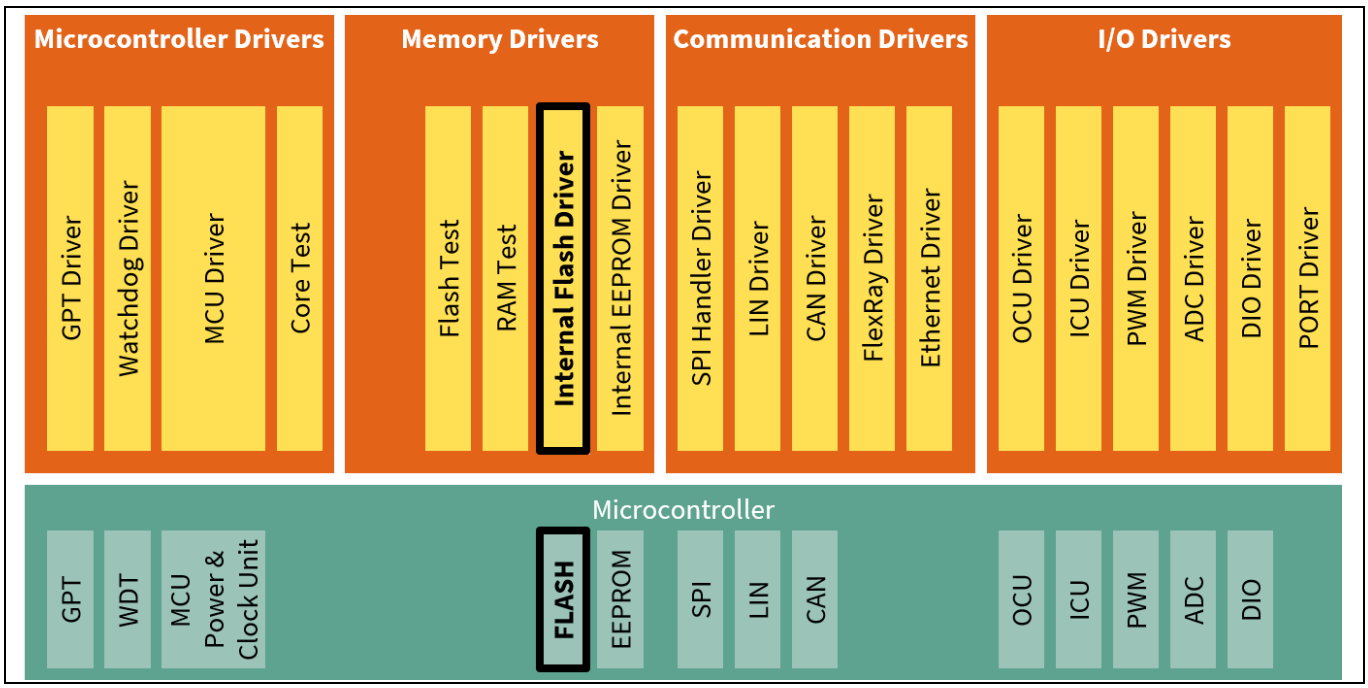

Figure 2 Flash driver in MCAL layer

1.4 Supported hardware

This version of the flash driver supports the TRAVEO™ T2G microcontroller family. No further special external hardware devices are required. The supported derivatives are listed in the release notes.

Additional derivatives that contain only a subset of the capabilities of one derivative mentioned above can be implemented or supported by providing a resource file with its properties.

1.5 Development environment

The development environment corresponds to AUTOSAR release 4.2.2. The Base, Platforms, Make, and Resource modules are required for proper functionality of the flash driver.

1.6 Character set and encoding

All source code files of the flash driver are restricted to the ASCII character set. The files are encoded in UTF-8 format, with only the 7-bit subset (values 0x00 … 0x7F) being used.

1.7 HSM support

The flash driver is provided to handle flash memory from HSM on CM0+ in addition to the driver for application. The plugin of the driver for HSM (on CM0+) is called Fls_TS_T40D13M2I0R0. Whereas, the plugin of the driver for application (on CM4, CM7_0, CM7_1, CM7_2, or CM7_3) is called Fls_TS_T40D13M1I0R0.

This document describes the common and plugin-specific features of both plugins.

2 Using the flash driver

This chapter describes the necessary steps to incorporate the flash driver into your application.

2.1 Installation and prerequisites

Note: *Before you start, see the EB tresos Studio for ACG8 user's guide [7] for the following information.*

1. *Installation procedure of EB tresos ECU AUTOSAR components*
2. *Usage of the EB tresos Studio software*
3. *Usage of the EB tresos ECU AUTOSAR build environment (it includes an explanation of how to set up and integrate your application within the EB tresos ECU AUTOSAR build environment)*

The installation of the flash driver complies with the general installation procedure for EB tresos ECU AUTOSAR components given in the documents mentioned above. If the driver is successfully installed, the driver will appear in the module list of the EB tresos Studio.

In the following sections, it is assumed that the project is properly set up and is using the application template as described in the *EB tresos Studio for ACG8 user's guide* [7]. This template provides the necessary folder structure, project and makefiles needed to configure and compile an application within the build environment. You also have to be familiar with the usage of the command line shell.

2.2 Configuring the flash driver

This section provides a short overview about the configuration structure defined by AUTOSAR to use the flash driver.

The following three basic containers are used to configure common behavior.

1. `FlsConfigSet`: Container for runtime configuration parameters of the flash driver. Implementation type: `Fls_ConfigType`.
2. `FlsGeneral`: Container for general parameters of the flash driver. These parameters are always precompiled.
3. `FlsPublishedInformation`: Container for published parameters. These parameters do not have any configuration class setting because they are published information.

For detailed information and description, see EB tresos Studio configuration interface.

Note: *Ensure that the application also includes an AUTOSAR-compliant default error tracer when default error detection and/or runtime error detection are enabled. If not, the application will not compile.*

See EB tresos Studio configuration interface for details of a configuration to be set up.

2.2.1 Architecture details

- `FlsErrorCalloutFunction`: Specifies an error callout handler, which is called when any errors are detected during runtime.
- `FlsIncludeFile`: Specifies the file name, which is used to include definitions (such as declaration for error callout handler).
- `FlsEraseVerification`, `FlsBeforeWriteVerification`, and `FlsWriteVerification`: Specifies whether each verification at writing or erasing is enabled or disabled.
- `FlsEraseCalloutFunction`: Specifies an erase callout handler, which is called when an erase job set up by `Fls_Erase()` is accepted.
- `FlsDedErrorNotification`: Specifies a DED error notification, which is called when a double-bit error (DED) is detected.
- `FlsSedErrorNotification`: Specifies a SED error notification, which is called when a single-bit error (SED) is detected.
- `FlsDmaChannel`: Specifies a DMA channel used for reading from work flash.
- `FlsAuxiliaryBufferSize`: The size of the auxiliary buffer that stores data read from work flash by DMA transfer at a time, for reading, verifying, or comparing process.
- `FlsSetFlashCtlRegister`: Specifies the bit fields of FLASH_CTL register that are set by the flash driver.
- `FlsSetWorkFlashSafetyRegister`: Specifies whether WORK_FLASH_SAFETY register is set by the flash driver.
- `FlsSetWorkFlashFaultMaskRegister`: Specifies whether the fault mask 1 (MASK1) and mask 2 (MASK2) registers for work flash are set by the flash driver.
- `FlsDefineWdgClear`: Specifies whether the function `Fls_WdgClear` (described later) to clear the watchdog timer is defined by the flash driver.
- `FlsUseNonBlockingWrite`: Specifies whether the flash driver writes to work flash in non-blocking mode. This parameter is not applied for the write operation to work flash block#1.
- `FlsUseDmaForRead`: Specifies whether the flash driver reads from work flash with DMA transfer.
- `FlsReportErrorIfNotBlank`: Specifies whether the flash driver calls the error callout function when a blank check job detects non-blank.
- `FlsUseSafetyMechanism`: Specifies whether eCT flash safety mechanism for write/erase is enabled or disabled. The mechanism is used to inform another flash driver (for application or HSM) of flash embedded (write or erase) operations or to be notified of flash embedded operations by them.
- `FlsHsmPresent`: Specifies whether the hardware security module (HSM) is present. If HSM exists, it will perform setting of important registers.
- `FlsArbitrationTimeout`: Specifies tolerant time for arbitration (waiting for) to finish the flash operation that was started from another core, typically maximum time to erase one flash sector. For more details on the maximum time, see the device datasheet.
- `FlsSystemcallCalloutFunction`: Specifies a callout function, which is called whenever the flash driver calls the system-call.
- `FlsSetCycleModeApi`: Specifies whether the `Fls_SetCycleMode` function is enabled or disabled.

The job end notification is configurable on configuration parameter `FlsJobEndNotification`. The job error notification is configurable on configuration parameter `FlsJobErrorNotification`.

To avoid the watchdog timer trigger reset, you may have to clear the watchdog timer from the flash driver in the following cases:

- The following parameters are high value: `FlsConfigSet/FlsMaxReadFastMode`, `FlsConfigSet/FlsMaxReadNormalMode`, `FlsConfigSet/FlsMaxWriteFastMode`, and `FlsConfigSet/FlsMaxWriteNormalMode`.
- The execution time of `Fls_MainFunction()` takes longer than the timeout of the watchdog timer due to low CPU operating frequency and so on.

In such cases, you must implement `Fls_WdgClear()`. The template of `Fls_WdgClear()` is defined in *Fls_CfgDer.c*. If the configuration parameter `FlsDefineWdgClear` is TRUE, implement it directly in *Fls_CfgDer.c*. Otherwise, you must define the function in any of your source file.

For example, in the case of configuring the WDG module:

```
#include <Wdg.h> /* Wdg Driver header file */
FUNC(void, FLS_CODE) Fls_WdgClear(void)
{
     /* This function is implemented for clearing the watchdog timer by user. */
     Wdg_SetTriggerCondition( xxxx );

     return;
}
```

2.3 Adapting your application

To use the flash driver in your application, you first have to include the flash driver header file by adding the following code line to your source file:

```
#include "Fls.h" /* Fls Driver */
```

This publishes all the required function/data prototypes and symbolic names of the configuration to the application.

In addition, you should also implement the error callout function for ASIL safety extension.

Declare the error callout function in the specified file by the `FlsIncludeFile` parameter and implement in your application (see Required callback functions, Error callout API).

The error callout function name can be configured by the `FlsErrorCalloutFunction` parameter.

The erase callout function name can be optionally configured by the `FlsEraseCalloutFunction` parameter.

The DED error notification name can be optionally configured by the `FlsDedErrorNotification` parameter.

The SED error notification name can be optionally configured by the `FlsSedErrorNotification` parameter.

The callout function for invocation of system-call can be optionally configured by the `FlsSystemcallCalloutFunction` parameter.

In the next step, the FLS should be initialized and configured. The configuration of the FLS with the flash driver using EB tresos Studio is explained in *EB tresos Studio for ACG8 user's guide* [7].

The FLS initialization can be done with the following function call and parameter:

When configuration variant is VARIANT-POST-BUILD (Postbuild), the parameter is the address of a const variable called `Fls_Config_<number>` (for example, `Fls_Config_0`),

```
Fls_Init(&Fls_Config_0);
```

Note: *`Fls_Config_<number>` can be referred by including `Fls_PBcfg.h`.*

When the configuration variant is VARIANT-PRE-COMPILE (Precompile) and only one `FlsConfigSet` is configured,

```
Fls_Init(NULL_PTR);
```

All other API calls can be used after successful initialization of the FLS whenever necessary.

If you use the MCU with data cache, and the data cache is enabled, the following areas must be allocated to non-cacheable region by setting of memory protection unit (MPU):

- Work flash region
- A section `FLS_START_SEC_VAR_NO_INIT_UNSPECIFIED` in *Fls_MemMap.h*
- A section `FLS_START_SEC_SYSCALLSHARED_VAR_NO_INIT_32` in *Fls_MemMap.h* (Fls_TS_T40D13M2I0R0)

For detailed information, see Memory allocation keyword and Memory protection unit (MPU).

Use DMA to read data from the flash memory unless the configuration parameter `FlsGeneral/FlsUseDmaForRead` is set to FALSE (it may be set to FALSE if you do not need to detect ECC errors). In such uses of DMA, you must enable the DMA controller before using the flash driver by using one of the following ways because the flash driver does not enable the DMA controller:

- Set ENABLED bit (Bit No.31) in DMAC_CTL register to 1.
- Configure the MCU module with `McuDmaEnable=true` and call the `Mcu_SetMode()` function with the configured mode.

For detailed information, see **DMA**.

2.4 Starting the build process

Do the following to build your application:

Note: *For a clean build, use the build command with target `clean_all`. before (`make clean_all`).*

1. On the command shell, type the following command to generate the necessary configuration-dependent files. See Generated files for details.
   ```
   > make generate
   ```
2. Type the following command to resolve all required file dependencies.
   ```
   > make depend
   ```
3. Type the following command to compile and link the application:
   ```
   > make (optional target: all)
   ```

The application is now built. All files are compiled and linked to a binary file, which can be downloaded to the target hardware.

2.5 Measuring stack consumption

Do the following to measure stack consumption. It requires the Base module for proper measurement.

Note: *All files (including library files) should be rebuilt with the dedicated compiler option. The executable file built in this step must be used only to measure stack consumption.*

1. Add the following compiler option to the Makefile to enable stack consumption measurement.
   ```
   -DSTACK_ANALYSIS_ENABLE
   ```
2. Type the following command to clean library files.
   ```
   > make clean_lib
   ```
3. Follow the build process described in section **Starting the build process**.

Follow the instructions in the release notes and measure the stack consumption.

2.6 Memory mapping

The *Fls_MemMap.h* file in the *$(TRESOS_BASE)/plugins/MemMap_TS_T40D13M0I0R0/include* directory is a sample. This file is replaced by the file generated by the MEMMAP module. Input to the MEMMAP module is generated as *Fls_Bswmd.arxml* in the *$(PROJECT_ROOT)/output/generated/swcd* directory of your project folder.

2.6.1 Memory allocation keyword

- `FLS_START_SEC_CODE / FLS_STOP_SEC_CODE`

  The memory section type is CODE. All executable code is allocated in this section.

- `FLS_START_SEC_CONST_UNSPECIFIED / FLS_STOP_SEC_CONST_UNSPECIFIED`

  The memory section type is CONST. The following contents are allocated in this section:

  - All configuration data
  - Hardware register base address data

- `FLS_START_SEC_VAR_INIT_UNSPECIFIED / FLS_STOP_SEC_VAR_INIT_UNSPECIFIED`

  The memory section type is VAR. The following variable is allocated in this section:

  - Flash driver state

- `FLS_START_SEC_VAR_NO_INIT_UNSPECIFIED / FLS_STOP_SEC_VAR_NO_INIT_UNSPECIFIED`

Note: *When `FLS_STOP_SEC_VAR_NO_INIT_UNSPECIFIED` is allocated to the end of SRAM area, the write operation will be failed due to Silicon Errata 229.*

- `FLS_START_SEC_SYSCALLSHARED_VAR_NO_INIT_32 / FLS_STOP_SEC_SYSCALLSHARED_VAR_NO_INIT_32 (Fls_TS_T40D13M2I0R0)`

  The memory section type is VAR. The following variables are allocated in this section:

  - All variables except for flash driver state (status)

Note: *When data cache is enabled in the MCU (for example, Arm® Cortex®-M7 CPU), this memory section must be in non-cacheable region by setting of MPU. For further information, see Memory protection unit (MPU).*

For allocation of this memory section to given section name *.autosar_fls_bss*, an example of *Fls_MemMap.h* is shown as follows. By linker, the section of *.autosar_fls_bss* must be allocated to address in non-cacheable region.

(This is example for GHS compiler. If other compiler is used, confirm and follow the syntax rule of it.)

```
#ifdef FLS_START_SEC_VAR_NO_INIT_UNSPECIFIED
      :
  #else
    #define MEMMAP_STARTED
    #pragma ghs section bss=".autosar_fls_bss"    // add
      :
#endif
#ifdef FLS_STOP_SEC_VAR_NO_INIT_UNSPECIFIED
      :
  #else
    #undef MEMMAP_STARTED
    #pragma ghs section bss=default               // add
      :
#endif
```

3 Structure and dependencies

The flash driver consists of static, configuration, and generated files.

3.1 Static files

Static files of the flash driver are located in the directory *$(TRESOS_BASE)/plugins/Fls_TS_**. These files contain the functionality of the driver, which does not depend on the current configuration.

All necessary source files are automatically compiled and linked during the build process and all include paths are set.

3.2 Configuration files

The configuration of the flash driver is done using the EB tresos Studio software. When saving a project, the configuration description is written in the *Fls.xdm* file, located in your project folder under *$(PROJECT_ROOT)/config*. This file serves as the input to generate the configuration-dependent source and header files during the build process.

Note: In the Fls.epc file, each sector container included in the `FlsSectorList` *container must be arranged in the order of the value of the* `FlsSectorStartaddress` *parameter.*

3.3 Generated files

During the build process, the following files are generated based on the current configuration. They are located in the subfolder *output/generated* of your project folder.

- *include/Fls_Cfg.h* and *include/Fls_CfgDer.h* contain the configuration declarations for the AUTOSAR module FLS.
- *include/Fls_Irq.h* contains the configuration declarations of the interrupt service routine.
- *include/Fls_PBcfg.h* contains declarations of configuration variables required by the `Fls_Init` API.
- *make/Fls_cfg*.mak is currently empty.
- *src/Fls_CfgDer.c* contains the configuration relevant routine.
- *src/Fls_Irq.c* contains the interrupt service routine.
- *src/Fls_PBcfg.c* contains the structure of the `FlsConfigSet` and the memory map information of the flash sectors.

Note: You do not need to add the generated source files to your application make file. They are compiled and linked automatically during the build process.

- *swcd/Fls_Bswmd.arxml* contains BSW module description.

Note: Additional steps are required for the generation of the BSW module description.
In EB tresos Studio, follow the menu path ***Project*** > ***Build Project*** *and select* ***generate_swcd****.*

3.4 Dependencies

Figure 3 shows how the flash driver is embedded in the memory stack.

Note: *To use the flash driver, the flash EEPROM emulation (see Specification of flash EEPROM emulation [3]) and the BSW scheduler module (see Specification of RTE [5]) must be enabled and configured. Optionally, the default error tracer (see Specification of default error tracer [4]) can be enabled and configured.*

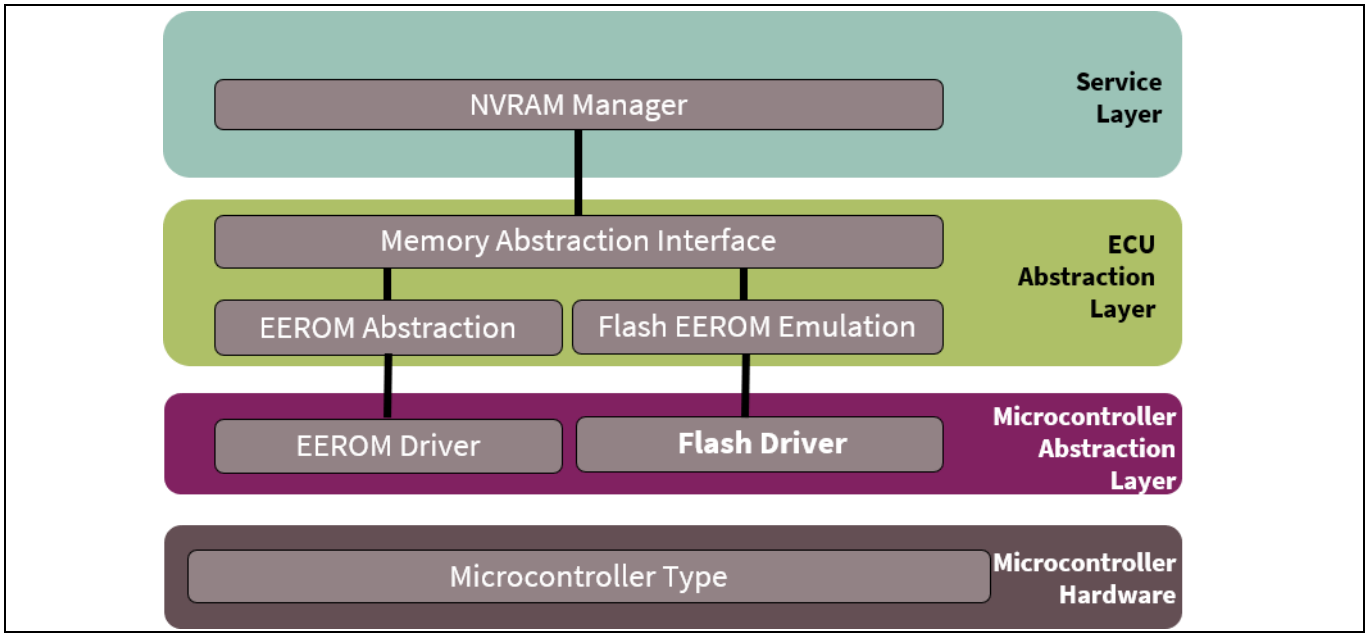

Figure 3 Relationship between the flash driver and other AUTOSAR modules

3.4.1 Flash EEPROM emulation (FEE)

The FEE is part of the ECU abstraction layer, which is located above the flash driver. It is the only module that calls flash driver functions and provides callback functions for flash driver events such as the job end notification or the job error notification.

3.4.2 DET

The default error tracer is optional and handles all errors.

3.4.3 BSW scheduler

The basic software scheduler calls the main function and handles the critical sections that are used within the flash driver.

3.4.4 Error callout handler

The error callout handler is called on every error that is detected, independently of whether default error detection is enabled or disabled. The error callout handler is an ASIL safety extension that is not specified by AUTOSAR. It is configured via the `FlsErrorCalloutFunction` configuration parameter.

4 EB tresos Studio configuration interface

The GUI is not part of the current delivery. For further information see *EB tresos Studio for ACG8 user's guide* [7].

Note: *The ECU parameter description of the Elektrobit automotive flash driver basically corresponds to the one defined by AUTOSAR in Specification of flash driver [2], chapter 10. However, because there are some vendor-specific extensions, use the ECU parameter description file that is delivered with the flash driver located in $(TRESOS_BASE)/plugins/Fls_TS_*/config/Fls.xdm.*

4.1 General configuration

The flash driver configuration, including different parameters and their meaning, is described in *Specification of flash driver* [2] and *Specification of ECU configuration parameters* [6]. See these documents for further information.

4.2 Vendor specific configuration

This section summarizes the differences between the configuration given in the *Specification of flash driver* [2] and *Specification of ECU configuration parameters* [6] and the configuration necessary for this flash driver.

4.2.1 Parameter constraints

The range of several parameters of the general flash driver configuration was reduced to hardware-specific values for the TRAVEO™ T2G microcontroller. These parameters are listed here, together with new hardware-specific and vendor-specific parameters. The parameters are preconfigured by using default values relevant for the selected derivative (when changing the derivative, a manual update is possible by clicking the **Calc** button). If a parameter is not used by the driver or if the parameter is not configurable, the field cannot be edited.

4.2.1.1 Container FlsGeneral

4.2.1.1.1 FlsAcLoadOnJobStart

Name

`FlsAcLoadOnJobStart`

Range

FALSE

Annotation

Driver does not load flash access code to RAM, so currently set as FALSE.

4.2.1.1.2 FlsBaseAddress

Name

`FlsBaseAddress`

Range

-

Annotation

Flash memory starts exactly at the given addresses. `FlsBaseAddress` is gathered from the Resource module and therefore configuration is not required.

4.2.1.1.3 FlsBlankCheckApi

Name

`FlsBlankCheckApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_BlankCheck` function.

4.2.1.1.4 FlsCancelApi

Name

`FlsCancelApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_Cancel` function.

4.2.1.1.5 FlsCompareApi

Name

`FlsCompareApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_Compare` function.

4.2.1.1.6 FlsDevErrorDetect

Name

`FlsDevErrorDetect`

Range

TRUE, FALSE

Annotation

Enables/disables the default error notification for the FLS driver. Setting this parameter to FALSE disables the notification of default errors via DET. However, in contrast to AUTOSAR specification, detection of default errors is still enabled as safety mechanisms (fault detection).

4.2.1.1.7 FlsDriverIndex

Name

`FlsDriverIndex`

Range

0

Annotation

Index of the driver. This parameter is not used in the flash driver. This will be assigned to the following symbolic names. The symbolic name derived of the general container short name prefixed with "`FlsConf_`" (`FlsConf_FlsGeneral`).

4.2.1.1.8 FlsGetJobResultApi

Name

`FlsGetJobResultApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_GetJobResult` function.

4.2.1.1.9 FlsGetStatusApi

Name

`FlsGetStatusApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_GetStatus` function.

4.2.1.1.10 FlsRuntimeErrorDetect

Name

`FlsRuntimeErrorDetect`

Range

TRUE, FALSE

Annotation

Enables/disables the runtime errors notification for the FLS driver. Setting this parameter to FALSE disables the notification of runtime errors via DET. However, in contrast to AUTOSAR specification, detection of runtime errors is still enabled as safety mechanisms (fault detection).

4.2.1.1.11 FlsSetModeApi

Name

`FlsSetModeApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_SetMode` function.

4.2.1.1.12 FlsTotalSize

Name

`FlsTotalSize`

Range

Total size of the available work flash memory. See the hardware manual.

Annotation

Flash memory length must exactly correspond to the available total size of work flash on the target device.

4.2.1.1.13 FlsUseInterrupts

Name

`FlsUseInterrupts`

Range

TRUE, FALSE

Annotation

Job processing triggered by hardware interrupt (TRUE) or not triggered by interrupt (FALSE). When this parameter is set to TRUE, the parameter `FlsGeneral/FlsUseNonBlockingWrite` cannot be set to TRUE.

4.2.1.1.14 FlsVersionInfoApi

Name

`FlsVersionInfoApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_GetVersionInfo` function.

4.2.1.2 Container FlsConfigSet

4.2.1.2.1 FlsAcErase

Name

`FlsAcErase`

Range

-

Annotation

This driver does not execute in RAM. Therefore, this parameter cannot be specified by the configuration tool.

4.2.1.2.2 FlsAcWrite

Name

`FlsAcWrite`

Range

-

Annotation

This driver does not execute in RAM. Therefore, this parameter cannot be specified by the configuration tool.

4.2.1.2.3 FlsCallCycle

Name

`FlsCallCycle`

Range

0.000..1.000

Annotation

Cycle time of calls of the flash driver's main function. The unit of this parameter is seconds. Therefore, the configured value of this parameter is rounded down in milliseconds. If the value is 0.000, Timeout supervision is not performed.

4.2.1.2.4 FlsDefaultMode

Name

`FlsDefaultMode`

Range

MEMIF_MODE_FAST, MEMIF_MODE_SLOW

Annotation

Default FLS device mode after initialization.

4.2.1.2.5 FlsJobEndNotification

Name

`FlsJobEndNotification`

Range

<FUNCTION_NAME>

Annotation

Mapped to the job end notification routine provided by some upper layer module, typically the FEE module.

Note: *Notifications must be declared and defined outside the FLS module. The file containing the declarations must be included using the parameter* `FlsGeneral/FlsIncludeFile`.

4.2.1.2.6 FlsJobErrorNotification

Name

`FlsJobErrorNotification`

Range

<FUNCTION_NAME>

Annotation

Mapped to the job error notification routine provided by some upper layer module, typically the FEE module.

Note: *Notifications must be declared and defined outside the FLS module. The file containing the declarations must be included using the parameter* `FlsGeneral/FlsIncludeFile`.

4.2.1.2.7 FlsMaxReadFastMode

Name

`FlsMaxReadFastMode`

Range

4..* (multiple of 4)

Annotation

Default value is 1024. This value is a multiple of 4. This value is used in reading, comparing, blank checking and verifying the written data after writing in fast mode.

4.2.1.2.8 FlsMaxReadNormalMode

Name

`FlsMaxReadNormalMode`

Range

4..* (multiple of 4)

Annotation

Default value is 128. This value is a multiple of 4. This value is used in reading, comparing, blank checking and verifying the written data after writing in normal mode.

4.2.1.2.9 FlsMaxWriteFastMode

Name

`FlsMaxWriteFastMode`

Range

4..* (multiple of 4)

Annotation

Default value is 64. This value is a multiple of 4. This value is used in writing job without hardware interrupt in fast mode.

4.2.1.2.10 FlsMaxWriteNormalMode

Name

`FlsMaxWriteNormalMode`

Range

4..* (multiple of 4)

Annotation

Default value is 16. This value is a multiple of 4. This value is used in writing the job without hardware interrupt in normal mode.

4.2.1.2.11 FlsProtection

Name

`FlsProtection`

Range

-

Annotation

The flash driver does not support protection. Therefore, this parameter cannot be specified by the configuration tool.

4.2.1.3 Container FlsDemEventParameterRefs

This container is not present because it is obsolete.

4.2.1.4 Container FlsExternalDriver

This container is not present because external flash is not supported.

4.2.1.5 Container FlsSector

4.2.1.5.1 FlsNumberOfSectors

Name

`FlsNumberOfSectors`

Range

1..*

Annotation

Number of continuous identical flash sectors. The maximum value depends on subderivative.

4.2.1.5.2 FlsPageSize

Name

`FlsPageSize`

Range

4

Annotation

Page size for write access of a sector.

4.2.1.5.3 FlsSectorSize

Name

`FlsSectorSize`

Range

2048 (large sector) or 128 (small sector)

Annotation

Size of a sector in bytes.

4.2.1.5.4 FlsSectorStartaddress

Name

`FlsSectorStartaddress`

Range

Virtual start address of a flash sector. See Virtual flash memory layout.

Annotation

Start address of a flash sector.

4.2.1.6 Container FlsPublishedInformation

4.2.1.6.1 FlsAcLocationErase

Name

`FlsAcLocationErase`

Range

-

Annotation

This driver does not execute in RAM. Therefore, this parameter is not used.

4.2.1.6.2 FlsAcLocationWrite

Name

`FlsAcLocationWrite`

Range

-

Annotation

This driver does not execute in RAM. Therefore, this parameter is not used.

4.2.1.6.3 FlsAcSizeErase

Name

`FlsAcSizeErase`

Range

-

Annotation

This driver does not execute in RAM. Therefore, this parameter is not used.

4.2.1.6.4 FlsAcSizeWrite

Name

`FlsAcSizeWrite`

Range

-

Annotation

This driver does not execute in RAM. Therefore, this parameter is not used.

4.2.1.6.5 FlsEraseTime

Name

`FlsEraseTime`

Range

0.16

Annotation

The unit of this parameter is seconds and represents the maximum time to erase one complete flash sector in all supported derivatives.

4.2.1.6.6 FlsErasedValue

Name

`FlsErasedValue`

Range

0xFFFFFFFF

Annotation

The erased value is regarded as 0xFFFFFFFF for four bytes.

4.2.1.6.7 FlsExpectedHwId

Name

`FlsExpectedHwId`

Range

TRAVEO

Annotation

The flash driver does not support external flash. Therefore, this parameter is not used.

4.2.1.6.8 FlsSpecifiedEraseCycles

Name

`FlsSpecifiedEraseCycles`

Range

250000

Annotation

Number of erase cycles specified for the flash device.

4.2.1.6.9 FlsWriteTime

Name

`FlsWriteTime`

Range

0.001

Annotation

The unit of this parameter is seconds.

4.2.2 Vendor and driver specific parameters

4.2.2.1 Container FlsGeneral

4.2.2.1.1 FlsErrorCalloutFunction

Name

`FlsErrorCalloutFunction`

Range

<FUNCTION_NAME>

Annotation

`FlsErrorCalloutFunction` is used to specify the error callout function name. The function is called on every error. The ASIL level of this function limits the ASIL level of the FLS driver.

Note: *`FlsErrorCalloutFunction` must be a valid C function name; otherwise an error can occur in the configuration phase.*

4.2.2.1.2 FlsIncludeFile

Name

`FlsIncludeFile`

Range

File names

Annotation

`FlsIncludeFile` is a list of the file names that shall be included within the driver. Any application-specific symbol that is used by the Fls configuration such as error callout function should be included by configuring this parameter.

Note: *`FlsIncludeFile` must be a filename with the .h extension and a unique name; otherwise errors can occur in the configuration phase.*

Note: *If the configuration parameter `FlsJobEndNotification`, `FlsJobErrorNotification`, `FlsDedErrorNotification`, and / or `FlsSedErrorNotification` are configured, Fee_Cbk.h (or another file containing the declarations) must be included by configuring this parameter because notifications have to be declared and defined outside the FLS module.*

4.2.2.1.3 FlsEraseVerification

Name

`FlsEraseVerification`

Range

TRUE, FALSE

Annotation

Enables/disables the erase verification (blank check) after erasing a flash block. (Only if the upper-layer module can ensure blankness of flash block, `FlsEraseVerification` can be set to FALSE.)

4.2.2.1.4 FlsBeforeWriteVerification

Name

`FlsBeforeWriteVerification`

Range

TRUE, FALSE

Annotation

Enables/disables the verification (blank check) before writing a flash block. (Only if the upper-layer module can ensure availability of flash block, `FlsBeforeWriteVerification` can be set to FALSE.)

4.2.2.1.5 FlsWriteVerification

Name

`FlsWriteVerification`

Range

TRUE, FALSE

Annotation

Enables/disables the write verification (compare) after writing the flash block. (Only if the upper-layer module can ensure data consistency, `FlsWriteVerification` can be set to FALSE.)

4.2.2.1.6 FlsEraseCalloutFunction

Name

`FlsEraseCalloutFunction`

Range

<FUNCTION_NAME>

Annotation

`FlsEraseCalloutFunction` is used to specify the erase callout function name. The function is called after an erase job is accepted.

Note: *`FlsEraseCalloutFunction` must be a valid C function name; otherwise an error can occur in the configuration phase.*

4.2.2.1.7 FlsReadImmediateApi

Name

`FlsReadImmediateApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_ReadImmediate` function.

4.2.2.1.8 FlsSuspendResumeApi

Name

`FlsSuspendResumeApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_Suspend / Fls_Resume` function.

4.2.2.1.9 FlsDmaChannel

Name

`FlsDmaChannel`

Range

0..*

Annotation

The DMA channel that is used for reading from work flash. The maximum value depends on the subderivative. If flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0) are used, a separate DMA channel must be used for each flash driver. If the configuration parameter `FlsUseDmaForRead` is FALSE, this parameter is not valid.

4.2.2.1.10 FlsAuxiliaryBufferSize

Name

`FlsAuxiliaryBufferSize`

Range

4..2048 (multiple of 4)

Annotation

The size of auxiliary buffer that stores data read from work flash by DMA transfer at a time for reading, verifying or comparing process. Default value is 128. This value is a multiple of 4. If the configuration parameter `FlsUseDmaForRead` is FALSE, this parameter is not valid.

4.2.2.1.11 FlsSetFlashCtlRegister

Name

`FlsSetFlashCtlRegister`

Range

FLS_FLASH_CTL_WORKONLY, FLS_FLASH_CTL_USERVALUE, FLS_FLASH_CTL_NOTSET

Annotation

Specifies the bit fields of FLASH_CTL register that are set by the flash driver.

FLS_FLASH_CTL_WORKONLY (Default): The only bit fields regarding work flash are set to the FLASH_CTL register.

FLS_FLASH_CTL_USERVALUE: The user-specified value is set to the FLASH_CTL register. The value is defined by the configuration parameter `FlsUserValueForFlashCtlRegister`.

FLS_FLASH_CTL_NOTSET: FLS driver does not set any value to the FLASH_CTL register.

4.2.2.1.12 FlsUserValueForFlashCtlRegister

Name

`FlsUserValueForFlashCtlRegister`

Range

0..*

Annotation

A value for the FLASH_CTL register that is set by the user when the configuration parameter `FlsSetFlashCtlRegister` is FLS_FLASH_CTL_USERVALUE. All significant bits in the register must be specified, but the bits of FLASH macro wait states (LSB 4 bits) are not set by flash driver.

4.2.2.1.13 FlsSetWorkFlashSafetyRegister

Name

`FlsSetWorkFlashSafetyRegister`

Range

TRUE, FALSE

Annotation

Specifies whether the WORK_FLASH_SAFETY register is set by the flash driver. If TRUE, the flash driver sets the WORK_FLASH_SAFETY register. Otherwise, it does not set the WORK_FLASH_SAFETY register.

4.2.2.1.14 FlsDefineWdgClear

Name

`FlsDefineWdgClear`

Range

TRUE, FALSE

Annotation

Specifies whether the function `Fls_WdgClear` to clear the watchdog timer is defined by the flash driver. If TRUE, the Flash driver defines the function `Fls_WdgClear`. Otherwise, it does not define the function `Fls_WdgClear`.

4.2.2.1.15 FlsUseNonBlockingWrite

Name

`FlsUseNonBlockingWrite`

Range

TRUE, FALSE

Annotation

Specifies whether the flash driver writes to work flash in non-blocking mode. If TRUE, the Flash driver writes in non-blocking mode. Otherwise, writes in blocking mode (Default). This parameter is not applied for the write operation to work flash block#1. The direct register accesses to FLASHC1 implies non-blocking mode.

4.2.2.1.16 FlsHsmPresent

Name

`FlsHsmPresent`

Range

TRUE, FALSE

Annotation

This parameter indicates whether the hardware security module (HSM) is present. If the HSM is not supported yet, this parameter should be FALSE.

4.2.2.1.17 FlsUseSafetyMechanism

Name

`FlsUseSafetyMechanism`

Range

TRUE, FALSE

Annotation

Preprocessor switch to enable and disable eCT flash safety mechanism for flash embedded (write or erase) operation. If other flash drivers (such as HSM, SHE) have not supported the safety mechanism yet, this parameter should be FALSE. If both flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0) are used, this parameter should be TRUE for using the safety mechanism because of the arbitration between both flash drivers.

4.2.2.1.18 FlsIpcStructure

Name

`FlsIpcStructure`

Range

0..*

Annotation

IPC structure (number) used for eCT flash safety mechanism and HSM communication. The maximum value depends on subderivative. If both flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0) are used, this parameter should be used for safety mechanism because of the arbitration between both flash drivers. Do not choose the IPC structures that are reserved for system calls. Set this parameter to the same values in both flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0). Refer to 5.1.15.3 Arbitration sequences for the detailed usage and behavior.

4.2.2.1.19 FlsIpcInterruptStructure

Name

`FlsIpcInterruptStructure`

Range

0..*

Annotation

IPC interrupt structure (number) used for eCT flash safety mechanism and HSM communication. The maximum value depends on subderivative. If both flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0) are used, this parameter should be used for safety mechanism because of the arbitration between both flash drivers. Do not choose the IPC interrupt structures that are reserved for system calls. Set this parameter to different values for flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0). Refer to 5.1.15.3 Arbitration sequences for the detailed usage and behavior.

4.2.2.1.20 FlsIpcReleaseEventNotification

Name

`FlsIpcReleaseEventNotification`

Range

1..*

Annotation

IPC interrupt structures to generate the IPC release event used for eCT flash safety mechanism. Each bitfield from LSB corresponds to the IPC interrupt structure that triggers the interrupt for an IPC release event. The maximum value depends on the subderivative. If both flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0) are used, use this parameter for safety mechanism because of the arbitration between both flash drivers. Do not choose the IPC interrupt structures that are reserved for system calls. Refer to 5.1.15.3 Arbitration sequences for the detailed usage and behavior.

4.2.2.1.21 FlsIpcNotificationEventToHsm

Name

`FlsIpcNotificationEventToHsm`

Range

0..*

Annotation

IPC interrupt structure (number) used for flash processing request to HSM. The maximum value depends on the subderivative. This parameter is used only for flash driver for pplication (Fls_TS_T40D13M1I0R0). If both flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM (Fls_TS_T40D13M2I0R0) are used, use flash driver for application (Fls_TS_T40D13M1I0R0) to request buffer invalidation to flash driver for HSM (Fls_TS_T40D13M2I0R0). Set this parameter to the same value as `FlsIpcInterruptStructure` in the flash driver for HSM (Fls_TS_T40D13M2I0R0). Refer to 5.1.15.3 Arbitration sequences for the detailed usage and behavior.

4.2.2.1.22 FlsWorkEmbeddedNotification

Name

`FlsWorkEmbeddedNotification`

Range

<FUNCTION_NAME>

Annotation

Work flash embedded notification routine. The notification routine is called after flash embedded (write or erase) operation if the eCT flash safety mechanism is enabled. You must implement this; the flash driver does not care about the definition of the routine.

Syntax example: `void WorkEmbeddedNotification (void)`

Note: *Notifications must be declared and defined outside the FLS module. The file containing the declarations must be included using the parameter `FlsGeneral/FlsIncludeFile`.*

4.2.2.1.23 FlsArbitrationTimeout

Name

`FlsArbitrationTimeout`

Range

0.000..60.000

Annotation

Tolerant time for arbitration (waiting for) to finish the flash operation that was started from another core, typically maximum time to erase one flash sector. The unit of this parameter is seconds. If there is a conflict in flash operation, the current operation by FLS driver will wait for the earlier operation to finish, and then retry to start the current operation. The maximum retry time until timeout is calculated by dividing the value of the `FlsGeneral/FlsArbitrationTimeout` parameter by the value of the `FlsConfigSet/FlsCallCycle` parameter. When the `FlsConfigSet/FlsCallCycle` is zero, the timeout will not be caused permanently. The default value of this parameter is 0.2 s (200 ms), which is the maximum time for erasing one flash sector plus a margin.

4.2.2.1.24 FlsSystemcallCalloutFunction

Name

`FlsSystemcallCalloutFunction`

Range

<FUNCTION_NAME>

Annotation

`FlsSystemcallCalloutFunction` is used to define the existence and to specify the name of a callout function for invocation of system-call. The function is called whenever the flash driver calls the system-call.

Note: *`FlsSystemcallCalloutFunction` must be a valid C function name; otherwise an error can occur in the configuration phase. You must implement the callout function to call the system-call properly. Moreover, the parameter must have following interface:*

```
Std_ReturnType Systemcall_Callout_Function_Name (uint32 *Fls_IpcContext);
```

The `Fls_IpcContext` parameter indicates SRAM address (SRAM_SCRATCH_ADDR) where the system-call parameters have been stored and can be used to initiate the system-call request by such S-LLD IPC driver.

If the callout function calls the system-call successfully, it must return E_OK; otherwise it must return E_NOT_OK.

4.2.2.1.25 FlsFaultStructure

Name

`FlsFaultStructure`

Range

0..*

Annotation

Fault structure (number) used for fault reporting. See Fault. The maximum value depends on subderivative.

4.2.2.1.26 FlsSetCycleModeApi

Name

`FlsSetCycleModeApi`

Range

TRUE, FALSE

Annotation

Preprocessor switch for enabling the `Fls_SetCycleMode` function. If TRUE, the `Fls_SetCycleMode` function is enabled. Otherwise, it is disabled (default).

4.2.2.1.27 FlsUseDmaForRead

Name

`FlsUseDmaForRead`

Range

TRUE, FALSE

Annotation

This parameter is used to indicate whether reading from work flash is performed by DMA transfer. If TRUE, the Flash driver reads with DMA transfer (default). Otherwise, reads without DMA transfer (with CPU transfer).

4.2.2.1.28 FlsSetWorkFlashFaultMaskRegister

Name

`FlsSetWorkFlashFaultMaskRegister`

Range

TRUE, FALSE

Annotation

Specifies whether the fault mask 1 (MASK1) and mask 2 (MASK2) registers for work flash are set by the flash driver. If TRUE, the flash driver sets the fault mask 1 register (default). Otherwise, it does not set the fault mask 1 register. If TRUE and the target device has two flash blocks, the flash driver sets the fault mask 2 register (default). Otherwise, it does not set the fault mask 2 register. See Fault for details.

4.2.2.1.29 FlsReportErrorIfNotBlank

Name

`FlsReportErrorIfNotBlank`

Range

TRUE, FALSE

Annotation

Specifies whether the FLS calls error callout functions (i.e., Error Callout Handler and `Det_ReportError()`) when a blank check job started by `Fls_BlankCheck()` detects the `FLS_E_VERIFY_ERASE_FAILED` error, which indicates non-blank. If TRUE, the flash driver calls the error callout functions for non-blank (default). Otherwise, it does not call the error callout functions for non-blank.

4.2.2.2 Container FlsConfigSet

4.2.2.2.1 FlsDedErrorNotification

Name

`FlsDedErrorNotification`

Range

<FUNCTION_NAME>

Annotation

Mapped to the DED error notification routine provided by some upper layer module.

Note: *Notifications must be declared and defined outside the FLS module. The file containing the declarations must be included using the parameter `FlsGeneral/FlsIncludeFile`.*

4.2.2.2.2 FlsSedErrorNotification

Name

`FlsSedErrorNotification`

Range

<FUNCTION_NAME>

Annotation

Mapped to the SED error notification routine provided by some upper layer module.

Note: *Notifications must be declared and defined outside the FLS module. The file containing the declarations must be included using the parameter `FlsGeneral/FlsIncludeFile`.*

4.2.2.2.3 FlsNumberOfDelayLoop

Name

`FlsNumberOfDelayLoop`

Range

0..4294967295

Annotation

This parameter specifies the number of delay (wait) loops for writing a 32-bit data. This value is typically calculated by the following formula:

```
<FlsNumberOfDelayLoop>
= <CPU Clock> * <Write Time> * <Margin> / <Cycle per Loop>
```

Where,

- `<CPU Clock>` is CPU clock per a microsecond
- `<Write Time>` is Typ 32-bit (with ECC) write time (from datasheet)
- `<Margin>` is the margin considering the tolerance
- `<Cycle per Loop>` is the CPU cycle per a loop. (This value depends on compiler optimization. For example, it is 1 for compiling by GHS)

For example, if CPU clock is 160 MHz, write time is 30 us, and margin is +5%, then:

```
<FlsNumberOfDelayLoop>
= 160 (cycle/usec) * 30 (usec) * 1.05 / 1
= 5040
```

Note: *If the value of this parameter is large, the response of `Fls_MainFunction()` for writing will be delayed. When the 32-bit write time is longer than the typical (even if it is max), the writing is completed because the next calls of `Fls_MainFunction()` processes accordingly, although the number of times the function is called increases. Therefore, it is unnecessary to set this parameter to a large value. In addition, do not set the value such that the watchdog timer's counter can reach the limit value.*

Note: *If this parameter is set to 4294967295, `Fls_MainFunction()` will wait until the lesser size of `FlsConfigSet/FlsMaxWriteNormalMode` (or `FlsConfigSet/FlsMaxWriteFastMode`) or remaining data at that time has been written.*

4.2.2.3 Container FlsSector

4.2.2.3.1 FlsSectorIdentifier

Name

`FlsSectorIdentifier`

Range

Selectable list entry

Annotation

Identifier of the predefined flash sector as specified in the hardware manual.

4.2.3 Other modules

4.2.3.1 Flash EEPROM emulation

The flash EEPROM emulation must be configured according to *Specification of flash EEPROM emulation* [3].

4.2.3.2 DET

The default error tracer (DET) must be configured according to *Specification of default error tracer* [4].

If runtime errors notification is activated and runtime error is detected, the following four runtime errors are supported by this flash driver:

- `FLS_E_ERASE_FAILED`
- `FLS_E_WRITE_FAILED`
- `FLS_E_READ_FAILED`
- `FLS_E_COMPARE_FAILED`

4.2.3.3 BSW scheduler

The flash driver uses the following services of the BSW scheduler to enter and leave critical sections:

- `SchM_Enter_Fls_FLS_EXCLUSIVE_AREA_0(void)`
- `SchM_Exit_Fls_FLS_EXCLUSIVE_AREA_0(void)`

Ensure that the BSW scheduler is properly configured and initialized before using the flash driver.

5 Functional description

The flash driver provides a hardware-independent interface for the flash EEPROM emulation to read, write, erase, and compare data from or to the flash memory. The flash driver only uses the work flash memory.

The flash driver is usually used via the flash EEPROM emulation (*Specification of flash EEPROM emulation* [3]) and therefore, its functions should not be called directly by the application. In general, the flash driver's functions (except the main function) are exclusively called by the flash EEPROM emulation.

5.1 Function of the flash driver

5.1.1 Flash driver state machine

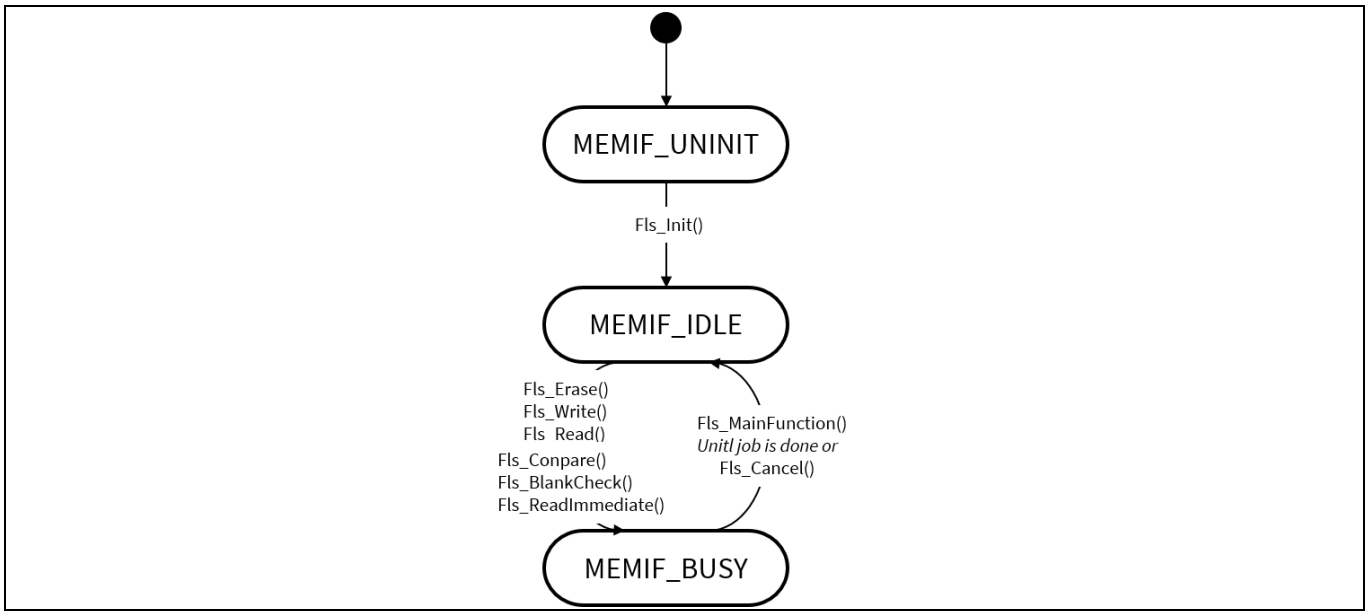

Figure 4 State machine of the flash driver

5.1.1.1 State MEMIF_UNINIT

After power on, the flash driver is in the `MEMIF_UNINIT` state in which it has not been initialized yet.

5.1.1.2 State MEMIF_IDLE

After successful initialization, the driver reaches the `MEMIF_IDLE` state and is ready. If an ongoing read, write, erase, compare, or blank check job is finished or canceled, the driver remains in this state and is ready for the next job.

Note: *After transition to the `MEMIF_IDLE` state, there is a possibility that the hardware is still working because the flash driver cannot abort the underlying hardware task even if it has been ready to accept a new job. In this case, it is necessary to be careful when the transition to the low-power consumption mode happens. See Retrieving the status information.*

5.1.1.3 State MEMIF_BUSY

In the `MEMIF_BUSY` state, the flash driver has accepted a read, write, erase, compare or blank check job, which will be executed during the next call(s) of the `Fls_MainFunction()` function until the job is finished or canceled by the user.

5.1.2 Flash driver job result state

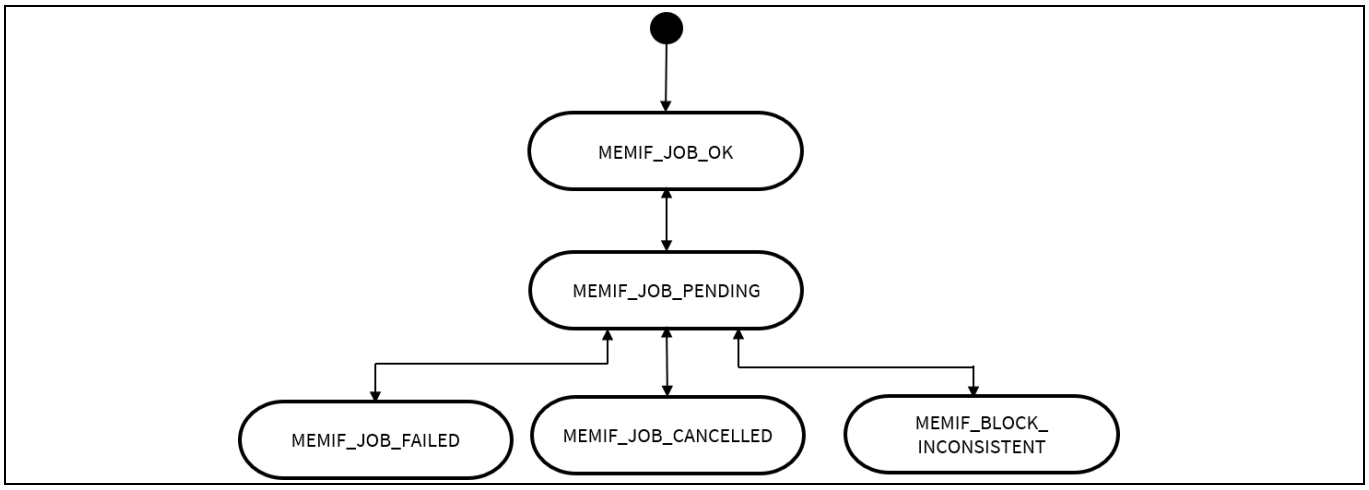

Figure 5 State machine of the job result

5.1.2.1 MEMIF_JOB_OK

The last job finished successfully. This state is also used after initialization.

5.1.2.2 MEMIF_JOB_PENDING

A read, write, erase, compare, or blank check job is pending and will be executed on the next call of `Fls_MainFunction()`.

5.1.2.3 MEMIF_JOB_CANCELED

The last job was canceled by the user via calling the `Fls_Cancel()` function.

5.1.2.4 MEMIF_JOB_FAILED

The last job failed due to a hardware error, timeout, and so on.

5.1.2.5 MEMIF_BLOCK_INCONSISTENT

This job result can only occur on compare jobs. It is set if the compare job yielded differences.

5.1.3 Initialization

The initialization is done via the function call.

In case of the VARIANT-POST-BUILD variant (Postbuild), the parameter is the address of a const variable `Fls_Config_<number>` (for example, `Fls_Config_0`):

```
Fls_Init(&Fls_Config_0);
```

In case of the VARIANT-PRE-COMPILE variant (Precompile), only one `FlsConfigSet` is configured:

```
Fls_Init(NULL_PTR);
```

After the initialization, the flash driver accepts a read, write, erase, compare, or blank check job for the flash memory.

5.1.4 Reading data from the flash memory

The flash driver supports reading data from the flash memory with blank checking (`Fls_Read()`) and without blank checking (`Fls_ReadImmediate()`). In the TRAVEO™ T2G microcontroller family, the blank check needs to be performed in advance of reading because undefined value is read from the blank (erased) area. Only if the upper-layer module has known where the blank areas are, for example, `Fls_BlankCheck()`, reading without blank checking can be used. See Checking blank for the flash memory. Otherwise, reading with the blank checking should be used. Generally, `Fls_Read()` is slower than `Fls_ReadImmediate()` due to significant overhead of blank checking.

A read job with the blank checking is set up via the following command:

```
ReturnValue = Fls_Read(SourceAddress, TargetAddressPtr, Length);
```

A read job without the blank checking is set up via the following command:

```
ReturnValue = Fls_ReadImmediate(SourceAddress, TargetAddressPtr, Length);
```

Note: *For use of `Fls_ReadImmediate()`, the configuration parameter `FlsGeneral/FlsReadImmediateApi` must be set to TRUE.*

If the function returns `E_OK`, the job was accepted and will be executed on the next call(s) of `Fls_MainFunction()`. The flash driver is now in the MEMIF_BUSY state and will not accept other commands. The job result is set to `MEMIF_JOB_PENDING`.

On each call of the main function, a specific number of bytes is copied from the flash memory `SourceAddress` to the `TargetAddressPtr`. The number of bytes depends on the memory layout (for example, gaps) and the configuration parameter such as `FlsConfigSet/FlsMaxReadNormalMode`. If both `SourceAddress` and `TargetAddressPtr` are multiples of 4, the latency of `Fls_MainFunction()` can reduce.

If the configuration parameter `FlsGeneral/FlsUseDmaForRead` is TRUE (default), the flash driver reads data with DMA transfer. The DMA channel used is specified by the configuration parameter `FlsGeneral/FlsDmaChannel`. The read data is stored once in the auxiliary buffer that the flash driver has prepared and is passed to the target data buffer that you have prepared. The auxiliary buffer size is determined by the configuration parameter `FlsGeneral/FlsAuxiliaryBufferSize`. The larger the size of the auxiliary buffer, the larger is the data read during a DMA transfer. However, this increases RAM consumption. The auxiliary buffer size is limited to the value of `FlsConfigSet/FlsMaxReadNormalMode` (or `FlsConfigSet/FlsMaxReadFastMode`) or a large sector size.

After the total number of bytes is successfully copied from the flash memory, the driver state is set back to MEMIF_IDLE and the job result is set to `MEMIF_JOB_OK`. In addition, the driver also calls an end notification function if it was configured with the `FlsConfigSet/FlsJobEndNotification` parameter.

If you are reading from a blank (erased) area, `Fls_Read()` copies all 0xFF data, whereas, `Fls_ReadImmediate()` copies indefinite data.

If a double-bit error was detected during the read process, the driver copies all 0xFF data and calls the error callout handler and the DET runtime errors notification (If the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_DED_FAILURE`, and the driver will continue the read job. DED error notification also will be called if it was configured with the parameter `FlsConfigSet/FlsDedErrorNotification`. In HSM (Fls_TS_T40D13M2I0R0), the driver detects the double-bit error as “read failed” and will set the job result to `MEMIF_JOB_FAILED` and call the error callout handler with the error code `FLS_E_READ_FAILED_FOR_CALLOUT` and the DET runtime errors notification (If the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_READ_FAILED` and the driver will abort the read job.

If a single-bit error was detected during the read process, the driver calls the error callout handler and the DET runtime errors notification (If the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_SED_FAILURE` and the driver will continue the read job. The SED error notification also will be called if it was configured with the parameter `FlsConfigSet/FlsSedErrorNotification`. In HSM (Fls_TS_T40D13M2I0R0), the driver cannot detect the single-bit error.

If there is a conflict in the flash operation (reading while erase/write), the flash driver will return to the upper layer (once) to wait for the earlier operation to finish, and then it will retry reading on the next call of `Fls_MainFunction()`. The maximum retry time until timeout is calculated by dividing the value of the configuration parameter `FlsGeneral/FlsArbitrationTimeout` by the value of the `FlsConfigSet/FlsCallCycle` parameter. If the maximum retry time exceeds, the driver will set the job result to `MEMIF_JOB_FAILED` and call the error callout handler and the DET error notification with the error code `FLS_E_TIMEOUT`.

If any other error occurred during the read process, the driver will set the job result to `MEMIF_JOB_FAILED` and call the error callout handler with the error code `FLS_E_READ_FAILED_FOR_CALLOUT` and the DET runtime errors notification (If the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_READ_FAILED` and the driver will abort the read job.

5.1.5 Writing data to the flash memory

The flash driver supports writing data to the flash memory with polling-controlled job and interrupt controlled job. The type of job used is determined by the configuration parameter `FlsGeneral/FlsUseInterrupts`. The interrupt-controlled job is used for performance enhancement for writing a large amount of data because it minimizes the calling and latency of `Fls_MainFunction()`. The polling-controlled job is simply used for writing data. In this case, calling and latency of `Fls_MainFunction()` depends on the configuration parameters such as `FlsConfigSet/FlsMaxWriteNormalMode` and `FlsConfigSet/FlsNumberOfDelayLoop`.

A write job is set up via the following command:

```
ReturnValue = Fls_Write(TargetAddress, SourceAddressPtr, Length);
```

Note: *The `TargetAddress` and the `Length` must be aligned to the flash page size. A flash page is the smallest amount of flash memory that can be programmed in one pass. The size of the flash page is architecture-dependent and outlined in Virtual flash memory layout.*

If the function returns `E_OK`, the job was accepted and will be executed on the next call(s) of `Fls_MainFunction()`. The flash driver is now in the MEMIF_BUSY state and will not accept other commands. The job result is set to MEMIF_JOB_PENDING.

If an interrupt-controlled job is used, on each call of the main function, up to one flash sector size of bytes is written from `SourceAddressPtr` to the flash memory `TargetAddress`. If an interrupt-controlled job is not finished within one call cycle, the main function must be called again until one flash sector is written. If polling-controlled job is used, on each call of the main function, a specific number of bytes is written from `SourceAddressPtr` to the flash memory `TargetAddress` at most. The number of bytes depends on the memory layout (such as gaps) and the configuration parameter such as `FlsConfigSet/FlsMaxWriteNormalMode`.

There are two operations modes to write (ProgramRow): blocking and non-blocking modes. By default, the flash driver uses blocking mode. If you want to use non-blocking mode for your use case, set the configuration parameter `FlsGeneral/FlsUseNonBlockingWrite` to TRUE. The write operation mode for work flash block#1 does not allow to configure and is handled as non-blocking mode.

After the total number of bytes was successfully written to the flash memory, the driver state is set back to MEMIF_IDLE and the job result is set to `MEMIF_JOB_OK`. In addition, the driver calls the end notification function if it was configured with the `FlsConfigSet/FlsJobEndNotification` parameter.

If any hardware error occurred during the write process, the driver will set the job result to `MEMIF_JOB_FAILED` and call the error callout handler with the error code `FLS_E_WRITE_FAILED_FOR_CALLOUT` and the DET runtime errors notification (If the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_WRITE_FAILED` and the driver will abort the write job.

The written data will be verified. On each call of the main function, a specific number of bytes is verified. The number of bytes depends on the memory layout (such as gaps) and the configuration parameter such as `FlsConfigSet/FlsMaxReadNormalMode`. For verification, if the configuration parameter `FlsGeneral/FlsUseDmaForRead` is TRUE (default), the flash driver reads data with DMA transfer. The DMA channel used is specified by the configuration parameter `FlsGeneral/FlsDmaChannel`. The read data is stored once in the auxiliary buffer that the flash driver has prepared and is compared with source data buffer that you have prepared. The larger the size of the auxiliary buffer, the larger is the data read during a DMA transfer. However, this increases RAM consumption. The auxiliary buffer size is limited to the value of `FlsConfigSet/FlsMaxReadNormalMode` (or `FlsConfigSet/FlsMaxReadFastMode`) or a large sector size.

If the verification fails, the driver will set the job result to `MEMIF_JOB_FAILED`. Additionally, the driver calls the error notification function if it was configured with the parameter `FlsConfigSet/FlsJobErrorNotification`. Only if the upper-layer module (typically the FEE module) can ensure data consistency by other means, the verification can be skipped by setting the configuration parameter `FlsGeneral/FlsWriteVerification` to FALSE to improve performance. Otherwise, the configuration parameter should be set to the default setting, TRUE, to ensure safety. Moreover, only if the upper-layer module can ensure availability of flash block for writing data by other means, verification (blank check) before writing flash block can be skipped by setting the configuration parameter `FlsGeneral/FlsBeforeWriteVerification` to FALSE to improve performance. Otherwise, the configuration parameter should be set to the default setting, TRUE, to ensure safety.

If there is a conflict in flash operation (writing or reading for verification while erase/write), the flash driver will return to the upper layer (once) to wait for the earlier operation to finish, and then it will retry writing or reading for verification on the next call of `Fls_MainFunction()`. The maximum retry time until timeout is calculated by dividing the value of the configuration parameter `FlsGeneral/FlsArbitrationTimeout` by the value of the parameter `FlsConfigSet/FlsCallCycle`. If the maximum retry time exceeds, the driver will set the job

result to MEMIF_JOB_FAILED and call the error callout handler and the DET error notification with the error code FLS_E_TIMEOUT.

5.1.6 Erasing data from the flash memory

The flash driver supports erasing (parts of) the flash memory with polling-controlled job and interrupt-controlled job. The type of job used is determined by the configuration parameter FlsGeneral/FlsUseInterrupts.

An erase job is set up via the following command:

```
ReturnValue = Fls_Erase(TargetAddress, Length);
```

Note: *The TargetAddress and the Length must be aligned to a flash sector. A flash sector is the smallest amount of flash memory that can be erased in one pass. The organization of flash sectors is architecture-dependent and outlined in Virtual flash memory layout.*

If the function returns E_OK, the job was accepted and will be executed on the next call(s) of Fls_MainFunction(). The flash driver is now in the MEMIF_BUSY state and will not accept other commands. The job result is set to MEMIF_JOB_PENDING.

If the erase callout function was configured with the parameter FlsGeneral/FlsEraseCalloutFunction, when the job was accepted, the function is called and the TargetAddress is passed as parameter.

After all the affected sectors are successfully erased, the driver state is set back to MEMIF_IDLE and the job result is set to MEMIF_JOB_OK. In addition, the driver calls the end notification function if it was configured with the parameter FlsConfigSet/FlsJobEndNotification.

If any hardware error occurred during the erase process, the driver will set the job result to MEMIF_JOB_FAILED and call the error callout handler with the error code FLS_E_ERASE_FAILED_FOR_CALLOUT and the DET runtime errors notification (if the configuration parameter FlsGeneral/FlsRuntimeErrorDetect is TRUE) with the error code FLS_E_ERASE_FAILED and the driver will abort the erase job.

The erase area will be verified. On each call of the main function, a specific number of bytes is verified. The number of bytes depends on the memory layout (such as gaps) and the configuration parameter such as FlsConfigSet/FlsMaxReadNormalMode. If the verification fails, the driver will set the job result to MEMIF_JOB_FAILED. In addition, the driver calls the error notification function if it was configured with the parameter FlsConfigSet/FlsJobErrorNotification. Only if the upper-layer module (typically the FEE module) can ensure that flash block is blank by other means, the verification can be skipped by setting the configuration parameter FlsGeneral/FlsEraseVerification to FALSE to improve performance. Otherwise, the configuration parameter should be set to TRUE (default) to ensure safety.

If there is a conflict in flash operation (erasing or verifying while erase/write), the flash driver will return to the upper layer (once) to wait for the earlier operation to finish, and then it will retry erasing or verifying on the next call of Fls_MainFunction(). The maximum retry time until timeout is calculated by dividing the value of the configuration parameter FlsGeneral/FlsArbitrationTimeout by the value of the parameter FlsConfigSet/FlsCallCycle. If the maximum retry time exceeds, the driver will set the job result to MEMIF_JOB_FAILED and call the error callout handler and the DET error notification with the error code FLS_E_TIMEOUT.

5.1.7 Comparing data from the flash memory

The flash driver supports comparing data between the flash memory and data in the RAM. A compare job is set up via the command:

```
ReturnValue = Fls_Compare(SourceAddress, TargetAddressPtr, Length);
```

Note: When reading for the comparing is done, it is performed without blank checking.

If the function returns `E_OK`, the job was accepted and will be executed on the next call(s) of `Fls_MainFunction()`. The flash driver is now in the `MEMIF_BUSY` state and will not accept other commands. The job result is set to `MEMIF_JOB_PENDING`.

On each call of the main function, a specific number of bytes is compared between the flash memory `SourceAddress` and the data at `TargetAddressPtr`. The number of bytes depends on the memory layout (such as gaps) and the configuration parameter such as `FlsConfigSet/FlsMaxReadNormalMode`. If both `SourceAddress` and `TargetAddressPtr` are multiples of 4, latency of `Fls_MainFunction()` can be minimized.

For comparison, if the configuration parameter `FlsGeneral/FlsUseDmaForRead` is TRUE (default), the flash driver reads data with DMA transfer. The used DMA channel is specified by the configuration parameter `FlsGeneral/FlsDmaChannel`. The read data is stored once in the auxiliary buffer that the flash driver has prepared and is compared with target data buffer that you have prepared. The auxiliary buffer size is determined by the configuration parameter `FlsGeneral/FlsAuxiliaryBufferSize`. The larger the size of the auxiliary buffer, the larger is the data read during a DMA transfer. However, this increases RAM consumption. The auxiliary buffer size is limited to the value of `FlsConfigSet/FlsMaxReadNormalMode` (or `FlsConfigSet/FlsMaxReadFastMode`) or a large sector size.

After the total number of bytes is successfully compared with the flash memory, the driver state is set back to `MEMIF_IDLE` and the job result is set to `MEMIF_JOB_OK`. In addition, the driver calls the end notification function if it was configured with the parameter `FlsConfigSet/FlsJobEndNotification`.

If the driver yielded differences between the two memory spaces, the driver will set the job result to `MEMIF_BLOCK_INCONSISTENT`, and calls the error notification function if it was configured.

If a double-bit error was detected during the compare process, the driver regards as all 0xFF data, calls the error callout handler and the DET runtime errors notification (if the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_DED_FAILURE`, and the driver will continue the compare job. DED error notification also will be called if it was configured with the parameter `FlsConfigSet/FlsDedErrorNotification`. In HSM (Fls_TS_T40D13M2I0R0), the driver detects the double-bit error as “compare failed” and will set the job result to `MEMIF_JOB_FAILED` and will call the error callout handler with the error code `FLS_E_COMPARE_FAILED_FOR_CALLOUT` and the DET runtime error notification (if the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_COMPARE_FAILED`, and the driver will abort the compare job.

If a single-bit error was detected during the compare process, the driver calls the error callout handler and the DET runtime errors notification (If the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_SED_FAILURE` and the driver will continue the compare job. SED error notification also will be called if it was configured with the parameter `FlsConfigSet/FlsSedErrorNotification`. In HSM (Fls_TS_T40D13M2I0R0), the driver cannot detect the single-bit error.

If there is a conflict in flash operation (reading for comparison while erase/write), the flash driver will return to the upper layer (once) to wait for the earlier operation to finish, and then it will retry reading for comparison on the next call of `Fls_MainFunction()`. The maximum retry time until timeout is calculated by dividing the value of the configuration parameter `FlsGeneral/FlsArbitrationTimeout` by the value of the parameter `FlsConfigSet/FlsCallCycle`. If the maximum retry time exceeds, the driver will set the job result to `MEMIF_JOB_FAILED` and call the error callout handler and the DET error notification with the error code `FLS_E_TIMEOUT`.

If any other error occurred during the compare process, the driver will set the job result to `MEMIF_JOB_FAILED` and call the error callout handler with the error code `FLS_E_COMPARE_FAILED_FOR_CALLOUT` and the DET runtime errors notification (if the configuration parameter `FlsGeneral/FlsRuntimeErrorDetect` is TRUE) with the error code `FLS_E_COMPARE_FAILED` and the driver will abort the compare job.

5.1.8 Checking blank for the flash memory

The flash driver supports checking blank for a given area in the flash memory. A blank check job is set up via the command:

```
ReturnValue = Fls_BlankCheck(TargetAddress, Length);
```

If the function returns `E_OK`, the job was accepted and will be executed on the next call(s) of `Fls_MainFunction()`. The flash driver is now in the `MEMIF_BUSY` state and will not accept other commands. The job result is set to `MEMIF_JOB_PENDING`.

On each call of the main function, a specific number of bytes is checked blank for the flash memory `TargetAddress`. The number of bytes depends on the memory layout (such as gaps) and the configuration parameter such as `FlsConfigSet/FlsMaxReadNormalMode`.

After the total number of bytes was judged blank for the flash memory, the driver state is set back to `MEMIF_IDLE` and the job result is set to `MEMIF_JOB_OK`. In addition, the driver also calls an end notification function if it was configured with the `FlsConfigSet/FlsJobEndNotification` parameter.

If any area was not judged blank, the driver will set the job result to `MEMIF_JOB_FAILED`. If the configuration parameter `FlsGeneral/FlsReportErrorIfNotBlank` is set to TRUE, the driver will call the error callout handler and the DET error notification with the error code `FLS_E_VERIFY_ERASE_FAILED`. In addition, the driver calls the error notification function if it was configured with the parameter `FlsConfigSet/FlsJobErrorNotification`.

If there is a conflict in flash operation (checking blank while erase/write), the flash driver will return to the upper layer (once) to wait for the earlier operation to finish, and then it will retry checking blank on the next call of `Fls_MainFunction()`. The maximum retry time until timeout is calculated by dividing the value of the configuration parameter `FlsGeneral/FlsArbitrationTimeout` by of the value of the parameter `FlsConfigSet/FlsCallCycle`. If the maximum retry time exceeds, the driver will set the job result to `MEMIF_JOB_FAILED` and call the error callout handler and the DET error notification with the error code `FLS_E_TIMEOUT`.

5.1.9 Canceling a job prior to maturity

Any ongoing flash job can be canceled by calling the function:

```
Fls_Cancel();
```

Note: *This function must not be called during the execution of the `Fls_MainFunction()`, `Fls_Suspend()` or `Fls_Resume()`.*

The function always cancels the ongoing job, sets the pending job result to MEMIF_JOB_CANCELED and sets the driver back to MEMIF_IDLE. It also calls the error notification function if it was configured with the parameter `FlsConfigSet/FlsJobErrorNotification`.

The driver is ready for the next job right after returning from this function call.

5.1.10 Retrieving the status information

Two API functions are offered to get the current state of the driver and the current state of the job result:

```
DriverState = Fls_GetStatus();
JobResult = Fls_GetJobResult();
```

For more information on the driver's state, see Flash driver state machine.

For more information on the job result, see Flash driver job result state.

Note: *While the flash memory cells are being programmed or erased, the microcontroller shall not be transited to low-power consumption modes. Whether the flash memory processing is ongoing can be known by calling the `Fls_GetStatusSub` function (Note that it is not `Fls_GetStatus`) and the microcontroller can be transited to the modes only if `Fls_GetStatusSub` returns `FLS_STATUS_IDLE`. The function `Fls_GetStatusSub` has the following interface (The interface and the macros `FLS_STATUS_IDLE` and `FLS_UCHAR` are declared in Fls.h.).*

```
FLS_UCHAR Fls_GetStatusSub(void);
```

Note: *The function `Fls_GetStatusSub` does not have critical sections (exclusive area).*

5.1.11 Setting the driver operation mode

The driver can be switched between slow and fast operation modes. The default mode configured with the `FlsConfigSet/FlsDefaultMode` parameter is applied right after initialization. To switch to the fast mode, the following function must be called:

```
Fls_SetMode(MEMIF_MODE_FAST);
```

The driver will switch to the fast operation mode in which the configured parameters for fast mode are valid. This affects the `FlsConfigSet/FlsMaxReadFastMode` parameter for read and compare jobs and the verify process of write and erase jobs and the `FlsConfigSet/FlsMaxWriteFastMode` parameter for polling controlled write job.

Note: *The mode change can only be executed when the driver is in MEMIF_IDLE state.*

To return to the slow mode, the `Fls_SetMode` function must be called with the parameter MEMIF_MODE_SLOW while the flash driver is in MEMIF_IDLE state.

5.1.12 Suspending a job

Any ongoing flash job can be suspended by calling the function:

```
ReturnValue = Fls_Suspend();
```

Note: *This function must not be called during the execution of the `Fls_MainFunction()` or `Fls_Resume()`.*

Note: *This function can be called for flash drivers for application (Fls_TS_T40D13M1I0R0) and HSM (Fls_TS_T40D13M2I0R0). However, make sure that the arbitration is taken care, for example, make sure that one core does not start an erase job while the other core is suspending the erase operation.*

Note: *The nested erase suspend operation is not supported. Suspending an erase job while the other erase operation is in the suspended state makes the previous erase job disappears and the erase resume job is only applicable for the later suspended erase operation.*
The following is the problematic sequence:
Erase sector #0
Erase suspend
Erase sector#1
Erase suspend (HW suspended information for erase sector#0 is removed)
Erase resume (resume erasing Sector#1)

If the function returns `E_OK`, the ongoing job was suspended. The flash driver is now in MEMIF_IDLE state and accepts other commands. The job result is set to MEMIF_JOB_OK.

`Fls_Write()`, `Fls_Read()`, `Fls_Compare()`, `Fls_BlankCheck()`, and `Fls_ReadImmediate()` can start a new job after returning from this function. However, if the target address (from start address to end address (start address + length)) for the job lies within the sector used by suspended job, the API functions reject the request, raise the default error `FLS_E_BUSY` and return with `E_NOT_OK`. Whereas, `Fls_Erase()` rejects for anywhere.

If this function is called to suspend an erase job that was resumed by `Fls_Resume()`, the call must be done at least 250 microseconds after `Fls_Resume()` finishes. Otherwise, the erase job cannot progress.

5.1.13 Resuming a suspended job

A suspended flash job can be resumed by calling the function:

```
ReturnValue = Fls_Resume();
```

Note: *This function must not be called during the execution of the `Fls_Suspend()`.*

Note: *This function can be called for flash drivers for application (Fls_TS_T40D13M1I0R0) and HSM (Fls_TS_T40D13M2I0R0). However, make sure that the arbitration is taken care, for example, make sure that one core does not start an erase job while the other core is suspending the erase operation.*

If the function returns `E_OK`, the suspended job was resumed. The flash driver is now in the `MEMIF_BUSY` state and will not accept other commands. The job result is set to `MEMIF_JOB_PENDING`.

5.1.14 Timeout supervision

The driver provides a timeout monitoring for the deadline of read, write, erase, compare and blank check functions.

The maximum timeout value is calculated based on the following.

- Specified length to read, write, erase or compare data, or blank check
- Cycle of `Fls_MainFunction()` function (`FlsCallCycle`)
- Kind of operation (Read, write, erase, compare or blank check)
- Conflict of flash operation (Division `FlsArbitrationTimeout` by `FlsCallCycle`).

The driver can disable the timeout monitoring by setting the value of the configuration parameter `FlsCallCycle` to 0.000 or calling the following function:

```
ReturnValue = Fls_SetCycleMode(MEMIF_MODE_FAST);
```

Note: The mode change can only be executed when the driver is in MEMIF_IDLE state.

If the function returns `E_OK`, the timeout monitoring is disabled afterward. To enable the timeout monitoring again, the `Fls_SetCycleMode` function must be called with the parameter `MEMIF_MODE_SLOW` while the flash driver is in `MEMIF_IDLE` state.

5.1.15 eCT flash safety mechanism

The flash driver provides the eCT flash safety mechanism which is required for multicore flash operations. This feature allows to arbitrate the write, erase, and blank check operations from multiple cores and prevent the simultaneous system calls and FLASHC1 register accesses. For this feature to work properly, both flash drivers for application (Fls_TS_T40D13M1I0R0) and for HSM Fls_TS_T40D13M2I0R0 must enable safety mechanism (`FlsUseSafetyMechanism` = TRUE).
The eCT flash safety mechanism supports callback function from each flash driver to notify its counterpart running on the other core when a flash embedded (write or erase) operation is complete. The callback function is configured with `FlsWorkEmbeddedNotification` and required to be implemented by the application.

5.1.15.1 Related configurations

The configuration parameters related to the eCT flash safety mechanism are below:

- `FlsUseSafetyMechanism`
- `FlsIpcStructure`
- `FlsIpcInterruptStructure`
- `FlsIpcReleaseEventNotification`
- `FlsIpcNotificationEventToHsm` (used for HSM communication)
- `FlsWorkEmbeddedNotification`
- `FlsHsmPresent`
- `FlsArbitrationTimeout`

5.1.15.2 IPC lock acquisition and release

The drivers, supporting the eCT flash safety mechanism, acquire IPC lock which is specified with the configuration `FlsIpcStructure` before starting a flash operation. If the IPC lock acquisition fails, the flash operation is not executed, and the flash driver retries to acquire it at the next opportunity (e.g. the next `Fls_MainFunction()` call). After finishing a flash operation, the flash driver releases the IPC lock for safety mechanism to allow the flash operations by other flash driver(s).

Table 2 shows the IPC lock acquisition and release timing.

Table 2 IPC lock acquisition and release for eCT flash safety mechanism

Operation	Acquisition timing	Release timing
Erase	Call `Fls_MainFunction()` (Start erase operation)	Call `Fls_MainFunction()` (Finish erase operation)
Write	Call `Fls_MainFunction()` (Start write operation)	Call `Fls_MainFunction()` (Finish write operation)
Blank check (erase verify)	Call `Fls_MainFunction()` (Every time `BlankCheck` system call is invoked)	Call `Fls_MainFunction()` (Every time `BlankCheck` system call is finished)
Blank check (pre-write verify)	Call `Fls_MainFunction()` (Every time `BlankCheck` system call is invoked)	Call `Fls_MainFunction()` (Every time `BlankCheck` system call is finished)
Blank check (`Fls_BlankCheck()`)	Call `Fls_MainFunction()` (Every time `BlankCheck` system call is invoked)	Call `Fls_MainFunction()` (Every time `BlankCheck` system call is finished)
Blank check (`read operation for single work flash or work flash block#0`)	-	-
Blank check (`read operation for work flash block#1`)	Call `Fls_MainFunction()` (Every time the blank check is started)	Call `Fls_MainFunction()` (Every time the blank check is finished)
Erase suspend	-	Call `Fls_Suspend()`
Erase resume	Call `Fls_Resume()`	-

Note: In case of the read operations for single work flash or work flash block#0, IPC lock for the eCT flash safety mechanism is not acquired. When the error status is returned due to the system call for erase or write by the other cores running, the flash driver suspends the read operation and retry it at the next `Fls_MainFunction()` call.

Note: When the blank check is executed for the work flash block#1, IPC lock for the eCT flash safety mechanism is acquired and released at the same timing as the system call `BlankCheck`.

If the flash driver fails to acquire IPC lock repeatedly, the flash operation is aborted due to the retry timeout. The maximum retry time is configured with `FlsArbitrationTimeout`.

5.1.15.3 Arbitration sequences

The sequence diagrams in this section show the arbitration behaviors for flash operations by multiple cores in an example use case. The preconditions for sequences are below:

- CM0+: Flash driver for HSM (Fls_TS_T40D13M2I0R0)
  - `FlsUseSafetyMechanism` = TRUE
  - `FlsIpcStructure` = 7 (IPC structure used for safety mechanism implementation)
  - `FlsIpcInterruptStructure` = 6 (IPC interrupt structure 6 is configured to notify to flash driver for HSM)
  - `FlsIpcReleaseEventNotification` = 0x000000C0 (IPC release event triggers interrupts on IPC interrupt structures #6 and #7)
- CM4: Flash driver for application (Fls_TS_T40D13M1I0R0)
  - `FlsUseSafetyMechanism` = TRUE
  - `FlsIpcStructure` = 7 (IPC structure used for safety mechanism implementation)
  - `FlsIpcInterruptStructure` = 7 (IPC interrupt structure 7 is configured to notify to flash driver for application)
  - `FlsIpcReleaseEventNotification` = 0x000000C0 (IPC release event triggers interrupts on IPC interrupt structures #6 and #7)
  - `FlsHsmPresent` = TRUE
  - `FlsIpcNotificationEventToHsm` = 6 (Flash driver for application should trigger interrupt on IPC interrupt structure #6 to notify the flash driver for HSM for occurrence of notify event)

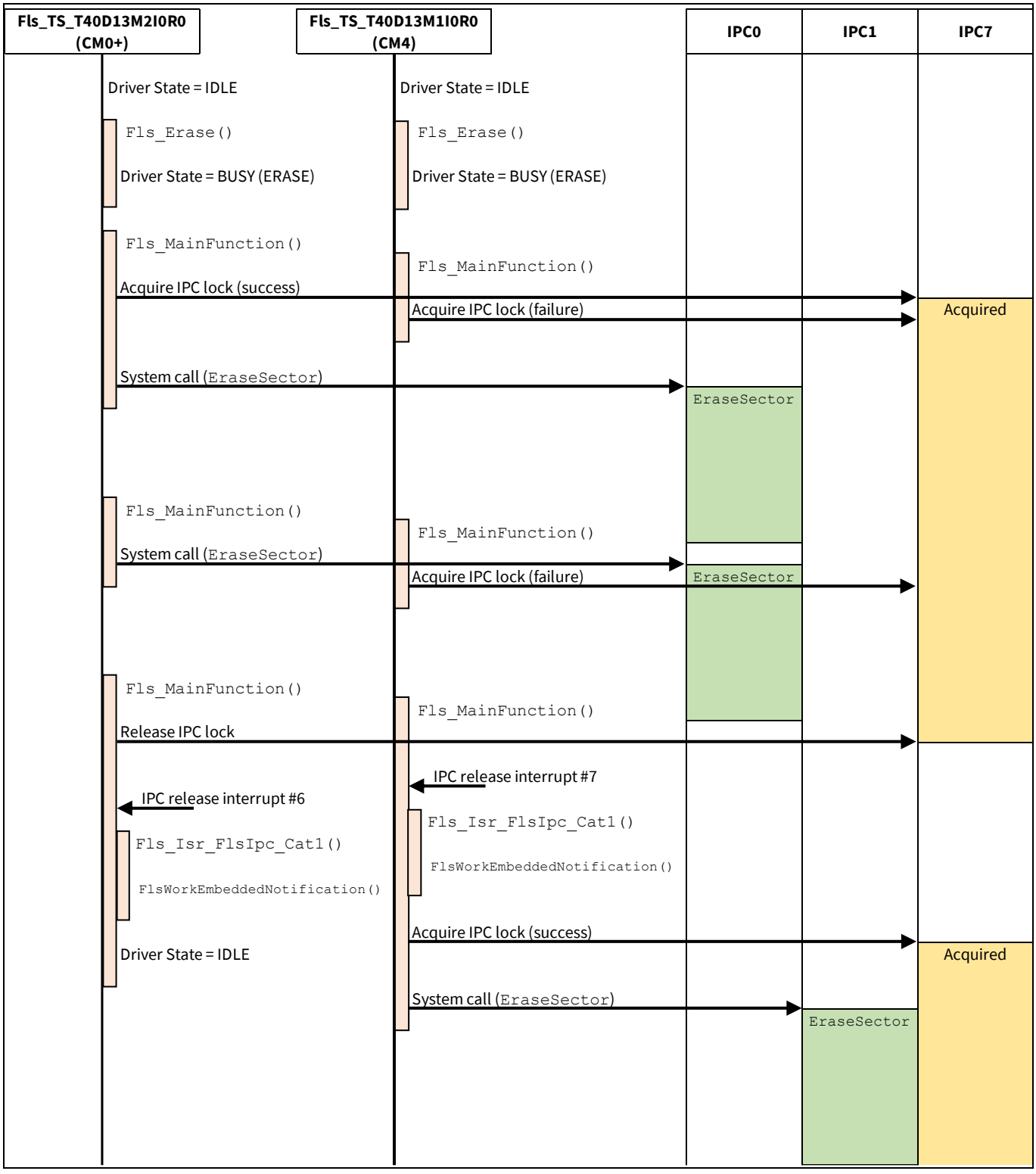

Figure 6 Erase operations by multiple flash drivers

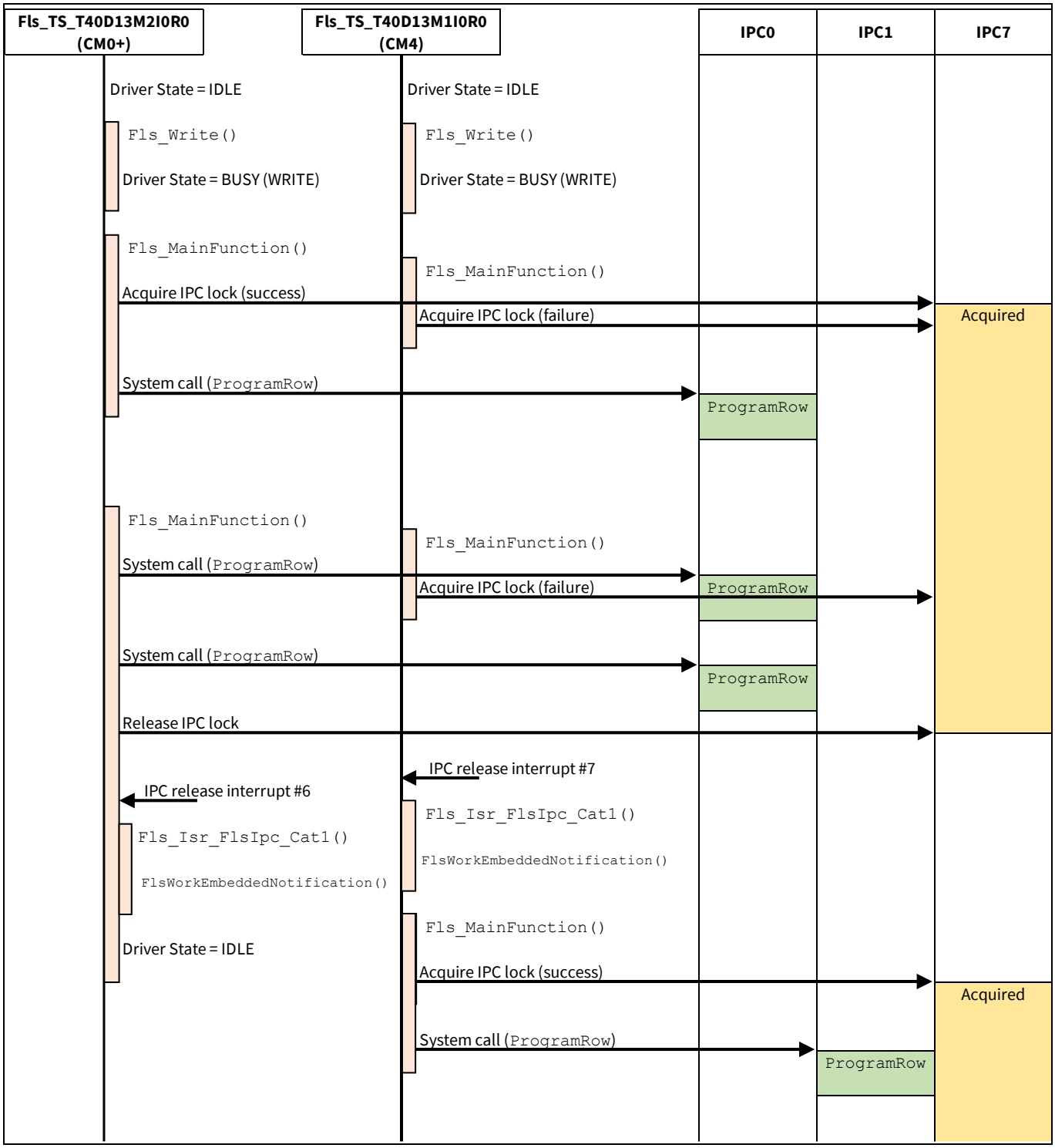

Figure 7 Write operations by multiple flash drivers

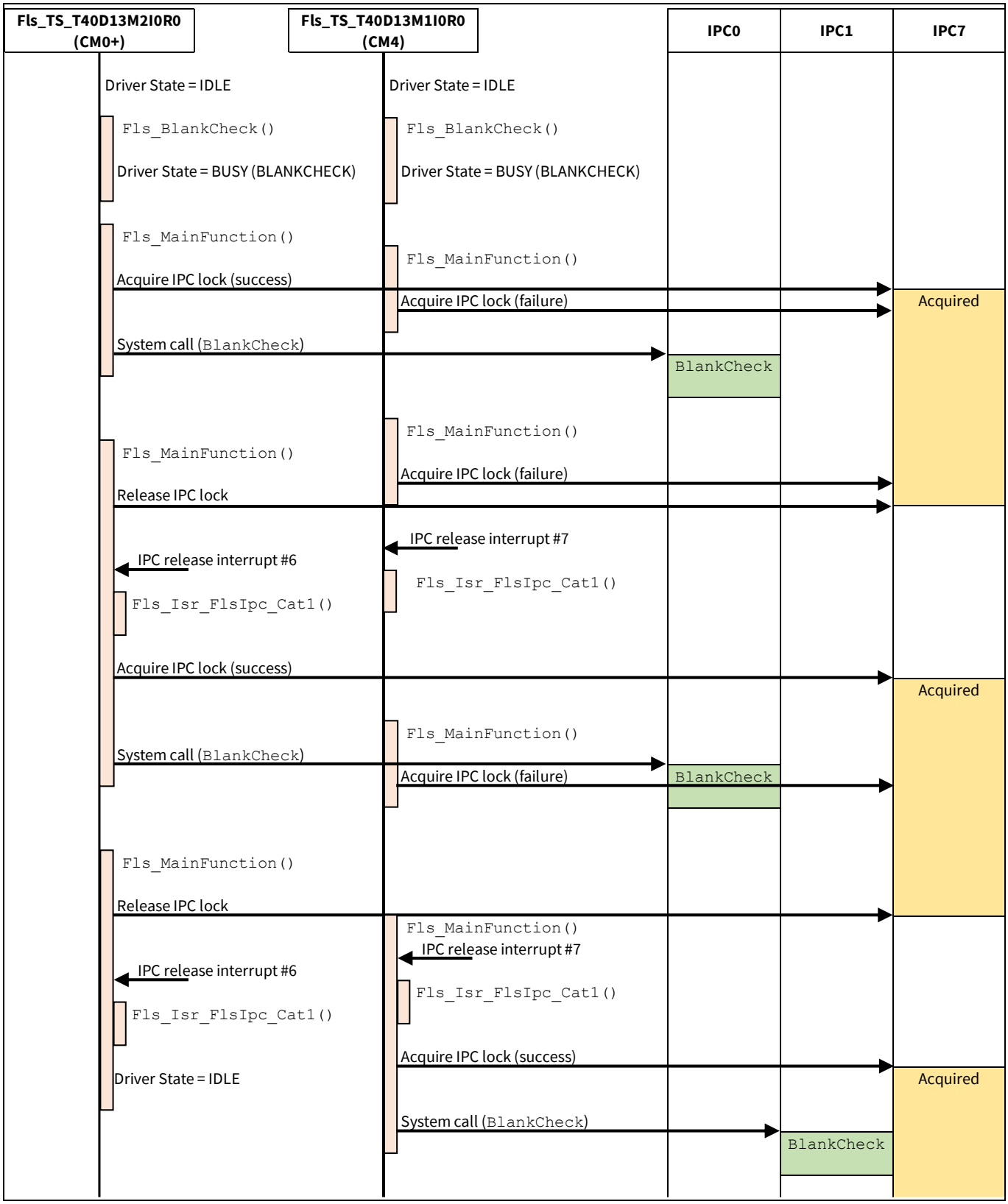

Figure 8 Blank check operations by multiple flash drivers

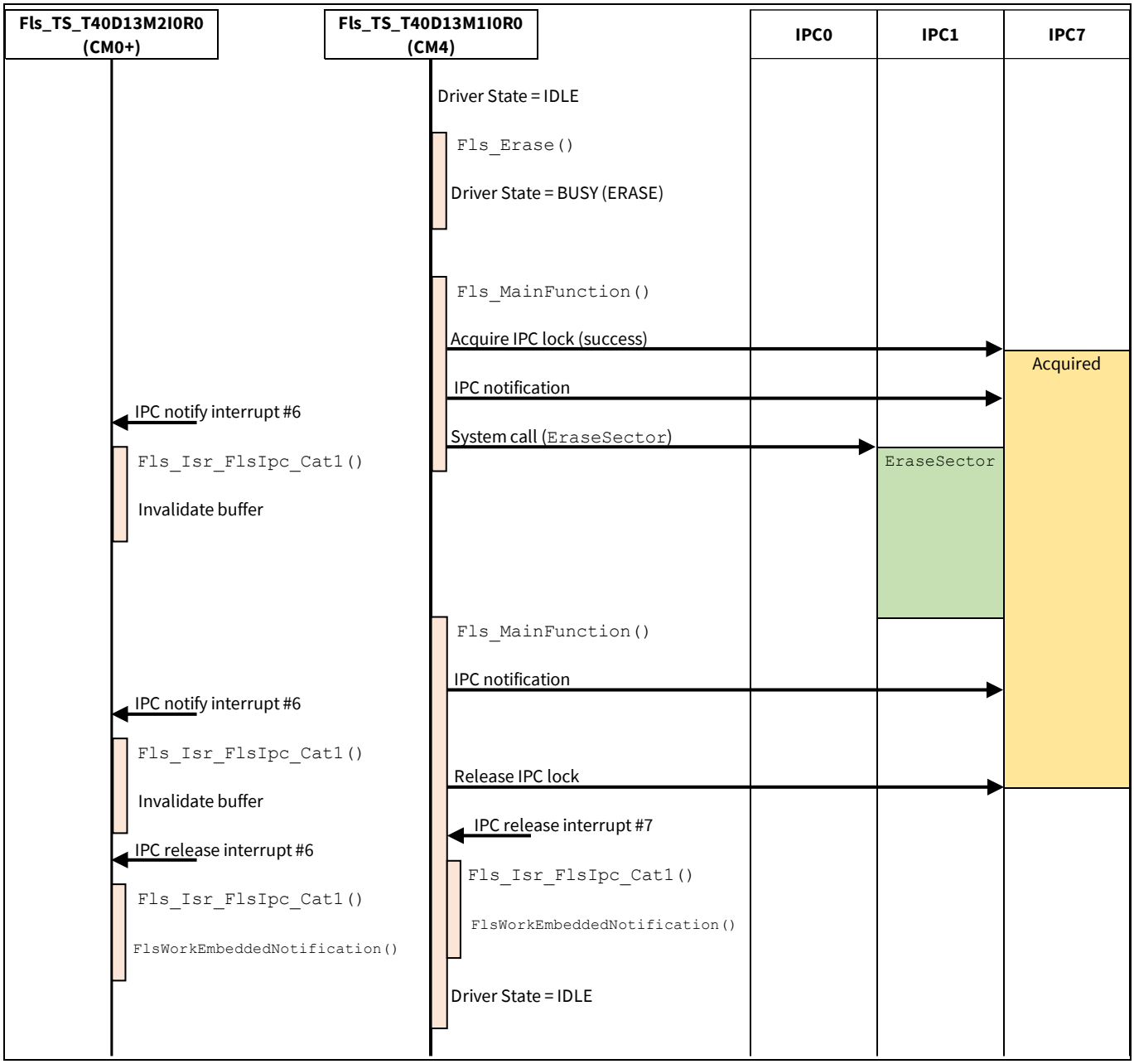

Figure 9 IPC notification and release events for HSM communication and safety mechanism

5.1.15.4 Assumptions of use

Basically, only 2 Fls instances are allowed for Fls multicore processing.
- Flash driver for HSM (Fls_TS_T40D13M2I0R0): CM0+
- Flash driver for application (Fls_TS_T40D13M1I0R0): CM4/CM7_0, CM7_1, CM7_2, or CM7_3

The following integrations are the wrong usages and are not supported:

- Multiple Flash drivers for application
  - Fls_TS_T40D13M1I0R0: CM7_0
  - Fls_TS_T40D13M1I0R0: CM7_1
- Implementation on the wrong core
  - Fls_TS_T40D13M1I0R0: CM0+
  - Fls_TS_T40D13M2I0R0: CM4 or CM7_X

Note: When non-Infineon flash drivers are integrated in the software along with flash driver for application and/or flash driver for HSM, it is the users' responsibility to implement and enable IPC based safety mechanism properly for smooth and error free flash operations.

5.1.15.5 Limitations

HW does not support the nested erase operation. The erase suspend job should be run only when there is no suspended erase job by the other cores.

5.2 Virtual flash memory layout

The flash driver always maps the available flash memory to a consecutive zero-based virtual flash address space. The flash driver uses the work flash memory only. Every subderivative has a specific work flash memory layout. See Hardware documentation about the physical address of the available work flash memory on each subderivative.

5.3 Parallel flash operations for separate work flash memories

There are two work flash blocks in several devices. Work flash block#0 is controlled by a flash controller (FLASHC registers), and system calls same as devices containing a single flash block. Flash operations for work flash block#1 are executed by FLASHC1 register accesses without system calls. The absence of resource competition allows for both flash blocks to be controlled by two CPU cores in parallel.

For example,

- CM0+: Flash driver for HSM (Fls_TS_T40D13M2I0R0)
  - `FlsUseSafetyMechanism` = FALSE
  - Container `FlsSector`: Work flash block#0 only
- CM7_0: Flash driver for application (Fls_TS_T40D13M1I0R0)
  - `FlsUseSafetyMechanism` = FALSE
  - `FlsHsmPresent` = TRUE
  - Container `FlsSector`: Work flash block#1 only

For parallel flash operations, `FlsUseSafetyMechanism` should be set to FALSE. If TRUE, a flash operation from one core is blocked while a flash driver in another core is executing a flash process.

Work flash block#1 is controlled with FLASHC1 register accesses. These direct register accesses are equivalent to the system calls in non-blocking mode. If the flash operations for work flash block#1 from multiple cores are required, the safety mechanism feature is mandatory.

For example,

- CM0+: Flash driver for HSM (Fls_TS_T40D13M2I0R0)
  - `FlsUseSafetyMechanism` = TRUE
  - Container `FlsSector`: Include work flash block#1
- CM7_0: Flash driver for application (Fls_TS_T40D13M1I0R0)
  - `FlsUseSafetyMechanism` = TRUE
  - `FlsHsmPresent` = TRUE
  - Container `FlsSector`: Include work flash block#1

5.4 Default error detection

The driver's services perform regular error checks.

When an error occurs, the error callout handler (configured via `FlsErrorCalloutFunction`) is called and the error code, service ID, module ID, and instance ID are passed as parameters.

If default error detection is enabled, all default errors are also reported to the default error tracer, a central error hook function within the AUTOSAR environment. The checking itself cannot be deactivated for safety reasons.

Table 3 shows the default error checks that are performed by the services of the flash driver.

Functions explains which error codes are reported by each API function.

Table 3 Default error codes

Related error code	Value	Type of error
FLS_E_PARAM_CONFIG	0x01	API service called with wrong parameter
FLS_E_PARAM_ADDRESS	0x02	API service called with wrong parameter
FLS_E_PARAM_LENGTH	0x03	API service called with wrong parameter
FLS_E_PARAM_DATA	0x04	API service called with wrong parameter
FLS_E_UNINIT	0x05	API service called without module initialization
FLS_E_BUSY	0x06	API service called while driver still busy
FLS_E_VERIFY_ERASE_FAILED	0x07	Erase verification (blank check) failed
FLS_E_VERIFY_WRITE_FAILED	0x08	Write verification (compare) failed
FLS_E_TIMEOUT	0x09	Timeout exceeded
FLS_E_PARAM_POINTER	0x0a	API service called with NULL pointer
FLS_E_ERASE_FAILED_FOR_CALLOUT	0x81	Flash erase failed (HW). This error id is used to call the error callout handler.
FLS_E_WRITE_FAILED_FOR_CALLOUT	0x82	Flash write failed (HW). This error id is used to call the error callout handler.

Related error code	Value	Type of error
FLS_E_READ_FAILED_FOR_CALLOUT	0x83	Flash read failed (HW). This error id is used to call the error callout handler.
FLS_E_COMPARE_FAILED_FOR_CALLOUT	0x84	Flash compare failed (HW). This error id is used to call the error callout handler.
FLS_E_DED_FAILURE	0x85	Double bit error was detected (DED)
FLS_E_SED_FAILURE	0x86	Single bit error was detected (SED)

5.5 Runtime error detection

The following errors are reported to the default error tracer as runtime errors by the flash driver:

See Functions for a correlation between API functions and reported runtime error codes.

Table 4 Runtime error codes

Related error code	Value	Type of error
FLS_E_ERASE_FAILED	0x01	Flash erase failed (HW).
FLS_E_WRITE_FAILED	0x02	Flash write failed (HW).
FLS_E_READ_FAILED	0x03	Flash read failed (HW).
FLS_E_COMPARE_FAILED	0x04	Flash compare failed (HW).

When an error occurs, the error callout handler (configured via `FlsErrorCalloutFunction`) is also called and the error code (related default error code), service ID, module ID, and instance ID are passed as parameters.

5.6 Reentrancy

The API functions `Fls_GetStatus()`, `Fls_GetJobResult()`, and `Fls_GetVersionInfo()` are reentrant. All other API functions of the flash driver are not reentrants.

5.7 Debugging support

The flash driver does not support debugging.

6 Hardware resources

6.1 Registers

The flash driver for the TRAVEO™ T2G microcontroller deals with the registers listed in Appendix B – Access register table.

Note: You should set the following registers before using the flash driver:

1. *FLASHC_FLASH_CTL and FLASHC1_FLASH_CTL (MAIN_WS[bit3:0]): FLASH macro main interface wait states)*
2. *FLASHC_FLASH_CTL and FLASHC1_FLASH_CTL (all bits): FLASH control register (Set all significant bits in this register if the configuration parameter `FLSGeneral/FlsSetFlashCtlRegister` is set to FLS_FLASH_CTL_NOTSET)*

Note: WORK_BANK_MODE bit must be 0. WORK_ECC_EN and WORK_ERR_SILENT bit must be 1.

3. *FLASHC_WORK_FLASH_SAFETY and FLASHC1_WORK_FLASH_SAFETY (WorkFlashWriteEnable[bit0]): Work flash security enable register (Set to 1 if the configuration parameter `FLSGeneral/FlsSetWorkFlashSafetyRegister` is FALSE.)*
4. *DMAC_CTL (ENABLED[bit31]): M-DMA control register (Set to 1 if the configuration parameter `FlsUseDmaForRead` is TRUE.)*

6.2 Interrupts

The flash driver uses the following interrupts if the configuration parameter `FlsGeneral/FlsUseInterrupts` is TRUE:

- The dedicated IPC interrupt for System call (for writing)
- FLASH macro interrupt (for erasing)
- FLASH#1 macro interrupt (for erasing and writing; only devices with two work flash blocks)

If the eCT flash safety mechanism (the configuration parameter `FlsGeneral/FlsUseSafetyMechanism)` is TRUE and/or the HSM communication (the configuration parameter `FlsGeneral/FlsHsmPresent)` is TRUE, the following interrupt is also used:

- Configured IPC interrupt (by the configuration parameter `FlsGeneral/FlsIpcInterruptStructure`)

Additionally, you must DEFINE ISR for fault, which calls the fault handling function provided by the flash driver if the configuration parameter `FlsGeneral/FlsSetWorkFlashFaultMaskRegister)` is TRUE.

See Fault for information about ISR for fault.

The ISR must be declared in the AUTOSAR OS as Category 1 Interrupt or Category 2 Interrupt.

Note: The interrupt number (IRQ) depends on the subderivative. See Hardware documentation.

Therefore, you need to declare the following ISRs in the interrupt vector table (**_Cat1* for Category-1 ISR) or (OS) interrupt service routine (**_Cat2* for Category-2 ISR). The ISR is located in the generated file at the following location: *output/generated/src/Fls_Irq.c*.

- ISR_NATIVE(Fls_Isr_Ipc_Cat1) or ISR(Fls_Isr_Ipc_Cat2) (for IPC interrupt for System call)
- ISR_NATIVE(Fls_Isr_Flash_Cat1) or ISR(Fls_Isr_Flash_Cat2) (for flash macro interrupt)
- ISR_NATIVE(Fls_Isr_Flash1_Cat1) or ISR(Fls_Isr_Flash1_Cat2) (for flash#1 macro interrupt)
- ISR_NATIVE(Fls_Isr_FlsIpc_Cat1) or ISR(Fls_Isr_FlsIpc_Cat2) (for configured IPC interrupt)

Note: *If the flash driver is used on CM0+, the priority for above-mentioned interrupts must be set to a value more than '1'.*

Note: *On the Arm® Cortex®-M4 CPU, priority inversion of interrupts may occur under specific timing conditions in the integrated system with TRAVEO™ T2G MCAL. For more details, see the following errata notice.*

Arm® Cortex®-M4 Software Developers Errata Notice - 838869:
"Store immediate overlapping exception return operation might vector to incorrect interrupt"

If the user application cannot tolerate the priority inversion, a DSB instruction should be added at the end of the interrupt function to avoid the priority inversion.

TRAVEO™ T2G MCAL interrupts are handled by an ISR wrapper (handler) in the integrated system. Thus, if necessary, the DSB instruction should be added just before the end of the handler by the integrator.

6.3 Fault

The flash driver gets the fault information such as single-bit error (SED) or double-bit error (DED) from a centralized fault report structure. This centralized nature enables a system-wide, consistent handling of faults and only a single fault interrupt handler is required. Therefore, the flash driver cannot directly do the processing (such as clearing the validity bit field) for the fault report structures. If interrupt is disabled, the errors cannot be detected.

The flash driver uses a fault structure that is specified by the configuration parameter `FlsGeneral/FlsFaultStructure`.

You should implement the fault interrupt handler for the fault structure that was specified by the configuration parameter and the handler should call a fault handing function (`Fls_Fault_Handling()`) provided by the flash driver. The function is defined in the generated file in the following location: *output/generated/src/Fls_Irq.c*.

Fls driver performs a dummy read after the write operation in non-blocking mode, and the erase operation to make the logical bank of the work flash ready for the read operation. However, due to Silicon Errata 206, the dummy read will trigger a bus error. Fls driver acquires the error status from FAULT struct at the appropriate timing (read/compare/verify) and ignores the read error during the erase and write operation. The application does not need to analyze the fault information for Fls handling. The application, in the fault handler interrupt that occurred due to the dummy read, needs to do the following:

- Call `Fls_Fault_Handling`
- Clear the fault interrupt cause
- Clear the fault status

Note: *This fault handling is not applicable in the flash driver for HSM (Fls_TS_T40D13M2I0R0). It means that the flash driver for HSM cannot detect SED, but can detect DED; therefore, handles the DED in the same way as bus error (like HW failure).*

Note: *If both flash drivers of the application (Fls_TS_T40D13M1I0R0) and of HSM (Fls_TS_T40D13M2I0R0) are used, and you cannot determine which CPU core caused the fault by checking the error-caused address in the FAULT.DATA0 register, the configuration parameter `FlsGeneral/FlsSetWorkFlashFaultMaskRegister` must be set to FALSE. Otherwise, if an ECC error occurs when FLS for HSM reads data from the work flash, the fault report interrupt can be generated at the other core of the application.*

The example of the fault interrupt handler is shown as follows.

```
void userIrqFaultReportHandler(void)
{
    /* FAULT_STRUCT is top address of fault report structure. */
    if(FAULT_STRUCT->STATUS.bitField.VALID == 1U)
    {
        /* Check if an error-caused address is within area for this core.   */
        /*   The error-caused address is calculated by appending 0x10000000 */
        /*   to [bit26:0] in DATA0 register of fault report structure.      */
        if(WITHIN_AREA_FOR_THIS_CORE(FAULT_STRUCT->DATA0))
        {
            Fls_Fault_Handling(); /* Fault handling for Flash driver            */
        }
        Xxx(...);                 /* Fault handling for other than Flash driver */
         ...
    }
    FAULT_STRUCT->INTR.bitField.FAULT = 1U;
    FAULT_STRUCT->STATUS = 0x00000000UL;
}
```

6.4 IPC

The flash driver uses inter processor communication (IPC) for performing flash memory operation (writing, erasing, blank checking, and so on) with system calls or eCT flash safety mechanism.

A dedicated IPC structure (mailbox) for system calls is associated with each CPU core and the flash driver uses the IPC structure for CM0+, CM4/CM7_0, CM7_1, CM7_2, or CM7_3. If acquisition of the IPC structure fails, the flash diver retries or reports hardware error. Similarly, for the IPC interrupt structure, dedicated structure for system calls is associated with each CPU core and the flash driver uses it for CM0+, CM4/CM7_0, CM7_1, CM7_2, or CM7_3. The used resources are summarized as follows.

- IPC structure 0 (for invoking System call form CM0+)
- IPC structure 1 (for invoking System call form CM4/CM7_0)
- IPC structure 2 (for invoking System call form CM7_1)
- IPC structure 3 (for invoking System call form CM7_2)

- IPC structure 4 (for invoking System call form CM7_3)
- IPC interrupt structure 0 (for notifying System call to CM0+)
- IPC interrupt structure 1 (for notifying finish of System call to CM0+)
- IPC interrupt structure 2 (for notifying finish of System call to CM4/CM7_0)
- IPC interrupt structure 3 (for notifying finish of System call to CM7_1)
- IPC interrupt structure 4 (for notifying finish of System call to CM7_2)
- IPC interrupt structure 5 (for notifying finish of System call to CM7_3)

For eCT flash safety mechanism and/or the HSM communication, the flash driver uses the IPC structure that is configured by the configuration parameter, `FlsGeneral/FlsIpcStructure`, and the IPC interrupt structure that is configured by the configuration parameter, `FlsGeneral/FlsIpcInterruptStructure`. For eCT flash safety mechanism, HSM communication, or both, do not choose the IPC structures and the IPC interrupt structures that are reserved for system calls.

6.5 System call

The system call is used for flash memory operations such as write and erase. The IPC mechanism is used to invoke a system call in TRAVEO™ T2G. A dedicated IPC structure is associated with each core (CM0+, CM4/CM7_0, CM7_1, CM7_2, and CM7_3) to trigger a system call. The CPU acquires this dedicated IPC structure (used as a mailbox), writes the system call opcode and argument to the data field of the mailbox, and notifies the IPC interrupt structure. Typically, the argument is a pointer to SRAM where the API's parameters are stored. This results in an IRQ0 interrupt in CM0+. Note that all system calls are serviced by the CM0+ core. A CM0+ IRQ0 interrupt triggered by this method executes the system call. The result of the system call is passed through the same IPC mechanism. Before running system calls, IRQ0 and IRQ1 should be enabled and IRQ0 priority set to '1'. This is to make sure that IRQ1 has higher interrupt priority than IRQ0. By default, IRQ1 priority will be set to '0'. In addition, a part of the available SRAM is allocated for system call, and not available for users. You must keep the power of the SRAM area in enabled or retained state. For details, see hardware documents.

In the case of devices in which work flash block#0 and bllock#1 are mounted, system calls are invoked only for work flash block#0.

The system call can be invoked by the user's callout function. See `FlsSystemcallCalloutFunction` in Vendor and driver specific parameters.

Note: The system call must not be used on CPU core which the flash driver runs.

Table 5 shows a summary of the system calls that the flash driver uses.

Table 5 System calls

Name	Opcode	Description
`SiliconID`	0x00	SROM firmware version
`ProgramRow`	0x06	Programs the addressed flash page
`ConfigureFmInterrupt`	0x08	Configures FM interrupt
`EraseSector`	0x14	Erases the addressed flash sector
`EraseSuspend`	0x22	Suspends ongoing erase operation
`EraseResume`	0x23	Resumes an erase suspend operation
`BlankCheck`	0x2A	Performs blank check on eCT work flash memory

6.6 Memory protection unit (MPU)

As mentioned in IPC, the flash driver communicates with CM0+ via IPC for performing flash memory operation. If the data cache in Arm® CM7 processor is enabled and the areas of IPC accessed from both CM0+ and CM7_0, CM7_1, CM7_2, or CM7_3 are allocated in cacheable region, it is impossible for the areas to be assured coherency of the content for each core. Moreover, data to be written to flash memory is passed through certain SRAM area referred by IPC (writing is not performed by specifying an address); so, if the data cache is enabled, subsequent reading of written data would be incorrect because only the data held in data cache is read. Therefore, both flash memory area and the areas for IPC must be in non-cacheable regions. MPU can be used for setting of the region attribute. Memory protection including the region attribute should be performed on system level, so, the flash driver does not set up the MPU. When you enable data cache, you must configure the MPU.

The following areas must be allocated to non-cacheable region by setting of MPU.

- Work flash region
- A section `FLS_START_SEC_VAR_NO_INIT_UNSPECIFIED` in *Fls_MemMap.h* (for the areas for IPC)
- A section `FLS_START_SEC_SYSCALLSHARED_VAR_NO_INIT_32` in *Fls_MemMap.h* (Fls_TS_T40D13M2I0R0)

The following is an example of the MPU setting in Arm® Cortex®-M7 processor:

```
/* MPU configuration sample for ARM Cortex-M7 Processor           */
/* Note: This sample should be valid only for privileged accesses */
#define MPU_RASR_SIZE_64KB          (0x0FUL << 1U)// Region size 64KB
#define MPU_RASR_SIZE_512KB         (0x12UL << 1U)// Region size 512KB

#define MPU_NORMAL_NON_CACHEABLE    (1UL << 19U)  // Normal, Non-cacheable
#define MPU_SHARED_DEVICE           (1UL << 16U)  // Shared device
#define MPU_STRONGLY_ORDERED_DEVICE (0UL)                 // Strongly ordered

#define MPU_RASR_AP_FULL_ACCESS     (0x3 << 24U)  // Full access

#define MPU_RASR_ENABLE                     (1UL)           // Enables this region

#define MPU_CTRL_ENABLE                     (1UL)           // Enables the MPU
#define MPU_CTRL_PRIVDEFENA         (1UL << 2U)       // Enables background region

#define MPU                 ((MPU_Type *)0xE000ED90UL) // MPU registers base address

typedef struct
{
    uint32_t rbar;
    uint32_t rasr;
} stc_mpu_cfg_t;

const stc_mpu_cfg_t mpuConfig[] =
{
```

```
/* FLS bss region */ {0x28030000, (MPU_RASR_SIZE_64KB |
         MPU_NORMAL_NON_CACHEABLE | MPU_RASR_AP_FULL_ACCESS | MPU_RASR_ENABLE)},
/* Work Flash region */      {0x14000000, (MPU_RASR_SIZE_512KB |
         MPU_NORMAL_NON_CACHEABLE | MPU_RASR_AP_FULL_ACCESS | MPU_RASR_ENABLE)}
};

#define MPU_SETTING_NUM      (sizeof(mpuConfig)/sizeof(stc_mpu_cfg_t))
#define MPU_MAX_NUM   ((MPU->TYPE == 0x00001000)? (16U): (8U))

void userMpuSetting(void)
{
    volatile unsigned long i;

    /* Cleans and Invalidates Data Cache */
        ...

    __DMB();          // Make sure outstanding transfers are done

    MPU->CTRL = 0;    // Disable the MPU

    for (i = 0; i < MPU_SETTING_NUM; i++)
    {
        MPU->RNR  = i;                 // Select which MPU region to configure
        MPU->RBAR = mpuConfig[i].rbar; // Set region base address register
        MPU->RASR = mpuConfig[i].rasr; // Set region attribute and size register
    }

    /* Disabled unused regions */
    for (i = MPU_SETTING_NUM; i < MPU_MAX_NUM; i++)
    {
        MPU->RNR  = i;           // Select which MPU region to configure
        MPU->RBAR = 0;           // Set region base address register to 0
        MPU->RASR = 0;           // Set region attribute and size register to 0
    }

    /* Enable the MPU and background region (only for privileged accesses) */
    MPU->CTRL = (MPU_CTRL_ENABLE | MPU_CTRL_PRIVDEFENA);

    __DSB();     // Make sure outstanding transfers are done
    __ISB();     // Make sure outstanding transfers are done
}
```

6.7 DMA

The flash driver uses DMA transfer for reading from work flash if the configuration parameter `FlsGeneral/FlsUseDmaForRead` is TRUE. The reading is done at following instances when `Fls_MainFunction()` running.

- Read job initiated by `Fls_Read()` or `Fls_ReadImmediate()`.
- Verifying after data is written by the write job initiated by `Fls_Write()`.
- Compare job initiated by `Fls_Compare()`.

Note: *The flash driver does not enable the DMA controller. Therefore, you must enable the DMA controller before starting the jobs by using one of the following ways if the configuration parameter `FlsGeneral/FlsUseDmaForRead` is TRUE.*

- *Set ENABLED bit (Bit No.31) in DMAC_CTL register to 1.*
- *Configure the MCU module with `McuDmaEnable=true` and call the `Mcu_SetMode()` function with the configured mode.*

The DMA transfer resolves the following restrictions in the TRAVEO™ T2G microcontroller.

- Work flash is always read 64-bit wide via AXI on CM7_0/CM7_1/CM7_2/CM7_3. It will result in unexpected ECC error for 32-bit reading. DMA reads via AHB which has 32-bit width.
- ECC error can be notified to only one CPU via the fault structure.

When DMA reads from work flash, the ECC error will be detected and informed as bus error via interrupt register of DMA (even in the absence of fault handling). If a separate DMA channel is used for the flash driver on each core, uncorrectable error can be detected for the flash driver on each core.

If you do not detect the ECC error, the configuration parameter `FlsGeneral/FlsUseDmaForRead` can be set to FALSE.

The flash driver prepares the target (auxiliary) buffer for reading with DMA transfer on SRAM. The buffer is located in non-cacheable region by MPU as described in Memory protection unit (MPU). It allows the driver to copy from auxiliary buffer into user's area with keeping cache coherency. The size of auxiliary buffer is determined by the configuration parameter `FlsGeneral/FlsAuxiliaryBufferSize`. The size means maximum size of read data by one DMA transfer, but it affects RAM consumption. You must appropriately configure the size. The auxiliary buffer size is considered to be limited to the value of `FlsConfigSet/FlsMaxReadNormalMode` (or `FlsConfigSet/FlsMaxReadFastMode`) or a large sector size.

7 Appendix A – API reference

7.1 Data types

7.1.1 Flash driver data types

7.1.1.1 Fls_AddressType

Type

```
typedef uint32 Fls_AddressType;
```

Description

This type is used for address information.

7.1.1.2 Fls_LengthType

Type

```
typedef uint32 Fls_LengthType;
```

Description

This type is used for length information.

7.1.1.3 Fls_ConfigType

Type

Hardware specific.

Description

This is the type of the external data structure containing the overall initialization data of the flash driver.

7.1.1.4 External data types

The flash driver imports data types from the MemIf module and AUTOSAR standard data types.

7.1.1.5 Std_ReturnType

Description

AUTOSAR standard API return type.

7.1.1.6 Std_VersionInfoType

Description

This type is used to request the version of the flash driver using the `Fls_GetVersionInfo()` function.

7.1.1.7 MemIf_ModeType

Description

This type denotes the driver operation mode. It is used as the parameter value of the `Fls_SetMode()` function.

7.1.1.8 MemIf_StatusType

Description

This type denotes the current status of the underlying abstraction module and device driver. It is used as the return value of the `Fls_GetStatus()` function.

7.1.1.9 MemIf_JobResultType

Description

This type denotes the result of the last job.

7.2 Macros

7.2.1 Error codes

The service may return the following error codes if default error detection is enabled.

Table 6 Error codes

Name	Value	Description
`FLS_E_PARAM_CONFIG`	0x01	Address of the given configuration for `Fls_Init()` is not within the allowed range.
`FLS_E_PARAM_ADDRESS`	0x02	Address parameter is not within the correct range such as in flash memory area.
`FLS_E_PARAM_LENGTH`	0x03	Length parameter or address + length parameter are not within the correct range.
`FLS_E_PARAM_DATA`	0x04	Address pointer parameter is a NULL pointer.
`FLS_E_UNINIT`	0x05	Flash driver is not yet initialized.
`FLS_E_BUSY`	0x06	Flash driver is currently busy.
`FLS_E_VERIFY_ERASE_FAILED`	0x07	Erase or write operation failed. Data in the affected sector was not erased properly.
`FLS_E_VERIFY_WRITE_FAILED`	0x08	Write operation failed. Data in the affected sector was not written properly.
`FLS_E_TIMEOUT`	0x09	The maximum time has been exceeded during operation or the maximum retry time has been exceeded when there is conflict in flash operation.
`FLS_E_PARAM_POINTER`	0x0a	`Fls_GetVersionInfo()` function called with NULL pointer.
`FLS_E_ERASE_FAILED_FOR_CALLOUT`	0x81	Flash erase failed. This error ID is used to call the error callout handler.

Name	Value	Description
FLS_E_WRITE_FAILED_FOR_CALLOUT	0x82	Flash write failed. This error ID is used to call the error callout handler.
FLS_E_READ_FAILED_FOR_CALLOUT	0x83	Flash read failed. This error ID is used to call the error callout handler.
FLS_E_COMPARE_FAILED_FOR_CALLOUT	0x84	Flash compare failed. This error ID is used to call the error callout handler.
FLS_E_DED_FAILURE	0x85	Double-bit error is detected (DED).
FLS_E_SED_FAILURE	0x86	Single-bit error is detected (SED).

7.2.2 Version information

The following version information is published in the driver's header file:

Table 7 Version information

Name	Value	Description
FLS_SW_MAJOR_VERSION	Refer to release notes	Vendor-specific major version number
FLS_SW_MINOR_VERSION	Refer to release notes	Vendor-specific minor version number
FLS_SW_PATCH_VERSION	Refer to release notes	Vendor-specific patch version number

7.2.3 Module information

Table 8 Module information

Name	Value	Description
FLS_MODULE_ID	92	Module ID
FLS_VENDOR_ID	66	Vendor ID

7.2.4 API service IDs

The following service IDs are used to call the default error tracer in different API functions:

Table 9 API service IDs

Name	Value	API name
FLS_ID_INIT	0x00	Fls_Init()
FLS_ID_ERASE	0x01	Fls_Erase()
FLS_ID_WRITE	0x02	Fls_Write()
FLS_ID_CANCEL	0x03	Fls_Cancel()
FLS_ID_GETSTATUS	0x04	Fls_GetStatus()
FLS_ID_GETJOBRESULT	0x05	Fls_GetJobResult()
FLS_ID_MAINFUNCTION	0x06	Fls_MainFunction()
FLS_ID_READ	0x07	Fls_Read()
FLS_ID_COMPARE	0x08	Fls_Compare()
FLS_ID_SETMODE	0x09	Fls_SetMode()

Name	Value	API name
FLS_ID_BLANKCHECK	0x0A	Fls_BlankCheck()
FLS_ID_GETVERSIONINFO	0x10	Fls_GetVersionInfo()
FLS_ID_SETCYCLEMODE	0xFA	Fls_SetCycleMode()
FLS_ID_READIMMEDIATE	0xFB	Fls_ReadImmediate()
FLS_ID_SUSPEND	0xFC	Fls_Suspend()
FLS_ID_RESUME	0xFD	Fls_Resume()

7.3 Functions

7.3.1 Fls_Init

Syntax

```
void Fls_Init ( const Fls_ConfigType* ConfigPtr )
```

Service ID

0x00

Sync/Async

Synchronous

Reentrancy

Non re-entrant

Parameters (in)

- `ConfigPtr` - pointer to FLS configuration set (Postbuild) or NULL pointer (Precompile).

Parameters (out)

None

Return value

None

Development errors

- `FLS_E_PARAM_CONFIG` - If configuration variant is post-build time, address of the given configuration for Fls_Init() is not within the allowed range where is generated for FlsConfigSet container by EB tresos Studio. If configuration variant is pre-compile time (and there is only one FlsConfiget), pointer other than NULL is passed. Flash driver for application (Fls_TS_T40D13M1I0R0) runs on CM0+ or flash driver for HSM (Fls_TS_T40D13M2I0R0) runs on CM4, CM7_0, CM7_1, CM7_2, or CM7_3.
- `FLS_E_BUSY` - The driver is currently busy.

Runtime errors

None

Description

Flash driver module initialization. This function shall be called with pointer to FLS configuration set (Postbuild) or NULL pointer (Precompile).

Caveats

- This service shall be called before any other service of the flash driver.

7.3.2 Fls_Erase

Syntax

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

Service ID

0x01

Sync/Async

Asynchronous

Reentrancy

Non re-entrant

Parameters (in)

- `TargetAddress` - Virtual target address in flash memory.
- `Length` - Number of bytes to erase.

Parameters (out)

None

Return value

- `E_OK` - Erase command was accepted.
- `E_NOT_OK` - Erase command was not accepted.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_PARAM_ADDRESS` - Parameter `TargetAddress` is greater than the total flash memory size or parameter `TargetAddress` is not aligned to a flash sector boundary.
- `FLS_E_PARAM_LENGTH` - Parameter `Length` is 0 or parameter `TargetAddress` + `Length` is greater than the total flash memory size or the `TargetAddress` parameter + `Length` parameter is not aligned to a flash sector boundary.
- `FLS_E_BUSY` - Driver is currently busy or another job has been already suspended.

Runtime errors

None

Description

Sets up an erase job for the flash driver. The driver will erase the affected sectors that include the area given.

Caveats

- The flash driver must be initialized before this service is called.
- Only one read, write, erase, compare or blank check job can be accepted at the same time.
- An erase job can be accepted only if another job has not been suspended.

7.3.3 Fls_Write

Syntax

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

Service ID

0x02

Sync/Async

Asynchronous

Reentrancy

Non re-entrant

Parameters (in)

- `TargetAddress` - Virtual target address in flash memory.
- `SourceAddressPtr` - Pointer to source data buffer.
- `Length` - Number of bytes to write.

Parameters (out)

None

Return value

- `E_OK` - Write command was accepted.
- `E_NOT_OK` - Write command was not accepted.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_PARAM_ADDRESS` - The `TargetAddress` parameter is greater than the total flash memory size or the `TargetAddress` parameter is not a multiple of FlsPageSize.
- `FLS_E_PARAM_LENGTH` – The `Length` parameter is 0 or the `TargetAddress` parameter + `Length` is greater than the total flash memory size or the `TargetAddress` parameter + `Length` is not a multiple of `FlsPageSize`.
- `FLS_E_PARAM_DATA` - The `SourceAddressPtr` parameter is a NULL pointer.
- `FLS_E_BUSY` - Driver is currently busy or the target address (from start address to end address (start address + length)) is within the range of sector used by suspended job.

Runtime errors

None

Description

Sets up a write job for the flash driver. The driver will write `Length` data from `SourceAddressPtr` to `TargetAddress`.

Caveats

- The flash driver must be initialized before this service is called.
- Only one read, write, erase, compare or blank check job can be accepted at the same time.
- A write job can be accepted only if the target address (from start address to end address (start address + length)) is not within the range of sector used by suspended job.
- If `FlsGeneral/FlsUseNonBlockingWrite` is TRUE, the flash driver writes in non-blocking mode. Otherwise, writes in blocking mode. This parameter is not applied for the write operation for work flash block#1.

7.3.4 Fls_Cancel

Syntax

```
void Fls_Cancel ( void )
```

Service ID

0x03

Sync/Async

Synchronous

Reentrancy

Non re-entrant

Parameters (in)

None

Parameters (out)

None

Return value

None

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_BUSY` – This service was called during a running `Fls_Suspend() or Fls_Resume()` invocation.

Runtime errors

None

Description

This function cancels an ongoing read, write, erase, compare or blank check job immediately. The suspended job can't be cancelled.

Caveats

- The flash driver must be initialized before this service is called.
- The states and data of the affected flash memory cells are undefined.
- This function must only be called from one source (for example, flash EEPROM emulation).
- The FLS module's environment shall not call the function `Fls_Cancel()` during a running `Fls_MainFunction()` invocation.
- The suspended job can't be cancelled. After resume, it can be cancelled.

7.3.5 Fls_GetStatus

Syntax

```
MemIf_StatusType Fls_GetStatus ( void )
```

Service ID

0x04

Sync/Async

Synchronous

Reentrancy

Re-entrant

Parameters (in)

None

Parameters (out)

None

Return value

- `MEMIF_UNINIT` - Driver is not yet initialized.
- `MEMIF_IDLE` - Driver is currently idle.
- `MEMIF_BUSY` - Driver is currently busy.

Development errors

None

Runtime errors

None

Description

This function returns the current state of the driver.

Caveats

- If a job has been suspended and new job doesn't run, this function returns `MEMIF_IDLE`.

7.3.6 Fls_GetJobResult

Syntax

```
MemIf_JobResultType Fls_GetJobResult ( void )
```

Service ID

0x05

Sync/Async

Synchronous

Reentrancy

Re-entrant

Parameters (in)

None

Parameters (out)

None

Return value

- `MEMIF_JOB_OK` - Last job was successful.
- `MEMIF_JOB_PENDING` - Job is currently pending.
- `MEMIF_JOB_FAILED` - Last job was failed.
- `MEMIF_JOB_CANCELED` - Last job was canceled.
- `MEMIF_BLOCK_INCONSISTENT` - Last compare job yielded differences.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.

Runtime errors

None

Description

This function returns the last job result of the driver.

Caveats

- The flash driver must be initialized before this service is called.
- If a job has been suspended and new job doesn't run, this function returns `MEMIF_JOB_OK`.

7.3.7 Fls_Read

Syntax

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

Service ID

0x07

Sync/Async

Asynchronous

Reentrancy

Non re-entrant

Parameters (in)

- `SourceAddress` - Virtual source address in flash memory.
- `Length` - Number of bytes to read.

Parameters (out)

- `TargetAddressPtr` - Pointer to target data buffer.

Return value

- `E_OK` - Read command was accepted.
- `E_NOT_OK` - Read command was not accepted.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_PARAM_ADDRESS` - The `SourceAddress` parameter is greater than the total flash memory size.
- `FLS_E_PARAM_LENGTH` - The `Length` parameter is 0 or the `SourceAddress` parameter + `Length` is greater than the total flash memory size.
- `FLS_E_PARAM_DATA` - The `TargetAddressPtr` parameter is a NULL pointer.
- `FLS_E_BUSY` - Driver is currently busy or the source address (from start address to end address (start address + length)) is within the range of sector used by suspended job.

Runtime errors

None

Description

Sets up a read job for the flash driver. The driver will read the `Length` data from `SourceAddress` to `TargetAddressPtr` (with performing a blank check before reading).

Caveats

- The flash driver must be initialized before this service is called.
- Only one read, write, erase, compare or blank check job can be accepted at the same time.

- A read job can be accepted only if the source address (from start address to end address (start address + length)) is not within the range of sector used by suspended job.
- If blank area is read, this function writes 0xFF.. to target data buffer.

7.3.8 Fls_Compare

Syntax

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

Service ID

0x08

Sync/Async

Asynchronous

Reentrancy

Non re-entrant

Parameters (in)

- `SourceAddress` - Virtual source address in flash memory.
- `TargetAddressPtr` - Pointer to target data buffer.
- `Length` - Number of bytes to compare.

Parameters (out)

None

Return value

- `E_OK` - Compare command was accepted.
- `E_NOT_OK` - Compare command was not accepted.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_PARAM_ADDRESS` - The `SourceAddress` parameter is greater than the total flash memory size.
- `FLS_E_PARAM_LENGTH` - The `Length` parameter is 0 or the `SourceAddress` parameter + `Length` is greater than the total flash memory size.
- `FLS_E_PARAM_DATA` - The `TargetAddressPtr` parameter is a NULL pointer.
- `FLS_E_BUSY` - Driver is currently busy or the source address (from start address to end address (start address + length)) is within the range of sector used by suspended job.

Runtime errors

None

Description

Sets up a compare job for the flash driver. The driver will compare the `Length` data between `SourceAddress` and `TargetAddressPtr`.

Caveats

- The flash driver must be initialized before this service is called.
- Only one read, write, erase, compare or blank check job can be accepted at the same time.
- A compare job can be accepted only if the source address (from start address to end address (start address + length)) is not within the range of sector used by suspended job.
- When reading for the comparing is done, it is performed without blank checking.

7.3.9 Fls_SetMode

Syntax

```
void Fls_SetMode ( MemIf_ModeType Mode )
```

Service ID

0x09

Sync/Async

Synchronous

Reentrancy

Non re-entrant

Parameters (in)

- `Mode` - Mode to set the flash driver to.

Parameters (out)

None

Return value

None

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_BUSY` - Driver is currently busy.

Runtime errors

None

Description

This function sets the flash driver to either SLOW or FAST mode.

Caveats

- The flash driver must be initialized before this service is called.
- This service shall not be called during a running operation.

7.3.10 Fls_GetVersionInfo

Syntax

```
void Fls_GetVersionInfo ( Std_VersionInfoType* VersioninfoPtr )
```

Service ID

0x10

Sync/Async

Synchronous

Reentrancy

Re-entrant

Parameters (in)

None

Parameters (out)

- `VersioninfoPtr` - Pointer to store the version information of this module to.

Return value

None

Development errors

- `FLS_E_PARAM_POINTER` - Parameter `VersionInfoPtr` is a NULL pointer.

Runtime errors

None

Description

This function returns the version information of this module.

Caveats

None

7.3.11 Fls_BlankCheck

Syntax

```
Std_ReturnType Fls_BlankCheck( Fls_AddressType TargetAddress, Fls_LengthType Length )
```

Service ID

0x0A

Sync/Async

Asynchronous

Reentrancy

Non re-entrant

Parameters (in)

- `TargetAddress` - Virtual target address in flash memory.
- `Length` - Number of bytes to be checked blank.

Parameters (out)

None

Return value

- `E_OK` - Blank check command was accepted.
- `E_NOT_OK` - Blank check command was not accepted.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_PARAM_ADDRESS` - The `TargetAddress` parameter is greater than the total flash memory size.
- `FLS_E_PARAM_LENGTH` - The `Length` parameter is 0 or the `TargetAddress` parameter + `Length` is greater than the total flash memory size.
- `FLS_E_BUSY` - Driver is currently busy or the target address (from start address to end address (start address + length)) is within the range of sector used by suspended job.

Runtime errors

None

Description

Sets up a blank check job for the flash driver. The driver will check if the `Length` data from `TargetAddress` is blank (it has been erased but not yet been programmed).

Caveats

- The flash driver must be initialized before this service is called.
- Only one read, write, erase, compare or blank check job can be accepted at the same time.
- A blank check job can be accepted only if the target address (from start address to end address (start address + length)) is not within the range of sector used by suspended job.

7.3.12 Fls_ReadImmediate

Syntax

```
Std_ReturnType Fls_ReadImmediate( Fls_AddressType SourceAddress, uint8*
TargetAddressPtr, Fls_LengthType Length )
```

Service ID

0xFB

Sync/Async

Asynchronous

Reentrancy

Non re-entrant

Parameters (in)

- `SourceAddress` - Virtual source address in flash memory.
- `Length` - Number of bytes to read.

Parameters (out)

- `TargetAddressPtr` - Pointer to target data buffer.

Return value

- `E_OK` - Read immediate command was accepted.
- `E_NOT_OK` - Read immediate command was not accepted.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_PARAM_ADDRESS` - The `SourceAddress` parameter is greater than the total flash memory size.
- `FLS_E_PARAM_LENGTH` - The `Length` parameter is 0 or the `SourceAddress` parameter + `Length` is greater than the total flash memory size.
- `FLS_E_PARAM_DATA` - The `TargetAddressPtr` parameter is a NULL pointer.
- `FLS_E_BUSY` - Driver is currently busy or the source address (from start address to end address (start address + length)) is within the range of sector used by suspended job.

Runtime errors

None

Description

Sets up a read job for the flash driver. The driver will read the `Length` data from `SourceAddress` to `TargetAddressPtr` without performing a blank check before reading.

Caveats

- The flash driver must be initialized before this service is called.
- Only one read, write, erase, compare or blank check job can be accepted at the same time.
- A read job can be accepted only if the source address (from start address to end address (start address + length)) is not within the range of sector used by suspended job.
- If blank area is read, this function will read undefined value.

7.3.13 Fls_Suspend

Syntax

```
Std_ReturnType Fls_Suspend( void )
```

Service ID

0xFC

Sync/Async

Synchronous

Reentrancy

Non re-entrant

Parameters (in)

None

Parameters (out)

None

Return value

- `E_OK` - a write or an erase was in progress and could be suspended.
- `E_NOT_OK` - a write or an erase was in progress and could not be suspended because another job was already suspended or no job operation was in progress.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.

Runtime errors

None

Description

This function suspends a job in progress.

Caveats

- The flash driver must be initialized before this service is called.
- After this service, the FLS module status is MEMIF_IDLE and the job result is `MEMIF_JOB_OK`.
- The FLS module's environment shall not call the function `Fls_Suspend`() during a running `Fls_MainFunction`() or `Fls_Resume`() invocation.
- If this function is called to suspend an erase job that was resumed by `Fls_Resume`(), the call must be done at least 250 microseconds after `Fls_Resume`() finishes. Otherwise, the erase job cannot progress.

7.3.14 Fls_Resume

Syntax

```
Std_ReturnType Fls_Resume( void )
```

Service ID

0xFD

Sync/Async

Synchronous

Reentrancy

Non re-entrant

Parameters (in)

None

Parameters (out)

None

Return value

- `E_OK` - an operation had been suspended and could be resumed.
- `E_NOT_OK` - no job was suspended or the suspended job cannot be resumed because there is a job in progress already.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.

Runtime errors

None

Description

This function resumes a suspended job, erase or write. Only the operation previously suspended can be resumed.

Caveats

- The flash driver must be initialized before this service is called.
- After this service, the FLS module status is MEMIF_BUSY and the job result is `MEMIF_JOB_PENDING`.
- The FLS module's environment shall not call the function `Fls_Resume()` during a running `Fls_Suspend()` invocation.

7.3.15 Fls_SetCycleMode

Syntax

```
void Fls_SetCycleMode ( MemIf_ModeType Mode )
```

Service ID

0xFA

Sync/Async

Synchronous

Reentrancy

Non re-entrant

Parameters (in)

- `Mode` – Indicates whether the flash driver checks timeout. If `Mode` is `MEMIF_MODE_FAST`, timeout monitoring is disabled. Otherwise, timeout monitoring is enabled.

Parameters (out)

None

Return value

- `E_OK` – This setting was accepted.
- `E_NOT_OK` - This setting was not accepted.

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_BUSY` - Driver is currently busy.

Runtime errors

None

Description

This function determines whether the flash driver checks timeout.

Caveats

- The flash driver must be initialized before this service is called.
- This service must not be called during a running operation.

7.4 Scheduled functions

7.4.1 Fls_MainFunction

Syntax

```
void Fls_MainFunction ( void )
```

Service ID

0x06

Timing

FIXED_CYCLIC

Reentrancy

Non re-entrant

Parameters (in)

None

Parameters (out)

None

Return value

None

Development errors

- `FLS_E_UNINIT` - Driver is not yet initialized.
- `FLS_E_VERIFY_ERASE_FAILED` - Verification of erase, before write or blank check yielded a non-erased area. (If `FlsGeneral/FlsReportErrorIfNotBlank` is FALSE, blank check does not report the error.)
- `FLS_E_VERIFY_WRITE_FAILED` - Verification of write yielded incorrect written data.
- `FLS_E_TIMEOUT` - Read, write, erase, compare, or blank check job operation exceeded the maximum timeout or the maximum retry time when there is conflict in flash operation.

Runtime errors

- `FLS_E_ERASE_FAILED`: Erase failed due to a hardware error.
- `FLS_E_WRITE_FAILED`: Write failed due to a hardware error.
- `FLS_E_COMPARE_FAILED`: Compare failed due to a hardware error.
- `FLS_E_READ_FAILED`: Read failed due to a hardware error.
- `FLS_E_DED_FAILURE`: Double bit error is detected (DED).
- `FLS_E_SED_FAILURE`: Single bit error is detected (SED).

Description

This function performs the asynchronous processing of the flash read, write, erase, compare or blank check job.

Caveats

- The flash driver must be initialized before this service is called.

7.5 Expected interfaces

7.5.1 Mandatory interface

There are no mandatory interfaces that is expected by the flash driver.

7.5.2 Optional interfaces

If default error detection is enabled, the flash driver uses the following callback function that is provided by the default error tracer. If the default error tracer is not used, this function must be implemented separately.

7.5.2.1 Det_ReportError

Syntax

```
Std_ReturnType Det_ReportError ( uint16 ModuleId, uint8 InstanceId, uint8 ApiId,
uint8 ErrorId )
```

Sync/Async

Synchronous

Reentrancy

Re-entrant

Parameters (in)

- `ModuleId` - Module ID of the flash driver.
- `InstanceId` - Instance ID of the flash driver.
- `ApiId` - ID of the API service that calls this function.
- `ErrorId` - ID of the detected default error.

Return value

Returns always `E_OK` (is required for services).

Description

Service for reporting default errors.

If runtime error detection is enabled, the flash driver uses the following callback function that is provided by the default error tracer. If the default error tracer is not used, this function must be implemented separately.

7.5.2.2 Det_ReportRuntimeError

Syntax

```
Std_ReturnType Det_ReportRuntimeError ( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId )
```

Sync/Async

Synchronous

Reentrancy

Re-entrant

Parameters (in)

- `ModuleId` - Module ID of the flash driver.
- `InstanceId` - Instance ID of the flash driver.
- `ApiId` - ID of the API service that calls this function.
- `ErrorId` - ID of the detected runtime error.

Return value

Returns always `E_OK` (is required for services).

Description

Service for reporting runtime errors.

7.5.3 Configurable interfaces

The following callback functions are configurable and usually provided by the flash EEPROM emulation.

7.5.3.1 Fee_JobEndNotification

Syntax

```
void Fee_JobEndNotification ( void )
```

Reentrancy

Don't care

Parameters (in)

None

Return value

None

Description

This callback function shall be called when a job has been completed with a positive result:

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

Configurable

If a function name is configured for the `FlsConfigSet/FlsJobEndNotification` parameter, the function is called.

7.5.3.2 Fee_JobErrorNotification

Syntax

```
void Fee_JobErrorNotification ( void )
```

Reentrancy

Don't care

Parameters (in)

None

Return value

None

Description

This callback function shall be called when a job has been canceled or finished with negative result:

- Read job aborted or failed
- Write job aborted or failed
- Erase job aborted or failed

- Compare job aborted or failed
- Compare job finished & memory blocks differ
- Blank check job aborted or failed

Configurable

If a function name is configured for the `FlsConfigSet/FlsJobErrorNotification` parameter, the function is called.

7.5.3.3 Fee_DedErrorNotification

Syntax

```
void Fee_DedErrorNotification ( void )
```

Reentrancy

Don't care

Parameters (in)

None

Return value

None

Description

This callback function is called when 2 bit or more ECC error is detected.

Configurable

If a function name is configured for the `FlsConfigSet/FlsDedErrorNotification` parameter, the function is called.

7.5.3.4 Fee_SedErrorNotification

Syntax

```
void Fee_SedErrorNotification ( void )
```

Reentrancy

Don't care

Parameters (in)

None

Return value

None

Description

This callback function is called when single bit ECC error is detected.

Configurable

If a function name is configured for the `FlsConfigSet/FlsSedErrorNotification` parameter, the function is called.

7.5.3.5 Systemcall callout function

Syntax

```
Std_ReturnType Systemcall_Callout_Function_Name
(
    uint32 *Fls_IpcContext
)
```

Reentrancy

Non re-entrant

Parameters (in)

- `Fls_IpcContext` - SRAM address (SRAM_SCRATCH_ADDR) where the system-call parameters have been stored. This can be used to initiate the system-call request by such S-LLD IPC driver.

Return value

- `E_OK` - The callout function calls system-call successfully.
- `E_NOT_OK` - The callout function fails to call system-call.

Description

The callback function is called whenever the flash driver calls the system-call.

Configurable

If a function name is configured for the `FlsGeneral/FlsSystemcallCalloutFunction` parameter, the function is called.

The following callback function is configurable and usually provided by the user, if required.

7.5.3.6 Erase callout API

Syntax

```
void Erase_Handler_Name
(
    Fls_AddressType TargetAddress
)
```

Reentrancy

Don't care

Parameters (in)

- `TargetAddress` - Virtual target address in flash memory (that was passed to `Fls_Erase`()).

Return value

None

Description

This callback function is called after an erase job is accepted.

Configurable

If a function name is configured for the `FlsGeneral/FlsEraseCalloutFunction` parameter, the function is called.

7.6 Required callback functions

7.6.1 Callout functions

7.6.2 Error callout API

The AUTOSAR FLS module requires an error callout handler. Each error is reported to this handler; error checking cannot be switched off. The name of the function to be called can be configured by the `FlsErrorCalloutFunction` parameter.

Syntax

```
void Error_Handler_Name
(
    uint16 ModuleId,
    uint8 InstanceId,
    uint8 ApiId,
    uint8 ErrorId
)
```

Reentrancy

Re-entrant

Parameters (in)

- `ModuleId` - Module ID of calling module.
- `InstanceId` - Instance ID of calling module.
- `ApiId` - ID of the API service that calls this function.
- `ErrorId` - ID of the detected error.

Return value

None

Description

Service for reporting errors.

8 Appendix B – Access register table

8.1 FLASHC

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
FLASH_CTL	31:0	Word (32 bits)	[FlsSetFlashCtlRegister is FLS_FLASH_CTL_WORK ONLY] 0x00500000 | MAIN_WS[3:0]	Control	Fls_Init	0x0070220F	0x00500000 | MAIN_WS[3:0] (After Fls_Init)
FLASH_PWR_CTL	31:0	Word (32 bits)	-	Flash power Control	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
FLASH_CMD	31:0	Word (32 bits)	0x00000002	Command	Fls_MainFunction Fls_Cancel Fls_Suspend Fls_Resume	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_CA_CTL<n> (<n>=0,1,2)	31:0	Word (32 bits)	-	CM4 cache control	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_CA_STATUS<n> (<n>=0,1,2)	31:0	Word (32 bits)	-	CM4 cache status	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM0_STATUS	31:0	Word (32 bits)	0x00000002	CM0+ interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_STATUS	31:0	Word (32 bits)	0x00000002	CM4 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_0_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #0 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CM7_1_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #1 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_2_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #2 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_3_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #3 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Note: The registers relevant to only DFT (BIST), CM0+, CRYPTO, Datawire, DMAC and external master are omitted from above table.

8.2 FLASHC_FM_CTL_ECT

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
MAIN_FLASH_SAFETY	31:0	Word (32 bits)	-	Main (Code) flash Security enable	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
STATUS	31:0	Word (32 bits)	-	Status read from flash macro	Read-only	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
WORK_FLASH_SAFETY	31:0	Word (32 bits)	[FlsSetWorkFlashSafetyRegister is TRUE] 0x00000001 (Before start of writing, Before start of erasing After Fls_Resume) 0x00000000 (After Fls_Init, After finish of writing, After finish of erasing, After Fls_Suspend)	Work flash security enable	Fls_Init Fls_MainFunction Fls_Suspend Fls_Resume	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Note: The registers used only by System call are omitted from above table.

8.3 FLASHC1

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
FLASH_CTL	31:0	Word (32 bits)	[FlsSetFlashCtlRegister is FLS_FLASH_CTL_WORK ONLY] 0x00500000 | MAIN_WS[3:0]	Control	Fls_Init	0x0070220F	0x00500000 | MAIN_WS[3:0] (After Fls_Init)
FLASH_PWR_CTL	31:0	Word (32 bits)	-	Flash power Control	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
FLASH_CMD	31:0	Word (32 bits)	0x00000002	Command	Fls_MainFunction Fls_Cancel Fls_Suspend Fls_Resume	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_CA_CTL<n> (<n>=0,1,2)	31:0	Word (32 bits)	-	CM4 cache control	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_CA_STATUS<n> (<n>=0,1,2)	31:0	Word (32 bits)	-	CM4 cache status	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM0_STATUS	31:0	Word (32 bits)	0x00000002	CM0+ interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_STATUS	31:0	Word (32 bits)	0x00000002	CM4 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_0_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #0 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_1_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #1 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CM7_2_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #2 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_3_STATUS	31:0	Word (32 bits)	0x00000002	CM7 #3 interface status	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Note: The registers relevant to only DFT (BIST), CM0+, CRYPTO, Datawire, DMAC, and external master. are omitted from above table.

8.4 FLASHC1_FM_CTL_ECT

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
FM_CTL	31:0	Word (32 bits)	0x00000000 (when a flash operation is finished) 0x80000007 (perform flash writing) 0x8000000C (perform sector erasing) 0x8000000D (start blank check) 0x8000000E (perform blank check) 0x8000000F (finish blank check) 0x80000011 (perform erase suspending) 0x80000012 (perform erase resuming)	Flash Macro Control	Fls_MainFunction Fls_Suspend Fls_Resume Fls_Cancel	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
FM_CODE_MARGIN	31:0	Word (32 bits)	-	Flash Macro Margin Mode on Code Flash	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
FM_ADDR	31:0	Word (32 bits)	Target work flash address (before start of writing, before start of erasing and before performing blank check)	Flash Macro Address	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
INTR	31:0	Word (32 bits)	0x00000001 (clear flash#1 macro interrupt)	Interrupt	Fls_MainFunction Fls_Suspend Fls_Resume Fls_Cancel Fls_GetStatusSub Fls_Isr_Flash1_Cat1 Fls_Isr_Flash1_Cat2	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_SET	31:0	Word (32 bits)	-	Interrupt Set	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_MASK	31:0	Word (32 bits)	0x00000001 (enable flash#1 macro interrupt) 0x00000000 (disable flash#1 macro interrupt)	Interrupt Mask	Fls_Init Fls_MainFunction Fls_Suspend Fls_Resume Fls_Cancel Fls_GetStatusSub	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_MASKED	31:0	Word (32 bits)	-	Interrupt Masked	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
ECC_OVERRIDE	31:0	Word (32 bits)	-	ECC Data In override information and control bits	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
FM_DATA	31:0	Word (32 bits)	Flash write value (before start of writing)	Flash macro data_in [31 to 0] both Code and Work Flash	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
BOOKMARK	31:0	Word (32 bits)	Address of sector erase (after start of erasing)	Bookmark register – keeps the current FW HV seq	Fls_MainFunction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
MAIN_FLASH_SAFETY	31:0	Word (32 bits)	-	Main (Code) flash Security enable	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
STATUS	31:0	Word (32 bits)	-	Status read from flash macro	Read-only	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
WORK_FLASH_SAFETY	31:0	Word (32 bits)	[FlsSetWorkFlashSafetyRegister is TRUE] 0x00000001 (Before start of writing, Before start of erasing After Fls_Resume) 0x00000000 (After Fls_Init, After finish of writing, After finish of erasing, After Fls_Suspend)	Work flash security enable	Fls_Init Fls_MainFunction Fls_Suspend Fls_Resume	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.5 FAULT

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CTL	31:0	Word (32 bits)	-	Fault control	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
STATUS	31:0	Word (32 bits)	-	Fault status	Read-only	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
DATA	31:0	Word (32 bits)	-	Fault data	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
PENDING<n> (<n>=0,1,2)	31:0	Word (32 bits)	-	Fault pending	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
MASK0	31:0	Word (32 bits)	-	Fault mask 0	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
MASK1	31:0	Word (32 bits)	[FlsSetWorkFlashFaultMaskRegister is TRUE] 0x00380000 (Fls_TS_T40D13M1I0R0) 0x00000000 (Fls_TS_T40D13M2I0R0)	Fault mask 1	Fls_Init	0x00380000	0x00380000 (Fls_TS_T40D13M1I0R0) 0x00000000 (Fls_TS_T40D13M2I0R0) (After Fls_Init)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
MASK2	31:0	Word (32 bits)	[FlsSetWorkFlashFaultMaskRegister is TRUE and the target device has two flash blocks] 0x00308000 (Fls_TS_T40D13M1I0R0) 0x00000000 (Fls_TS_T40D13M2I0R0)	Fault mask 2	Fls_Init	0x00308000	0x00308000 (Fls_TS_T40D13M1I0R0) 0x00000000 (Fls_TS_T40D13M2I0R0) (After Fls_Init)
INTR	31:0	Word (32 bits)	-	Interrupt	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_SET	31:0	Word (32 bits)	-	Interrupt set	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_MASK	31:0	Word (32 bits)	-	Interrupt mask	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_MASKED	31:0	Word (32 bits)	-	Interrupt masked	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.6 IPC

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
IPC_ACQUIRE	31:0	Word (32 bits)	-	IPC lock acquire register	Read-only	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_RELEASE	31:0	Word (32 bits)	Release event (After finish of writing, After finish of erasing After finish of blank check After Fls_Suspend)	IPC lock release register	Fls_MainFunction Fls_Suspend	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_NOTIFY	31:0	Word (32 bits)	Notification event (When call of System call, When communication with HSM)	IPC notification register	Fls_MainFunction Fls_Cancel Fls_Suspend Fls_Resume Fls_Isr_Flash_Cat1 Fls_Isr_Flash_Cat2	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_DATA0	31:0	Word (32 bits)	Address of SRAM where the System call (API) parameters (When call of System call, When communication with HSM) eCT flash safety mechanism information (for eCT flash safety mechanism)	IPC data register 0	Fls_MainFunction Fls_Cancel Fls_Suspend Fls_Resume Fls_Isr_Flash_Cat1 Fls_Isr_Flash_Cat2	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
IPC_DATA1	31:0	Word (32 bits)	Flash control request to flash driver for HSM (Fls_TS_T40D13M2I0R0)	IPC data register 1	Fls_MainFunction Fls_Cancel Fls_Suspend Fls_Resume Fls_Isr_Flash_Cat1 Fls_Isr_Flash_Cat2	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_LOCK_STATUS	31:0	Word (32 bits)	-	IPC lock status register	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_INTR	31:0	Word (32 bits)	IPC release event clear	IPC interrupt status register	Fls_MainFunction Fls_Cancel Fls_Suspend Fls_Resume Fls_Isr_Ipc_Cat1 Fls_Isr_Ipc_Cat2 Fls_Isr_FlsIpc_Cat1 Fls_Isr_FlsIpc_Cat2	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_INTR_SET	31:0	Word (32 bits)	-	IPC interrupt set register	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_INTR_MASK	31:0	Word (32 bits)	IPC release event mask (Enable interruption) 0x00000000 (Disable interruption)	IPC interrupt mask register	Fls_Init Fls_MainFunction Fls_Cancel Fls_Suspend Fls_Resume	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_INTR_MASKED	31:0	Word (32 bits)	-	IPC masked interrupt register	Not used.	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.7 CPUSS

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
IDENTITY	31:0	Word (32 bits)	-	Identity (Bus master identifier)	Read-only	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.8 M-DMA (DMAC)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CTL	31:0	Word (32 bits)	-	Control	User must set Bit No.31 to 1 (if FlsUseDmaForRead is TRUE)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
ACTIVE	31:0	Word (32 bits)	-	Active channels	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.9 DMAC_CH

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CTL	31:0	Word (32 bits)	0x80000000 (Before start of reading) 0x00000000 (After finish of reading)	Channel control	Fls_MainFunction (if FlsUseDmaFor-Read is TRUE)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IDX	31:0	Word (32 bits)	-	Channel current indices	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
SRC	31:0	Word (32 bits)	-	Channel current source address	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
DST	31:0	Word (32 bits)	-	Channel current destination address	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CURR	31:0	Word (32 bits)	Channel descriptor pointer (Before start of reading) 0x00000000 (After finish of reading)	Channel current descriptor pointer	Fls_MainFunction (if FlsUseDmaFor-Read is TRUE)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
TR_CMD	31:0	Word (32 bits)	0x00000001 (Trigger for reading)	Channel software trigger	Fls_MainFunction (if FlsUseDmaFor-Read is TRUE)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR	31:0	Word (32 bits)	0x000000FF (Clearing) (Before start of reading, After finish of reading)	Interrupt	Fls_MainFunction (if FlsUseDmaFor-Read is TRUE)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
INTR_SET	31:0	Word (32 bits)	-	Interrupt set	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_MASK	31:0	Word (32 bits)	0x00000000 (Before start of reading, After finish of reading)	Interrupt mask	Fls_MainFunction (if FlsUseDmaFor-Read is TRUE)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_MASKED	31:0	Word (32 bits)	-	Interrupt masked	Not used	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Note: *Registers relevant to the channel descriptor are omitted from this table.*

Revision history

Document revision	Date	Description of changes
**	2018-05-17	Initial release
*A	2018-10-04	Added some acronyms relevant to Arm® Cortex®-M7 CPU in Table 1. Glossary. Added two TRAVEO™ T2G Automotive Body Controller High Family TRMs in Hardware Documentation. Added “Platforms” as required modules in 1.5 Development Environment. Added or modified description about API which cannot execute concurrently in followings sections. 5.1.9 Canceling a Job Prior to Maturity 5.1.12 Suspending a Job 5.1.13 Resuming a Suspended Job A.1.3 Functions (Fls_Cancel, Fls_Suspend and Fls_Resume) Added or modified description relevant to Arm® Cortex®-M7 CPU in followings sections. 6.4 IPC 6.5 System Call (NMI exception) B.1.1 FLASHC (CM7_0_STATUS and CM7_1_STATUS)
*B	2018-12-13	Modified the Value and the Monitoing Value of FLASH_CTL register in B.1.1 FLASHC. Modified the Value, the Mask Value and the Monitoing Value of MASK1 register in B.1.3 FAULT.
*C	2019-02-21	Added an acronyms relevant to GHS in Table 1. Glossary. Added description in the case of enabling data cache in following sections. 2.3 Adapting Your Application 2.6 Memory Mapping Changed the Range and the Annotation for following parameters to new minimum and multiple of 4 in 4.2.1 Parameter Constraints. FlsMaxReadFastMode FlsMaxReadNormalMode FlsMaxWriteFastMode FlsMaxWriteNormalMode Added a section 6.6 Memory Protection Unit (MPU) for use of Flash driver in the case of enabling data cache.
*D	2019-06-11	Updated hardware documentation information.
*E	2019-11-14	Changed title to 6.5 System Call. Updated description in 6.5 System Call.
*F	2020-03-08	Added description regarding suspending. 5.1.12 Suspending a Job

Document revision	Date	Description of changes
		A.1.3 Functions (Fls_Suspend)
*G	2020-06-25	Added configuration parameter in 2.2.1 Architecture Details. FlsSetFlashCtlRegister FlsSetWorkFlashSafetyRegister FlsDefineWdgClear Changed description regarding Fls_WdgClear() in 2.2.1 Architecture Details. Added configuration parameter in 4.2.2 Vendor- and Driver-Specific Parametersails. FlsSetFlashCtlRegister FlsUserValueForFlashCtlRegister FlsSetWorkFlashSafetyRegister FlsDefineWdgClear FlsFaultStructure Added description of registers that is set before using the Flash driver in 6.1 Registers. FLASH_CTL WORK_FLASH_SAFETY Added description of fault structure that is specified by the configuration parameter FlsFaultStructure in 6.3 Fault. Added description of condition for setting registers to value. B.1.1 FLASH_CTL B.1.2 FM_CTL_ECT Deleted sentence that SourceAddressPtr must be an address on SRAM. 5.1.5 Writing Data to the Flash Memory A.1.3 Functions (Fls_Write)
*H	2020-09-07	Added Words and Terms in Glossary. DMA HSM Non-blocking mode S-LLD Added 1.7 HSM Support. Added configuration parameter in 2.2.1 Architecture Details. FlsDmaChannel FlsAuxiliaryBufferSize FlsUseNonBlockingWrite FlsArbitrationTimeout FlsSystemcallCalloutFunction Added description of FlsSystemcallCalloutFunction and section FLS_START_SEC_SYSCALLSHARED_VAR_NO_INIT in 2.3 Adapting Your Application.

Document revision	Date	Description of changes
		Changed a memmap file include folder in chapter 2.6. Added configuration parameter in 2.6 Memory Mapping and 6.6 Memory Protection Unit (MPU). FLS_START_SEC_SYSCALLSHARED_VAR_NO_INIT_32 FLS_STOP_SEC_SYSCALLSHARED_VAR_NO_INIT_32 Added description of configuration parameter in 4.2.1 Parameter Constraints. FlsUseInterrupts Added configuration parameter in 4.2.2 Vendor- and Driver-Specific Parametersails. FlsDmaChannel FlsAuxiliaryBufferSize FlsUseNonBlockingWrite FlsArbitrationTimeout FlsSystemcallCalloutFunction Added description of configuration parameter in 4.2.2 Vendor- and Driver-Specific Parametersails. FlsUseSafetyMechanism FlsIpcStructure FlsIpcInterruptStructure Added description regarding DMA transfer. 5.1.4 Reading Data from the Flash Memory 5.1.5 Writing Data to the Flash Memory 5.1.7 Comparing Data from the Flash Memory Added description regarding Fls_TS_T40D13M2I0R0 (Flash driver for HSM). 5.1.4 Reading Data from the Flash Memory 5.1.7 Comparing Data from the Flash Memory Added description regarding conflict of flash operation. 5.1.4 Reading Data from the Flash Memory 5.1.5 Writing Data to the Flash Memory 5.1.6 Erasing Data from the Flash Memory 5.1.7 Comparing Data from the Flash Memory 5.1.8 Checking Blank for the Flash Memory Added description regarding non-blocking mode. 5.1.5 Writing Data to the Flash Memory Added note. 5.1.12 Suspending a Job 5.1.13 Resuming a Suspended Job 6.2 Interrupts 6.3 Fault 6.5 System Call

Document revision	Date	Description of changes
		Added description of Conflict of flash operation in 5.1.14 Timeout Supervision. Changed example in 6.3 Fault. Added description regarding CM0+ and restriction in 6.4 IPC. Added description regarding FlsSystemcallCalloutFunction in 6.4 IPC. Added 6.7 DMA. Added description reagrding maximum retry time in Error Codes, A.1.2 Macros. Added Development Errors in A.1.3 Functions. Fls_Init Added Caveats in A.1.3 Functions. Fls_Write Added Development Errors in A.1.4 Scheduled Functions. Fls_MainFunction Added Configurable Interfaces in A.1.5 Expected Interfaces. Systemcall callout function Added description of value for setting registers to value. B.1.3 FAULT Added B.1.6 M-DMA (DMAC) Added B.1.7 DMAC_CH
*I	2020-11-20	Added configuration parameter in 2.2.1 Architecture Details. FlsUseDmaForRead FlsSetCycleModeApi Added configuration parameter in 4.2.2 Vendor- and Driver-Specific Parametersails. 4.2.2.1.24 FlsSetCycleModeApi 4.2.2.1.25 FlsUseDmaForRead Added description of configuration parameter in 4.2.2 Vendor- and Driver-Specific Parametersails. 4.2.2.1.9 FlsDmaChannel 4.2.2.1.10 FlsAuxiliaryBufferSize 4.2.2.2.3 FlsNumberOfDelayLoop Added description regarding condition for DMA transfer. 5.1.4 Reading Data from the Flash Memory 5.1.5 Writing Data to the Flash Memory 5.1.7 Comparing Data from the Flash Memory 6.7 DMA Added description regarding the disabling of timeout monitoring. 5.1.14 Timeout Supervision Added function in 7.1.3 Functions. 7.1.3.15 Fls_SetCycleMode Added description regarding the Fls_SetCycleMode function.

Document revision	Date	Description of changes
		7.1.2.4 API Service IDs Added description regarding condition for setting registers condition. 8.1.6 M-DMA (DMAC) 8.1.7 DMAC_CH Fixed typo. 5.4 Runtime Error Detection Updated to Infineon template.
*J	2020-11-25	Updated 6.1 Registers.
*K	2021-03-01	Added configuration parameter in 2.2.1 Architecture Details. FlsSetWorkFlashFaultMaskRegister FlsReportErrorIfNotBlank Added description regarding enabling DMA controller in 2.3 Adapting Your Application. Added configuration parameter in 4.2.2 Vendor- and Driver-Specific Parametersails. 4.2.2.1.26 FlsSetWorkFlashFaultMaskRegister 4.2.2.1.27 FlsReportErrorIfNotBlank Deleted description regarding other error and added description regarding FlsReportErrorIfNotBlank in 5.1.8 Checking Blank for the Flash Memory. Added 4. DMAC_CTL register in 6.1 Registers. Added description regarding FlsSetWorkFlashFaultMaskRegister in 6.2 Interrupts. Added note regarding use of both Flash drivers and modified example in 6.3 Fault. Added note regarding enabling DMA controller in 6.7 DMA. Added description regarding FlsReportErrorIfNotBlank for FLS_E_VERIFY_ERASE_FAILED in 7.4.1 Fls_MainFunction Modified Value of MASK1 register in 8.3 FAULT. Modified Value and Timing of CTL register in 8.6 M-DMA (DMAC).
*L	2021-05-18	Modified Note in 5.1.12 Suspending a Job. Modified Note in 5.1.13 Resuming a Suspended Job.
*M	2021-08-19	Added a note in 6.2 Interrupts.
*N	2021-12-07	Updated to the latest branding guidelines.
*O	2022-09-28	Added “Data buffer” in Abbreviation. Modified Annotation in 4.2.1.1.12 FlsTotalSize.
*P	2022-12-13	Modified Value and Timing of the IPC_RELEASE register in 8.6 IPC.
*Q	2023-03-03	Added terms in Abbreviation. CM7_2 CM7_3 Work flash block#0

Document revision	Date	Description of changes
		Work flash block#1 Added CM7_2 and CM7_3 in 1.7 HSM support, 6.6 Memory protection unit (MPU), 6.7 DMA, 7.3.1 Fls_Init. Added 5.3 Parallel flash operations for separate work flash memories. Added FLASHC1 registers in 6.1 Registers. Added flash#1 macro interrupt and ISRs in 6.2 Interrupts. Added IPC resources for CM7_2 and CM7_3 in 6.4 IPC. Added the description for work flash blocks in 6.5 System call. Added CM7_2_STATUS and CM7_3_STATUS registers in 8.1 FLASHC. Changed the section name of 8.2 FLASHC_FM_CTL_ECT to distinguish two FM_CTL_ECT registers. Added 8.3 FLASHC1 and 8.4 FLASHC1_FM_CTL_ECT.
*R	2023-05-31	Modified the description of `FlsSetWorkFlashFaultMaskRegister` in 2.2.1 Architecture details and 4.2.2.1.28 FlsSetWorkFlashFaultMaskRegister. Modified the description of `FlsUseNonBlockingWrite` in 2.2.1 Architecture details, 4.2.2.1.15 FlsUseNonBlockingWrite, 5.1.5 Writing data to the flash memory and 7.3.3 Fls_Write. Added the usecase for FLASHC1 operations from multiple cores in 5.3 Parallel flash operations for separate work flash memories. Modified IPC resources for CM7_2 and CM7_3 in 6.4 IPC. Modified MASK2 register in 8.5 FAULT.
*S	2023-10-06	Added the annotations in 4.2.2.1.18 FlsIpcStructure and 4.2.2.1.19 FlsIpcInterruptStructure. Added the configuration parameters 4.2.2.1.20 FlsIpcReleaseEventNotification and 4.2.2.1.21 FlsIpcNotificationEventToHsm Added the note for nested erase suspend in 5.1.12 Suspending a job. Added the description for eCT flash safety mechanism in 5.1.15 eCT flash safety mechanism. Modified DATA0 and DATA1 registers in 8.6 IPC.
*T	2023-12-08	Web release. No content updates.
*U	2025-04-24	Added a note in section 2.6.1 Memory allocation keyword.
*V	2025-08-12	Added detailed description about the dummy read and the error detection with Fault interrupt into section 6.3 Fault.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2025-08-12

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-23407 Rev. *V

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”).

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.