

Code Flash driver user guide

TRAVEO™ T2G family

About this document

Scope and purpose

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

The installation, the 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* [1] for detailed information of these topics.

Intended audience

This document is intended for anyone using the Code Flash driver software.

Document structure

The [1 General overview](#) chapter gives a brief introduction to the Code Flash driver, explains the embedding in the AUTOSAR environment, and describes the supported hardware and development environment.

The [2 Using the Code Flash driver](#) chapter details the steps required to use the Code Flash driver in your application.

The [3 Structure and dependencies](#) chapter describes the file structure and the dependencies for the Code Flash driver.

The [4 EB tresos Studio configuration interface](#) chapter describes the configuration of the Code Flash driver.

The [5 Functional description](#) chapter gives a functional description of all services offered by the Code Flash driver.

The [6 Hardware resources](#) chapter gives a description of all hardware resources used.

The [Appendix A](#) and [Appendix B](#) chapters provides a complete API reference and access register table.

Abbreviations and definitions

Table 1 **Abbreviations**

Abbreviations	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
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
CM7_3	Arm® Cortex®-M7 CPU fourth core

About this document

Abbreviations	Description
ECU	Engine Control Unit
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
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.
IPC	Inter Processor Communication
MCU	Microcontroller Unit
MPU	Memory Protection Unit
16M	TRAVEO™ T2G" microcontroller with 16 MB flash memory, and the flash area is divided into Block#0 and Block#1.

Related documents

Elektrobit automotive documentation

Bibliography

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

Hardware documentation

The hardware documents are listed in the delivery notes.

Table of contents

Table of contents

About this document	1
Table of contents	3
1 General overview	5
1.1 Introduction to the Code Flash driver	5
1.2 User profile	5
1.3 Supported hardware	5
1.4 Development environment	5
1.5 Character set and encoding	5
2 Using the Code Flash driver	6
2.1 Installation and prerequisites	6
2.2 Configuring the Code Flash driver	7
2.2.1 Configuration details	7
2.2.1.1 Container CFlsGeneral	7
2.2.1.2 Container CFlsCodeflashSectorList	8
2.2.1.3 Container CFlsSector, CFlsSectorDualFirst, CFlsSectorDualSecond, CFlsSector16M, CFlsSectorDualFirst16M, CFlsSectorDualSecond16M	8
2.3 Adapting your application	14
2.4 Starting the build process	16
2.5 Memory mapping	16
2.5.1 Memory allocation keyword	16
3 Structure and dependencies	18
3.1 Static files	18
3.2 Configuration files	18
3.3 Generated files	18
4 EB tresos Studio configuration interface	19
5 Functional description	20
5.1 Function of the Code Flash driver	20
5.1.1 Writing data to the flash memory	20
5.1.2 Erasing data from the flash memory	21
5.1.3 Retrieving the status information	22
5.1.4 Canceling a job prior to maturity	22
5.1.5 Timeout supervision	22
5.2 Re-entrancy	22
6 Hardware resources	23
6.1 Registers	23
6.2 Interrupts	23
6.3 Fault	23
6.4 IPC	24
6.5 System call	24
6.6 Function table	25
6.7 Memory protection unit (MPU)	25
7 Appendix A – API reference	28
7.1 Data types	28
7.1.1 Code Flash driver data types	28
7.1.1.1 CFls_OperationStatusType	28
7.1.1.2 CFls_SchModeType	28

Table of contents

7.1.1.3	CFls_ConfigType	28
7.1.1.4	CFls_SectorListType	28
7.2	Macros.....	29
7.2.1	Configuration parameters	29
7.2.2	Resource information	29
7.2.3	Configuration information.....	29
7.3	Functions	30
7.3.1	CFls_EraseBlock	30
7.3.2	CFls_Erase	31
7.3.3	CFls_Write.....	32
7.3.4	CFls_GetOperationStatus	33
7.3.5	CFls_GetOperationStatus2nd.....	34
7.3.6	CFls_Cancel	35
7.3.7	CFls_Cancel2nd.....	36
7.4	Required callback functions	36
7.4.1	Callout functions	36
7.4.1.1	CFls_WdgClear	36
7.4.1.2	CFls_GetCounterValue	37
7.4.1.3	CFls_GetElapsedValue	37
8	Appendix B – Access register table	38
8.1	FLASHC	38
8.2	FM_CTL_ECT(FLASHC)	39
8.3	FLASHC1	40
8.4	FM_CTL_ECT(FLASHC1)	41
8.5	FAULT	42
8.6	IPC	42
8.7	CPUSS	43
	Revision history	44
	Disclaimer	46

1 General overview

1 General overview

1.1 Introduction to the Code Flash driver

The Code Flash driver abstracts the hardware internal flash controller of the TRAVEO™ T2G microcontroller and provides API functions for writing and erasing the code 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 Supported hardware

This version of the Code Flash driver supports the TRAVEO™ T2G microcontroller family. No further special external hardware devices are required.

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.

1.4 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 Code Flash driver.

1.5 Character set and encoding

All source code files of the Code 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.

2 Using the Code Flash driver

2 Using the Code Flash driver

This chapter describes the necessary steps to incorporate the Code Flash driver in your application.

2.1 Installation and prerequisites

Note: Before you start, see the *EB tresos Studio for ACG8 user's guide* [1] for the following information.

1. The installation procedure of EB tresos ECU AUTOSAR components.
2. The usage of the EB tresos Studio software.
3. The usage of the EB tresos ECU AUTOSAR build environment (includes an explanation on how to set up and integrate your application within the EB tresos ECU AUTOSAR build environment).

The installation of the Code Flash driver complies with the general installation procedure for EB tresos ECU AUTOSAR components given in these above-mentioned documents. If the driver is successfully installed, the driver will appear **CFIs** under Available Modules in EB tresos Studio.

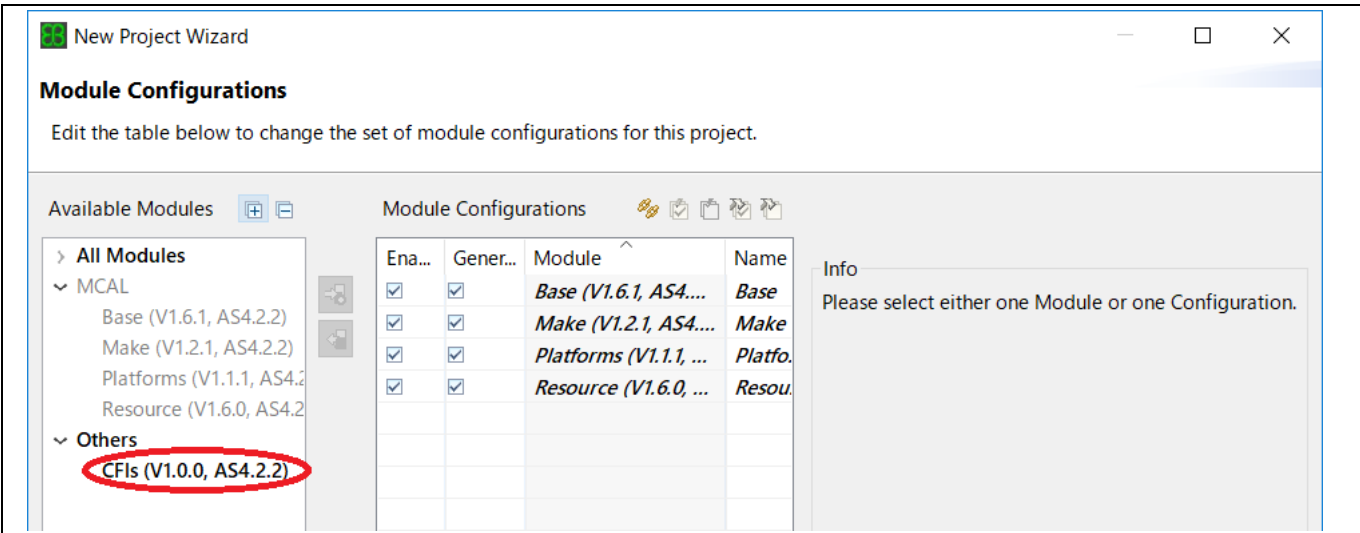


Figure 1 Code Flash driver in MCAL layer

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* [1]. 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.

Note: The configuration that is manually done without the EB tresos Studio is not supported.

2 Using the Code Flash driver

2.2 Configuring the Code Flash driver

This section provides an overview of the configuration structure defined to use the Code Flash driver.

The following two containers are used to configure the Code Flash driver. The parameters are always pre-compiled:

1. `CFlsGeneral`: Container for general parameters of Infineon Code Flash driver and is labeled as General.
2. `CFlsCodeflashSectorList`: Container for list of code flash sectors and pages (Code Flash), which is labeled as "CFls Sector List (Code Flash)", "CFls Sector Dual First half List", "CFls Sector Dual Second half List", "[16M] CFls Sector List (Code Flash)", "[16M] CFls Sector Dual First half List", and "[16M] CFls Sector Dual Second half List". This container has a child container `CFlsSector`, `CFlsSectorDualFirst`, `CFlsSectorDualSecond`, `CFlsSector16M`, `CFlsSectorDualFirst16M`, and `CFlsSectorDualSecond16M`.

See [4 EB tresos Studio configuration interface](#) chapter for details on GUI of the EB tresos Studio.

2.2.1 Configuration details

The following configuration parameters can be specified by the EB tresos Studio.

2.2.1.1 Container `CFlsGeneral`

2.2.1.1.1 `CFlsCodeflashTimeoutForWait`

Name

`CFlsCodeflashTimeoutForWait`

Range

0..4294967294

Annotation

Specifies the time-out for wait.

2.2.1.1.2 `CFlsCodeflashWdgclearPeriod`

Name

`CFlsCodeflashWdgclearPeriod`

Range

0..4294967294

Annotation

Specifies the frequency at which CFls calls `CFls_WdgClear(void)`.

2 Using the Code Flash driver

2.2.1.1.3 CFlsEnableDataCache

Name

CFlsEnableDataCache

Range

TRUE, FALSE

Annotation

Indicates whether data cache on MCU is enabled. If the data cache on MCU is enabled, this parameter must be set to true.

2.2.1.2 Container CFlsCodeflashSectorList

This container specifies the list of code flash sectors to be used and the list consists of multiple containers of CFlsSector, CFlsSectorDualFirst, CFlsSectorDualSecond, CFlsSector16M, CFlsSectorDualFirst16M, and CFlsSectorDualSecond16M. CFlsSector and CFlsSector16M support single bank mode. CFlsSectorDualFirst, CFlsSectorDualSecond, CFlsSectorDualFirst16M and CFlsSectorDualSecond16M support dual bank mode.

2.2.1.3 Container CFlsSector, CFlsSectorDualFirst, CFlsSectorDualSecond, CFlsSector16M, CFlsSectorDualFirst16M, CFlsSectorDualSecond16M

The following is an overview of the uses of each container.

CFlsSector: This is for derivatives with a single bank mode and only one code flash area (other than 16M).

CFlsSectorDualFirst: This is for derivatives that are the first half of dual bank mode and have only one code flash area (other than 16M).

CFlsSectorDualSecond: This is for derivatives that are the second half of dual bank mode and have only one code flash area (other than 16M).

CFlsSector16M: This is for derivatives with two code flash areas in single bank mode (Block#1 and Block#1 in 16M).

CFlsSectorDualFirst16M: This is for derivatives with the first half of dual bank mode and two code flash areas (Block#1 and Block#1 in 16M).

CFlsSectorDualSecond16M: This is for derivatives with the second half of dual bank mode and two code flash areas (Block#1 and Block#1 in 16M).

2.2.1.3.1 CFlsSectorIdentifier

Name

CFlsSectorIdentifier

Range

Selectable list entry

Annotation

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

2 Using the Code Flash driver

2.2.1.3.2 CFlsSectorNumber

Name

CFlsSectorNumber

Range

0..*

Annotation

Unique number of this sector.

2.2.1.3.3 CFlsNumberOfSectors

Name

CFlsNumberOfSectors

Range

1..*

Annotation

Number of continuous identical flash sectors (with identical values for CFlsSectorSize and CFlsPageSize). The maximum value depends on subderivative.

2.2.1.3.4 CFlsIsProtected

Name

CFlsIsProtected

Range

TRUE, FALSE

Annotation

Indicates whether this sector is protected from Erase. If this parameter is true, the sector is protected from Erase.

2.2.1.3.5 CFlsSectorSize

Name

CFlsSectorSize

Range

32768 (large sector) or 8192 (small sector)

Annotation

Indicates the size of this sector in bytes.

2 Using the Code Flash driver

2.2.1.3.6 CFlsSectorStartaddress

Name

CFlsSectorStartaddress

Range

Automated calculation

Annotation

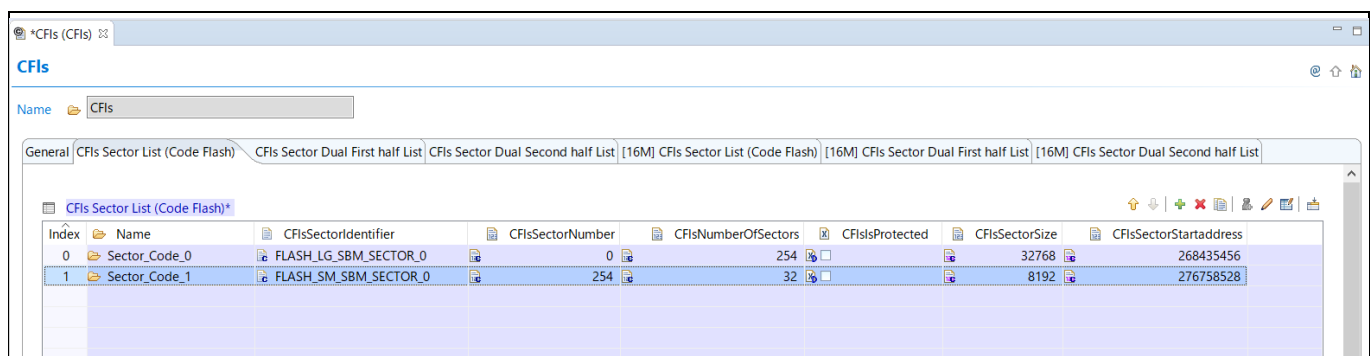
Indicates the start address of this sector.

While using the whole code flash sectors, if the sectors are not protected from erase, click the + button twice from the **CFls Sector List (Code Flash)** tab as shown in Figure 2. Figure 2 represents all (254) large sectors and all (32) small sectors that are configured.

Note: Configure the entire code flash sector, regardless of whether you use the whole sector.

Note: "CFls Sector List (Code Flash)" tab, "CFls Sector Dual First half List" tab, and "CFls Sector Dual Second half List" tab are the sector lists for derivatives with only one flash area. If a derivative with two flash areas is selected, then (Block#1 and Block#1 in 16M) "CFls Sector List (Code Flash)" tab, "CFls Sector Dual First half List" tab, and "CFls Sector Dual Second half List" tab cannot be entered.

Note: "[16M] CFls Sector List (Code Flash)" tab, "[16M] CFls Sector Dual First half List" tab, and "[16M] CFls Sector Dual Second half List" tab are the sector lists for derivatives with two code flash areas (Block#1 and Block#1 in 16M). If a derivative with only one flash area is selected, then "[16M] CFls Sector List (Code Flash)" tab, "[16M] CFls Sector Dual First half List" tab, and "[16M] CFls Sector Dual Second half List" tab cannot be entered.



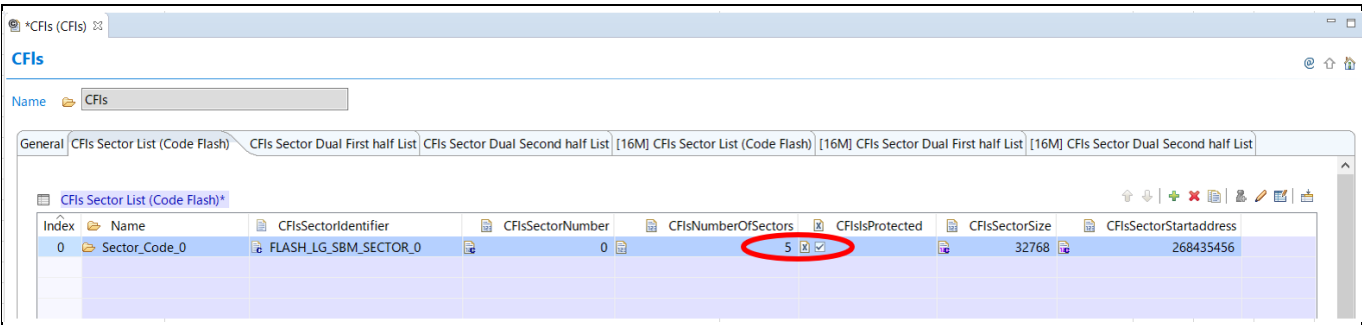
Index	Name	CFlsSectorIdentifier	CFlsSectorNumber	CFlsNumberOfSectors	CFlsProtected	CFlsSectorSize	CFlsSectorStartaddress
0	Sector_Code_0	FLASH_LG_SBM_SECTOR_0	0	254	<input type="checkbox"/>	32768	268435456
1	Sector_Code_1	FLASH_SM_SBM_SECTOR_0	254	32	<input type="checkbox"/>	8192	276758528

Figure 2 Configuration for all large sectors and all small sectors

If you protect some sectors from erase, for example if the top five large sectors and a middle small sector are protected, following operation should be done.

1. After clicking the + button, change **CFlsNumberOfSectors** to **5** and select the option to enable **CFlsProtected** for their sectors. Then, click the + button twice.

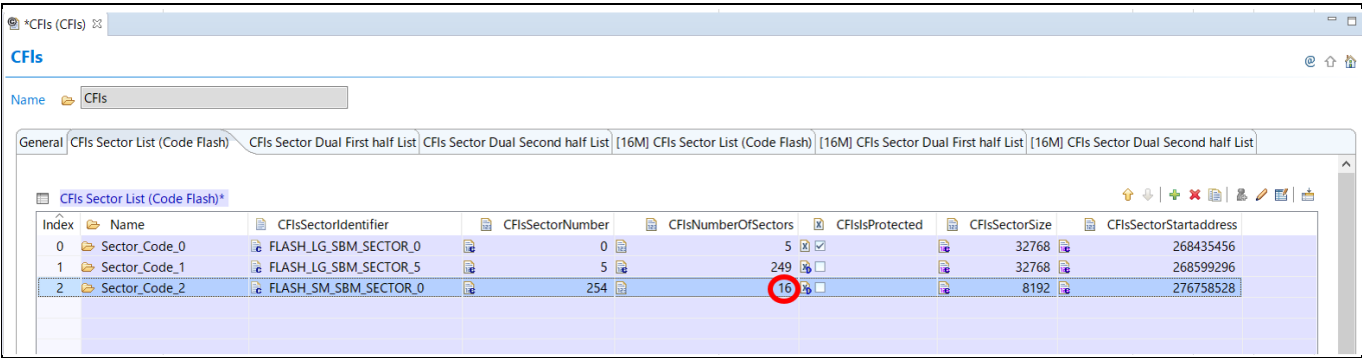
2 Using the Code Flash driver



Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_0	FLASH_LG_SBM_SECTOR_0	0	5	<input checked="" type="checkbox"/>	32768	268435456

Figure 3 Configuration for protecting the top five large sectors and a middle small sector (1)

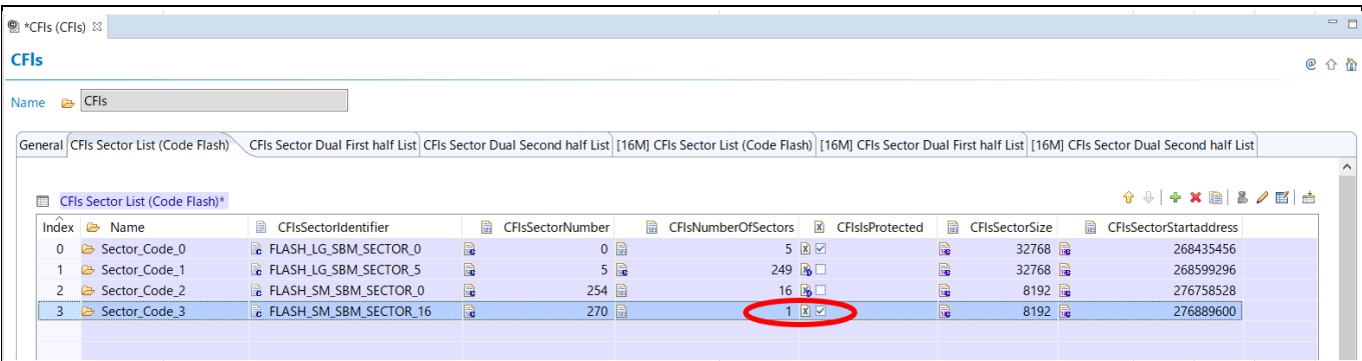
2. Change **CFIsNumberOfSectors** to **16**, and click the + button once.



Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_0	FLASH_LG_SBM_SECTOR_0	0	5	<input checked="" type="checkbox"/>	32768	268435456
1	Sector_Code_1	FLASH_LG_SBM_SECTOR_5	5	249	<input checked="" type="checkbox"/>	32768	268599296
2	Sector_Code_2	FLASH_SM_SBM_SECTOR_0	254	16	<input type="checkbox"/>	8192	276758528

Figure 4 Configuration for protecting the top five large sectors and a middle small sector (2)

3. Change **CFIsNumberOfSectors** to **1** and select the option to enable **CFIsProtected** for their sectors. Click the + button.

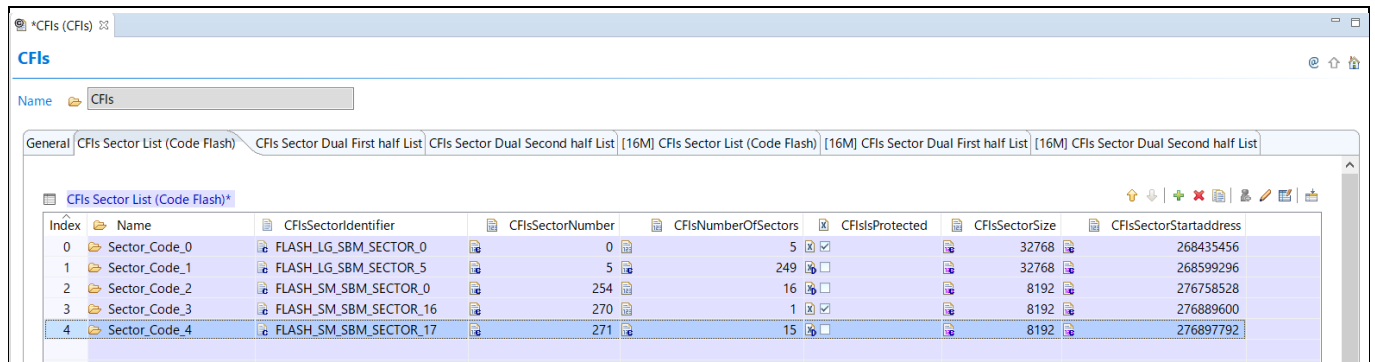


Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_0	FLASH_LG_SBM_SECTOR_0	0	5	<input checked="" type="checkbox"/>	32768	268435456
1	Sector_Code_1	FLASH_LG_SBM_SECTOR_5	5	249	<input checked="" type="checkbox"/>	32768	268599296
2	Sector_Code_2	FLASH_SM_SBM_SECTOR_0	254	16	<input type="checkbox"/>	8192	276758528
3	Sector_Code_3	FLASH_SM_SBM_SECTOR_16	270	1	<input checked="" type="checkbox"/>	8192	276889600

Figure 5 Configuration for protecting the top five large sectors and a middle small sector (3)

The sectors of code flash are configured completely.

2 Using the Code Flash driver

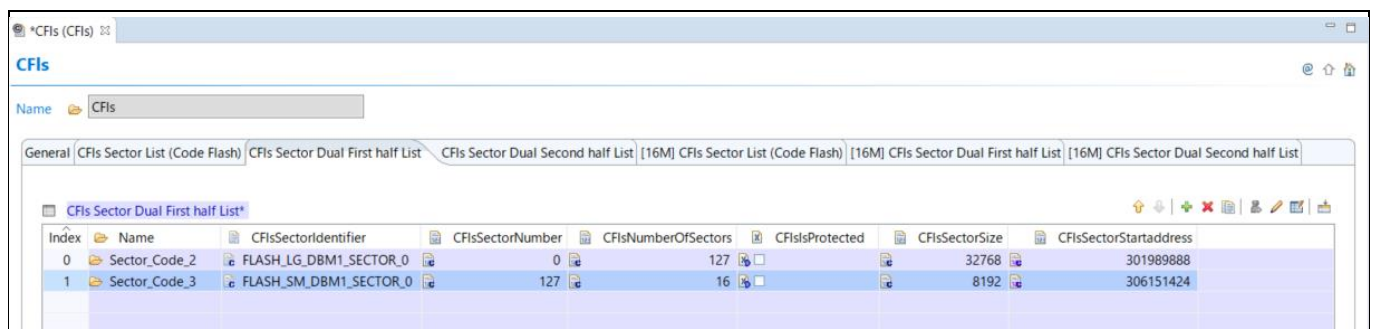


Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_0	FLASH_LG_SBM_SECTOR_0	0	5	<input checked="" type="checkbox"/>	32768	268435456
1	Sector_Code_1	FLASH_LG_SBM_SECTOR_5	5	249	<input type="checkbox"/>	32768	268599296
2	Sector_Code_2	FLASH_SM_SBM_SECTOR_0	254	16	<input type="checkbox"/>	8192	276758528
3	Sector_Code_3	FLASH_SM_SBM_SECTOR_16	270	1	<input checked="" type="checkbox"/>	8192	276889600
4	Sector_Code_4	FLASH_SM_SBM_SECTOR_17	271	15	<input type="checkbox"/>	8192	276897792

Figure 6 Configuration for protecting the top five large sectors and a middle small sector (4)

"CFIs Sector Dual First half List" tab and "CFIs Sector Dual Second half List" tab input method is the same as "CFIs Sector List (Code Flash)" tab.

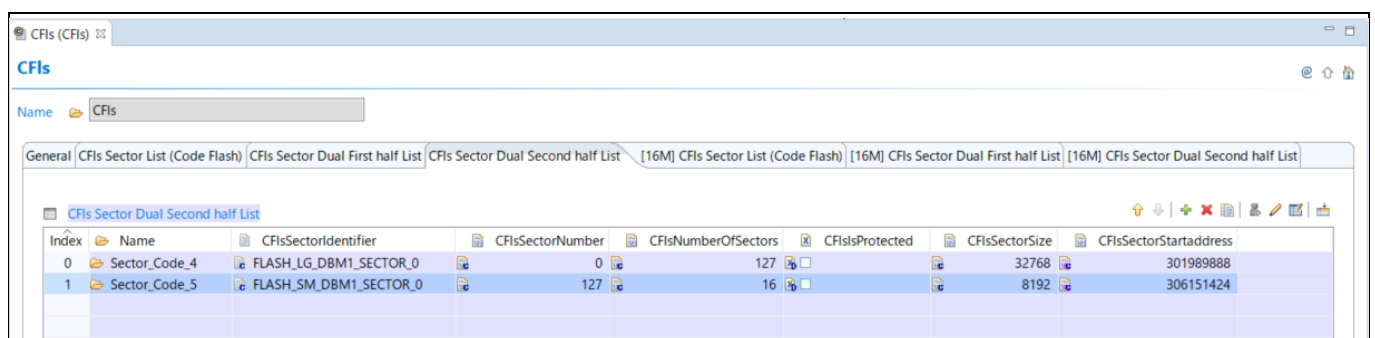
While using the whole code flash sectors, if the sectors are not protected from erase, click the + button twice from the **CFIs Sector List Dual First half** tab as shown in Figure 7. Figure 7 represents all (127) large sectors and all (16) small sectors that are configured.



Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_2	FLASH_LG_DBM1_SECTOR_0	0	127	<input type="checkbox"/>	32768	301989888
1	Sector_Code_3	FLASH_SM_DBM1_SECTOR_0	127	16	<input type="checkbox"/>	8192	306151424

Figure 7 Configuration for all large sectors and all small sectors of dual first half

While using the whole code flash sectors, if the sectors are not protected from erase, click the + button twice from the **CFIs Sector List Dual First half** tab as shown in Figure 8. Figure 8 represents all (127) large sectors and all (16) small sectors that are configured.



Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_4	FLASH_LG_DBM1_SECTOR_0	0	127	<input type="checkbox"/>	32768	301989888
1	Sector_Code_5	FLASH_SM_DBM1_SECTOR_0	127	16	<input type="checkbox"/>	8192	306151424

Figure 8 Configuration for all large sectors and all small sectors of dual second half

2 Using the Code Flash driver

For derivatives with two code flash areas (Block#0 and Block#1 in 16M), use the "[16M] CFIs Sector List (Code Flash)" tab, the "[16M] CFIs Sector Dual First half List" tab, and the "[16M] CFIs Sector Dual Second half List" tab.

While using the whole code flash sectors, if the sectors are not protected from erase, click the + button four times from the **[16M] CFIs Sector List (Code Flash)** tab as shown in Figure 9. Figure 9 represents all (Block#0=254 + Block#1=254) large sectors and all (Block#0=32 + Block#1=32) small sectors that are configured.

Index 0 to Index 1 is the sector list for Block#0, and Index 2 to Index 3 is the sector list for Block#1.

Note: Configure the entire code flash sector, regardless of whether you use the whole sector.

Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_0	FLASH_LG_SBM_SECTOR_0	0	254	<input type="checkbox"/>	32768	268435456
1	Sector_Code_1	FLASH_SM_SBM_SECTOR_0	254	32	<input type="checkbox"/>	8192	276758528
2	Sector_Code_2	FLASH1_LG_SBM_SECTOR_0	286	254	<input type="checkbox"/>	32768	402653184
3	Sector_Code_3	FLASH1_SM_SBM_SECTOR_0	540	32	<input type="checkbox"/>	8192	410976256

Figure 9 Configuration for all large sectors and all small sectors (for 16M: Block#0 and Block#1)

Sector protection is the same as for "CFIs Sector List (Code Flash)," but it must operate within the same Block. The execution result is shown in Figure 10.

Index 0 to Index 4 is the sector list for Block#0, and Index 5 to Index 9 is the sector list for Block#1.

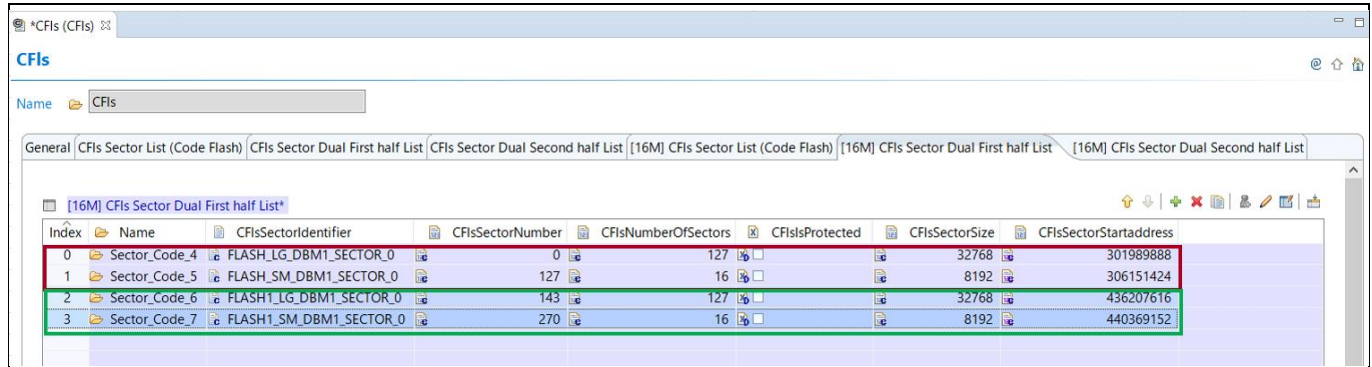
Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_0	FLASH_LG_SBM_SECTOR_0	0	5	<input checked="" type="checkbox"/>	32768	268435456
1	Sector_Code_1	FLASH_LG_SBM_SECTOR_5	5	249	<input type="checkbox"/>	32768	268599296
2	Sector_Code_2	FLASH_SM_SBM_SECTOR_0	254	16	<input type="checkbox"/>	8192	276758528
3	Sector_Code_3	FLASH_SM_SBM_SECTOR_16	270	1	<input checked="" type="checkbox"/>	8192	276889600
4	Sector_Code_8	FLASH_SM_SBM_SECTOR_17	271	15	<input type="checkbox"/>	8192	276897792
5	Sector_Code_9	FLASH1_LG_SBM_SECTOR_0	286	5	<input checked="" type="checkbox"/>	32768	402653184
6	Sector_Code_10	FLASH1_LG_SBM_SECTOR_5	291	249	<input type="checkbox"/>	32768	402817024
7	Sector_Code_11	FLASH1_SM_SBM_SECTOR_0	540	16	<input type="checkbox"/>	8192	410976256
8	Sector_Code_16	FLASH1_SM_SBM_SECTOR_16	556	1	<input checked="" type="checkbox"/>	8192	411107328
9	Sector_Code_17	FLASH1_SM_SBM_SECTOR_17	557	15	<input type="checkbox"/>	8192	411115520

Figure 10 Configuration for protecting the top five large sectors and a middle small sector of Block#0 and configuration for protecting the top five large sectors and a middle small sector of Block#1 (for 16M: Block#0 and Block#1)

The input method for "[16M] CFIs Sector Dual First half List" and "[16M] CFIs Sector Dual Second half List" is the same as that for "CFIs Sector List (Code Flash)" and "[16M] CFIs Sector List (Code Flash)".

2 Using the Code Flash driver

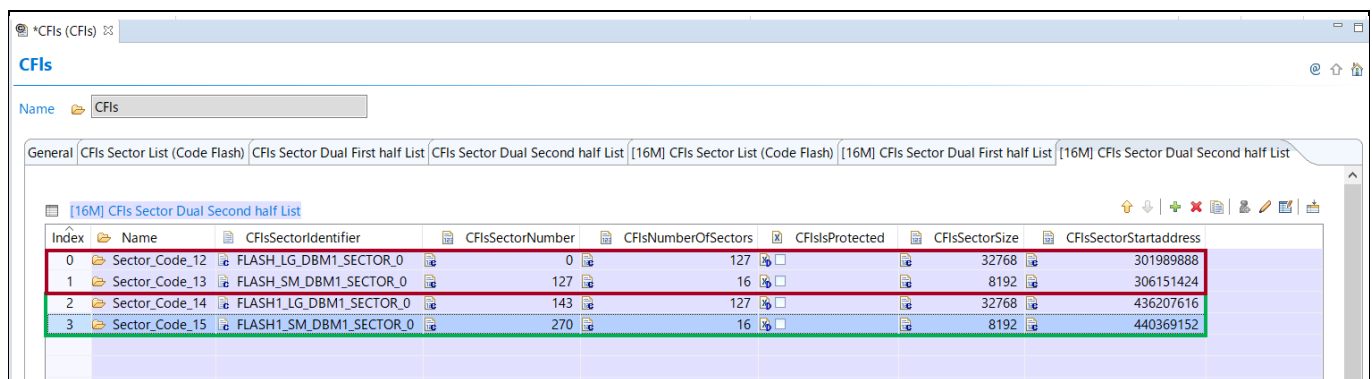
While using the whole code flash sectors, if the sectors are not protected from erase, click the + button four times from the **[16M] CFIs Sector List Dual First half** tab as shown in Figure 11. Figure 11 represents all (Block#0=127 + Block#1=127) large sectors and all (Block#0=16 + Block#1=16) small sectors that are configured.



Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_4	FLASH_LG_DBM1_SECTOR_0	0	127	<input type="checkbox"/>	32768	301989888
1	Sector_Code_5	FLASH_SM_DBM1_SECTOR_0	127	16	<input type="checkbox"/>	8192	306151424
2	Sector_Code_6	FLASH1_LG_DBM1_SECTOR_0	143	127	<input type="checkbox"/>	32768	436207616
3	Sector_Code_7	FLASH1_SM_DBM1_SECTOR_0	270	16	<input type="checkbox"/>	8192	440369152

Figure 11 Configuration for all large sectors and all small sectors of dual first half (for 16M: Block#0 and Block#1)

While using the whole code flash sectors, if the sectors are not protected from erase, click the + button four times from the **[16M] CFIs Sector List Dual Second half** tab as shown in Figure 12. Figure 12 represents all (Block#0=127 + Block#1=127) large sectors and all (Block#0=16 + Block#1=16) small sectors that are configured.



Index	Name	CFIsSectorIdentifier	CFIsSectorNumber	CFIsNumberOfSectors	CFIsProtected	CFIsSectorSize	CFIsSectorStartaddress
0	Sector_Code_12	FLASH_LG_DBM1_SECTOR_0	0	127	<input type="checkbox"/>	32768	301989888
1	Sector_Code_13	FLASH_SM_DBM1_SECTOR_0	127	16	<input type="checkbox"/>	8192	306151424
2	Sector_Code_14	FLASH1_LG_DBM1_SECTOR_0	143	127	<input type="checkbox"/>	32768	436207616
3	Sector_Code_15	FLASH1_SM_DBM1_SECTOR_0	270	16	<input type="checkbox"/>	8192	440369152

Figure 12 Configuration for all large sectors and all small sectors of dual second half (for 16M: Block#0 and Block#1)

2.3 Adapting your application

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

```
#include "CFIs.h" /* Code Flash Driver */
```

This publishes the required function/data prototypes to the application.

In the case of single bank mode, link and load the Code Flash driver to execute from RAM because the Code Flash driver performs the job in non-blocking mode and thus must return while the flash operation is ongoing, and no code can be executed from a logical bank containing a sector to be written/erased in code flash during the flash operation.

In the case of Mapping A in dual bank mode, link and load the Code Flash driver to execute from the first half area, and flash sectors in the second half area are targeted for write/erase operation.

2 Using the Code Flash driver

In the case of Mapping B in dual bank mode, link and load the Code Flash driver to execute from the second half area, and flash sectors in the first half area are targeted for write/erase operation.

If you use 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):

- Code flash region
- Variables between `CFLS_START_SEC_VAR_NONCACHEABLE` and `CFLS_STOP_SEC_VAR_NONCACHEABLE`

For detailed information,

see [2.5.1 Memory allocation keyword](#) and [6.6 Function table](#).

Here are some points to note while using the Code Flash driver:

1. Reading cannot be done from the flash sector on the logical bank that includes a target flash sector for write/erase operation. Therefore, if the user application is executed from a sector on a logical bank whose sector is written/erased, it leads to hardware error. Do not run the application from a sector on a logical bank containing a sector to be written/erased. As a special case, if the `CFLS_EraseBlock`, `CFLS_Erase`, or `CFLS_Write` API is called by specifying `CFLS_SchMode_Blocking` with `SchMode` as the parameter, it can execute the application on CM0+ from a sector on the logical bank that has another sector to be written or erased by the API.
2. User application on CM0+ is blocked (suspended) during erasing sector or writing by the `CFLS_EraseBlock`, `CFLS_Erase`, or `CFLS_Write` API, if the `CFLS_SchMode_Blocking` API is called with the `SchMode` parameter of. Note the sector erasing takes considerably long time. For details on sector erase time, see the datasheet.
3. Before executing the Code Flash driver, select the single or dual bank mode with the `MAIN_BANK_MODE` bit in the `FLASHC_FLASH_CTL` or `FLASHC1_FLASH_CTL` register depending on the flash area to be accessed. When dual bank mode is used, the single bank mode is used. To use dual bank mode, select mapping A or mapping B with the `MAIN_MAP` bit in the `FLASHC_FLASH_CTL` or `FLASHC1_FLASH_CTL` register.

2 Using the Code Flash driver

2.4 Starting the build process

Do the following to build your application. For a clean build, use the build command with the target `clean_all` before! (make `clean_all`)

1. Type the following command into the command shell to generate the necessary configuration dependent files. See [3.3 Generated files](#) for details.

```
> make generate
```

2. Type the following command to resolve all required file dependencies.

```
> make depend
```

3. Compile and link the application with the following command.

```
> 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 Memory mapping

Macros regarding memory mapping are available in the `CFls_Cfg.h` in the `generate/include` directory. These macros must be defined, as necessary.

2.5.1 Memory allocation keyword

- `CFLS_START_SEC_CODE` / `CFLS_STOP_SEC_CODE`
 - Memory section type is CODE. All executable code is allocated in this section.
- `CFLS_START_SEC_CONST` / `CFLS_STOP_SEC_CONST`
 - Memory section type is CONST. The following contents are allocated in this section:
 - All configuration data
 - Hardware register base address data
- `CFLS_START_SEC_VAR` / `CFLS_STOP_SEC_VAR`
 - Memory section type is VAR. The following variable is allocated in this section:
 - Internal data
- `CFLS_START_SEC_VAR_NONCACHEABLE` / `CFLS_STOP_SEC_VAR_NONCACHEABLE`
 - Memory section type is VAR. The variables in this section must be allocated in non-cacheable area. The following variable is allocated in this section:
 - Data referred by IPC.

Note: When data cache is enabled in the MCU (for example, Arm® Cortex®-M7 CPU), this memory section must be in the non-cacheable region by setting of MPU. For further information, see [6.7 Memory protection unit \(MPU\)](#).

Note: If the Arm® CM7 processor is configured in the Resource module and the configuration parameter `CFlsEnableDataCache` is set to true, this memory section is named as `cfls_noncacheable_bss` in `CFls_Cfg.h`. By linker. The section of `.cfls_noncacheable_bss` must be allocated to an address in the non-cacheable region.

2 Using the Code Flash driver

Note: The only macro definitions in CFls_Cfg.h that the user can edit are those that use the _Pragma operator, commented as / editable */.*

The following are editable sections in CFls_Cfg.h:

```
#ifndef __ghs__

#define CFLS_START_SEC_VAR_NONCACHEABLE /* CFls non-cacheable var start */ \
    _Pragma("ghs section bss=\".cfls_noncacheable_bss\"") /* editable */
#define CFLS_STOP_SEC_VAR_NONCACHEABLE /* CFls non-cacheable var stop */ \
    _Pragma("ghs section bss=default")

#else

#define CFLS_START_SEC_VAR_NONCACHEABLE /* CFls non-cacheable var start */ \
    _Pragma("default_variable_attributes = @ \".cfls_noncacheable_bss\"") /* editable */
#define CFLS_STOP_SEC_VAR_NONCACHEABLE /* CFls non-cacheable var stop */ \
    _Pragma("default_variable_attributes = ")

#endif
```

3 Structure and dependencies

3 Structure and dependencies

The Code Flash driver consists of static, configuration, and generated files.

3.1 Static files

Static files of the Code Flash driver are located in the directory `$(TRESOS_BASE)/plugins/CFLs_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 Code Flash driver is done using the EB tresos Studio software. When saving a project, the configuration description is written in the `CFLs.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.

3.3 Generated files

During the build process, the following file is generated based on the current configuration. This file is in the `output/generated` subfolder of your project folder.

- `include/CFLs_Cfg.h` contains the configuration declarations and the list of the code flash sectors for the Code Flash driver.

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.

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* [\[1\]](#).

Note: Description file that is delivered with the Code Flash driver is in
`$(TRESOS_BASE)/plugins/CFls_TS_*/config/CFls.xdm`.

5 Functional description

5 Functional description

The Code Flash driver provides interface to write data to and erase the code flash memory.

5.1 Function of the Code Flash driver

5.1.1 Writing data to the flash memory

The Code Flash driver supports writing data to the flash memory with polling-controlled job. The polling-controlled job is simply used for writing data in units of 8 bytes, 32 bytes, or 512 bytes. For writing data, a higher value is automatically selected from the length of `CFls_Write` API to reduce writing time.

A write job is set up via the following command:

```
Status = CFls_Write(TargetAddress, SourceAddress, Length, SchMode, Verification);
```

Note: *The `TargetAddress` 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 8 bytes. The `SourceAddress` must be an address on SRAM. If the `Length` is not a multiple of 8 bytes, the remainder is filled with erased value. In addition, if the `Length` is not a multiple of 8, the area from the `SourceAddress` to the `Length` rounded up to a multiple of 8 must be located within SRAM.*

Note: *The address of the code flash area to be written must be contiguous from the `TargetAddress` to the `Length`.*

Note: *If an area of different flash (different block numbers) is accessed, a job can be started even if the jobs in other flash areas (different block numbers) are busy.*

When `SchMode` is `CFls_SchMode_Blocking`, if the write job is completed successfully, data is written from `SourceAddress` to the flash memory `TargetAddress` up to `Length` and the function returns `CFls_OperationStatus_Complete`. Otherwise, it returns `CFls_OperationStatus_Failed`.

When `SchMode` is `CFls_SchMode_NonBlocking`, if the write job is accepted, then the function returns `CFls_OperationStatus_InProgress` and the job will be executed on the next call(s) of `CFls_GetOperationStatus()` (If the flash area to be accessed is Block#1 (in 16M), `CFls_GetOperationStatus2nd()`). If the job is rejected, the function returns `CFls_OperationStatus_Failed`.

When `SchMode` is `CFls_SchMode_NonBlocking`, on each call of the `CFls_GetOperationStatus` function (If the flash area to be accessed is Block#1 (in 16M), function `CFls_GetOperationStatus2nd`), 512-byte data is written from `SourceAddress` to the flash memory `TargetAddress` at most.

If the job is still ongoing, the function `CFls_GetOperationStatus` (If the flash area to be accessed is Block#1 (in 16M), function `CFls_GetOperationStatus2nd`) returns `CFls_OperationStatus_InProgress`. After the total number of bytes is successfully written to the flash memory, the function returns `CFls_OperationStatus_Complete` once.

If any hardware error occurred during the write process, the function `CFls_GetOperationStatus` (If the flash area to be accessed is Block#1 (in 16M), function `CFls_GetOperationStatus2nd`) returns `CFls_OperationStatus_Failed` and aborts the write job.

When `Verification` is `TRUE`, the written data will be verified. On each call of the `CFls_GetOperationStatus` function (If the flash area to be accessed is Block#1 (in 16M),

5 Functional description

CFIs_GetOperationStatus2nd function), data up to 1024 bytes at most is verified. If the verification fails, the function will return CFIs_OperationStatus_Failed.

5.1.2 Erasing data from the flash memory

The Code Flash driver supports erasing (parts of) the flash memory with polling-controlled job.

An erase job is set up via the following command; the target flash sector is specified by a sector number configured by the configuration parameter CFIsNumberOfSectors:

```
Status = CFIs_EraseBlock(SectorNumber, SchMode, Verification);
```

Note: *The SectorNumber must be valid number within the configured sector list. Only a target flash sector is erased.*

An erase job is set up via the following command and the target is specified by the top and end addresses:

```
Status = CFIs_Erase(Address, EndAddress, SchMode, Verification);
```

Note: *The Address must be aligned to a flash sector. The EndAddress does not need to be an aligned address. The last sector for erase is regarded as a sector where the EndAddress exists. For example, the top sector starts from 0x10000000 and the size is 32KB (0x8000 bytes). The only top sector is regarded as target for erase when the EndAddress is between 0x10000000 and 0x10007FFF. If EndAddress is 0x10008000, the next sector also is erased. A flash sector is the smallest amount of flash memory that can be erased in one pass.*

Note: *The address of the code flash area to be erased must be consecutive from Address to EndAddress.*

Note: *If an area of different flash (different block numbers) is accessed, a job can be started even if the jobs in other flash areas (different block numbers) are busy.*

When SchMode is CFIs_SchMode_Blocking, if the erase job is completed successfully, then the target sector(s) is erased entirely and the function returns CFIs_OperationStatus_Complete. Otherwise, it returns CFIs_OperationStatus_Failed.

When SchMode is CFIs_SchMode_NonBlocking, if the erase job is accepted, then the function returns CFIs_OperationStatus_InProgress and the job will be executed on the next call(s) of CFIs_GetOperationStatus() (If the flash area to be accessed is Block#1 (in 16M), CFIs_GetOperationStatus2nd()). If the job was rejected, the function returns CFIs_OperationStatus_Failed.

When SchMode is CFIs_SchMode_NonBlocking, on each call of the CFIs_GetOperationStatus function (If the flash area to be accessed is Block#1 (in 16M), function CFIs_GetOperationStatus2nd), polling is performed. If a sector has been erased and next sector is targeted for erase, erase for the next sector will start.

If the job is still ongoing, the function CFIs_GetOperationStatus (If the flash area to be accessed is Block#1 (in 16M), function CFIs_GetOperationStatus2nd) returns CFIs_OperationStatus_InProgress. After the affected sectors are successfully erased, the function returns CFIs_OperationStatus_Complete once.

If any hardware error occurred during the erase process, the function CFIs_GetOperationStatus (If the flash area to be accessed is Block#1 (in 16M), function CFIs_GetOperationStatus2nd) returns CFIs_OperationStatus_Failed and aborts the erase job.

5 Functional description

When `Verification` is `TRUE`, the erased sector will be verified (i.e. blank-check). On each call of the `CFIs_GetOperationStatus` function (If the flash area to be accessed is `Block#1` (in 16M), `CFIs_GetOperationStatus2nd` function), area up to 1024 bytes at most is verified. If the verification fails, the function will return `CFIs_OperationStatus_Failed`.

5.1.3 Retrieving the status information

API functions are available to get the current operation state of the Code Flash driver:

```
Status = CFIs_GetOperationStatus();
```

In the case of flash area to be accessed is `Block#1` (in 16M)

```
Status = CFIs_GetOperationStatus2nd();
```

If a job is accepted by any API with `SchMode` of `CFIs_SchMode_NonBlocking` and the job is ongoing, the function `CFIs_GetOperationStatus` (If the flash area to be accessed is `Block#1` (in 16M), function `CFIs_GetOperationStatus2nd`) performs the necessary operation to continue.

5.1.4 Canceling a job prior to maturity

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

```
CFIs_Cancel();
```

In the case of flash area to be accessed is `Block#1` (in 16M)

```
CFIs_Cancel2nd();
```

Note: Do not call this function during the precedent execution of any API.

The function cancels the ongoing job but does not attempt to stop an ongoing hardware operation. When the function `CFIs_GetOperationStatus` (If the flash area to be accessed is `Block#1` (in 16M), function `CFIs_GetOperationStatus2nd`) is called right after the return of the function `CFIs_Cancel` (If the flash area to be accessed is `Block#1` (in 16M), function `CFIs_Cancel2nd`), if hardware stops, it returns `CFIs_OperationStatus_Canceled`. Otherwise, it returns `CFIs_OperationStatus_InProgress`.

5.1.5 Timeout supervision

The Code Flash driver provides a timeout monitoring for the job.

5.2 Re-entrancy

All API functions of the Code Flash driver are non-reentrant to the same code flash area.

6 Hardware resources

6 Hardware resources

6.1 Registers

The Code Flash driver deals with the registers listed in Appendix B – Access register table.

Note: Set the following register before using the Code Flash driver:

- FLASH_CTL (WS[bit3:0]): FLASH macro main interface wait states
- FLASH_CTL (MAIN_MAP[bit8]): Specifies remapping of flash macro main region
- FLASH_CTL (MAIN_BANK_MODE[bit12]): Specifies bank mode of flash macro main array

6.2 Interrupts

The Code Flash driver does not use any interrupt.

6.3 Fault

The Code 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 Code Flash driver cannot directly do the processing (such as clearing the validity bit field) for the fault report structures.

You should implement the fault interrupt handler and the handler should call a fault handling function (CFIs_Fault_Handling() and CFIs_Fault_Handling1()) provided by the Code Flash driver.

CFIs_Fault_Handling1() gets the fault information of code flash area Block#1 (in 16M).

The following is an example of the fault interrupt handler:

```
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))
        {
            CFIs_Fault_Handling(); /* Fault handling for Code Flash Driver */
        }
        /* The error-caused address is calculated by appending 0x18000000 */
        /* to [bit26:0] in DATA0 register of fault report structure. */
        if(WITHIN_AREA_FOR_THIS_CORE_16M(FAULT_STRUCT->DATA0))
        {
            CFIs_Fault_Handling1(); /* Fault handling for Code Flash Driver (for Block#1) */
        }
        Xxx(...); /* Fault handling for other than Code Flash Driver */
        ...
    }
}
```

6 Hardware resources

```

}
FAULT_STRUCT->INTR.bitField.FAULT = 1U;
FAULT_STRUCT->STATUS = 0x00000000UL;
}

```

6.4 IPC

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

A dedicated IPC structure (mailbox) for system calls is associated with each CPU core and the Code 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 Code Flash driver reports error. Similarly, for the IPC interrupt structure, dedicated structure for system calls is associated with each CPU core and the Code Flash driver uses it for CM0+, CM4/CM7_0, or 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)

6.5 System call

The system call is used for flash memory operations such as flash write operation and flash erase operation. 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 a dedicated 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.

When the `CFIs_EraseBlock`, `CFIs_Erase` or `CFIs_Write` API is called by the `SchMode` parameter of `CFIs_SchMode_Blocking`, the system calls for the flash operation are invoked with blocking mode. Whereas, if the API is called by the parameter of `CFIs_SchMode_NonBlocking`, the system call is invoked with non-blocking mode.

[Table 2](#) shows a summary of the system calls that the Code Flash driver uses.

6 Hardware resources

Table 2 System calls

Name	Opcode	Description
ProgramRow	0x06	Programs the addressed FLASH page
EraseSector	0x14	Erases the addressed FLASH sector

6.6 Function table

There are derivatives where the code flash area is divided into Block#0 and Block#1 (in 16M). In that case, IPC and System call are used to perform flash memory operations on Block#0, whereas the Function table is used to perform flash memory operations on Block#1, such as flash write and flash erase operations.

Table 3 shows a summary of the Function table that the Code Flash driver uses.

Table 3 Function table

Name	Address	Description
FmECT_EraseSector_Ext	0x0000C004	Erases the addressed FLASH sector
FmECT_ProgramWord_Ext	0x0000C008	Programs the addressed FLASH page

6.7 Memory protection unit (MPU)

As mentioned in section 6.4 IPC, the Code 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 such a store instruction with 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 Code 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.

- Code flash region
- Variables between CFLS_START_SEC_VAR_NONCACHEABLE and CFLS_STOP_SEC_VAR_NONCACHEABLE (for the areas for IPC)

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_16MB     (0x17UL << 1U) // Region size 16MB

#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
```

6 Hardware resources

```

#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[] =
{
    /* CFI's Non-cacheable region */    {0x28030000, (MPU_RASR_SIZE_64KB |
        MPU_NORMAL_NON_CACHEABLE | MPU_RASR_AP_FULL_ACCESS | MPU_RASR_ENABLE)},
    /* Code Flash region */            {0x10000000, (MPU_RASR_SIZE_16MB |
        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
    }
}

```

6 Hardware resources

```
/* 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
}
```

7 Appendix A – API reference

7.1 Data types

7.1.1 Code Flash driver data types

7.1.1.1 CFls_OperationStatusType

Type

```
typedef enum cfls_operationstatustype
{
    CFls_OperationStatus_Idle = 0,
    CFls_OperationStatus_InProgress = 1,
    CFls_OperationStatus_Complete = 2,
    CFls_OperationStatus_Canceled = 5,
    CFls_OperationStatus_Failed = 6
} CFls_OperationStatusType;
```

Description

This is the type of operation status.

7.1.1.2 CFls_SchModeType

Type

```
typedef enum cfls_schmodetype
{
    CFls_SchMode_Blocking = 0,
    CFls_SchMode_NonBlocking = 1
} CFls_SchModeType;
```

Description

This is the type of operation mode.

7.1.1.3 CFls_ConfigType

Description

This is the type of configuration structure for the Code Flash driver.

7.1.1.4 CFls_SectorListType

Description

This is the type of structure for list of sectors to be used by the Code Flash driver.

7 Appendix A – API reference

7.2 Macros

7.2.1 Configuration parameters

The Code Flash driver uses the following configuration parameters.

Table 4 Configuration parameters

Name	Description
CODEFLASH_TIMEOUT_FOR_WAIT	Specifies time-out for wait, in ticks of free running timer.
CODEFLASH_WDGCLEAR_PERIOD	Specifies frequency CFLs calls <code>CFLs_WdgClear(void)</code> , in ticks of free running timer.
CODEFLASH_ENABLE_DATA_CACHE	If the data cache on MCU is enabled, this macro must be defined.
CODEFLASH_SECTOR_LIST	Specifies list of code flash sectors for use. Only single bank mode is supported.
CODEFLASH_SECTOR_LIST_1ST	Specifies list of code flash sectors for use. Only first half in dual bank mode is supported.
CODEFLASH_SECTOR_LIST_2ND	Specifies list of code flash sectors for use. Only second half in dual bank mode is supported.

7.2.2 Resource information

The Code Flash driver uses the following resource information.

Table 5 Resource information

Name	Value	Description
CFLS_CODEFLASH_BASEADDRESS	From RESOURCE module	Code Flash base address
CFLS_CPUSS_ADDR	From RESOURCE module	CPUSS registers access address
CFLS_FAULT_ADDR	From RESOURCE module	Fault structure registers access address
CFLS_IPC_STRUCT_ADDR	From RESOURCE module	IPC structure registers base address
CFLS_FLASHC_ADDR	From RESOURCE module	Flash registers access address
CFLS_FLASHC1_ADDR	From RESOURCE module	Flash registers access address (for 16M: Block#1)

7.2.3 Configuration information

The Code Flash driver uses the following configuration information.

Table 6 Configuration information

Name	Value	Description
CFLS_TOTAL_SIZE	Tresos generation	Total size of configured code flash sectors (for Block#0 in single bank mode)

7 Appendix A – API reference

Name	Value	Description
CFLS_TOTAL_SIZE1	Tresos generation	Total size of configured code flash sectors (for Block#1(16M) in single bank mode)
CFLS_TOTAL_SIZE_DUAL_1ST	Tresos generation	Total size of configured code flash sectors (for first half of Block#0 (16M) in dual bank mode or other than 16M)
CFLS_TOTAL_SIZE1_DUAL_1ST	Tresos generation	Total size of configured code flash sectors (for first half of Block#1 (16M) in dual bank mode)
CFLS_TOTAL_SIZE_DUAL_2ND	Tresos generation	Total size of configured code flash sectors (for second half of Block#0 (16M) in dual bank mode or other than 16M)
CFLS_TOTAL_SIZE1_DUAL_2ND	Tresos generation	Total size of configured code flash sectors (for second half of Block#1 (16M) in dual bank mode)
CFLS_NUMBER_OF_SECTOR_LISTS	Tresos generation	Number of elements of array in <code>CFls_SectorList</code>
CFLS_NUMBER_OF_SECTORS	Tresos generation	Number of configured code flash sectors

7.3 Functions

7.3.1 CFls_EraseBlock

Syntax

```
CFls_OperationStatusType CFls_EraseBlock
(
    uint32          SectorNumber,
    CFls_SchModeType SchMode,
    boolean         Verification
)
```

Reentrancy

Non-reentrant to the same code flash area.

Parameters (in)

- `SectorNumber` - Number of configured sector list.
- `SchMode` - Blocking or non-blocking mode.
- `Verification` - If TRUE, enables verification of Erase.

Parameters (out)

None

7 Appendix A – API reference

Return value

- `CFls_OperationStatus_InProgress` - Non-Blocking operation is active.
- `CFls_OperationStatus_Complete` - Operation completed successfully.
- `CFls_OperationStatus_Failed` - Operation not attempted due to hardware failure.

Description

Erase operation for a sector specified by sector number.

Caveats

- The `SectorNumber` parameter must be within the range of configured `CFlsSectorNumber`.
- The protected sector must not be a sector specified by the `SectorNumber` parameter.
- Only one erase or write job can be accepted at the same time. So, any job in non-blocking mode must not be ongoing.

7.3.2 CFls_Erase

Syntax

```
CFls_OperationStatusType CFls_Erase
(
    uint32 *           Address,
    uint32 *           EndAddress,
    CFls_SchModeType   SchMode,
    boolean            Verification
)
```

Reentrancy

Non-reentrant to the same code flash area.

Parameters (in)

- `Address` - Start Address for Erase.
- `EndAddress` - End Address for Erase.
- `SchMode` - Blocking or non-blocking mode.
- `Verification` - If TRUE, enables verification of Erase.

Parameters (out)

None

Return value

- `CFls_OperationStatus_InProgress` - Non-blocking operation is active.
- `CFls_OperationStatus_Complete` - Operation completed successfully.
- `CFls_OperationStatus_Failed` - Operation not attempted due to hardware failure.

Description

Erase operation for sector(s) specified by the start and end address.

7 Appendix A – API reference

Caveats

- The `Address` parameter must be the top address of a sector.
- The `EndAddress` parameter must be within the address range of configured sectors in code flash.
- The protected sector must not be included between the `Address` parameter and the `EndAddress` parameter.
- Only one erase or write job can be accepted at the same time. So, any job in non-blocking mode must not be ongoing now.

7.3.3 CFls_Write

Syntax

```
CFls_OperationStatusType CFls_Write  
(  
    uint32 *          TargetAddress,  
    uint32 *          SourceAddress,  
    uint32            Length,  
    CFls_SchModeType  SchMode,  
    boolean            Verification  
)
```

Reentrancy

Non-reentrant to the same code flash area.

Parameters (in)

- `TargetAddress` - Destination address for Write.
- `SourceAddress` - Source address (of written data).
- `Length` - Number of bytes to write.
- `SchMode` - Blocking or non-blocking mode.
- `Verification` - If TRUE, enable verification of written data.

Parameters (out)

None

Return value

- `CFls_OperationStatus_InProgress` - Non-blocking operation is active.
- `CFls_OperationStatus_Complete` - Operation completed successfully.
- `CFls_OperationStatus_Failed` - Operation not attempted due to hardware failure.

Description

Write operation.

Caveats

- The `TargetAddress` parameter must be within the address range of configured sectors in code flash and it must be a multiple of 8.
- The `SourceAddress` parameter must be an address on SRAM.

7 Appendix A – API reference

- The `SourceAddress` parameter must not be a NULL pointer.
- The `Length` parameter must not be 0.
- The `TargetAddress` parameter plus the `Length` parameter minus 1 must be within the address range of configured sectors in code flash.
- If the `Length` is not a multiple of 8, the remainder is filled with erased value.
- If the `Length` is not a multiple of 8, the area from the `SourceAddress` to length rounded up the `Length` to a multiple of 8 must be located within SRAM.
- Only one erase or write job can be accepted at the same time. So, any job in non-blocking mode must not be ongoing now.

7.3.4 CFls_GetOperationStatus

Syntax

```
CFls_OperationStatusType CFls_GetOperationStatus  
(  
    void  
)
```

Reentrancy

Non-reentrant to the same code flash area.

Parameters (in)

None

Parameters (out)

None

Return value

- `CFls_OperationStatus_Idle` - No operation in progress and no operation is finished.
- `CFls_OperationStatus_InProgress` - Non-blocking operation is active.
- `CFls_OperationStatus_Complete` - Operation completed successfully.
- `CFls_OperationStatus_Canceled` - Operation is canceled.
- `CFls_OperationStatus_Failed` - Operation not attempted due to hardware failure.

Description

This function performs the processing of the flash erase or write job on a single flash area (other than 16M) or a flash area of Block#0 (16M). When a job has been initiated, the integration environment calls the function `CFls_GetOperationStatus` until the job is finished. This function performs each job operation by Code Flash driver's internal state.

Caveats

- If a job operation in non-blocking mode was ongoing before the call of the function `CFls_GetOperationStatus` and the job operation continues after the return, the function returns `CFls_OperationStatus_InProgress`.

7 Appendix A – API reference

- If a job operation in non-blocking mode was ongoing before the call of the function `CFls_GetOperationStatus` and entire operation is finished successfully, the function returns `CFls_OperationStatus_Complete`.
- If a job operation in non-blocking mode was ongoing before the call of the function `CFls_GetOperationStatus` and an error was detected, the function returns `CFls_OperationStatus_Failed`.
- If the execution time for a job operation in non-blocking mode exceeds a value of the configuration `CFls_CodeflashTimeoutForWait`, the function `CFls_GetOperationStatus` returns `CFls_OperationStatus_Failed`.
- If a job operation in non-blocking mode was ongoing before the call of the function `CFls_GetOperationStatus`, the function `CFls_Cancel` was called before, and the hardware operation is finished, the function `CFls_GetOperationStatus` returns `CFls_OperationStatus_Canceled`.
- Only one erase or write job can be accepted at the same time.

7.3.5 CFls_GetOperationStatus2nd

Syntax

```
CFls_OperationStatusType CFls_GetOperationStatus2nd  
(  
    void  
)
```

Reentrancy

Non-reentrant to the same code flash area.

Parameters (in)

None

Parameters (out)

None

Return value

- `CFls_OperationStatus_Idle` - No operation in progress and no operation is finished.
- `CFls_OperationStatus_InProgress` - Non-blocking operation is active.
- `CFls_OperationStatus_Complete` - Operation completed successfully.
- `CFls_OperationStatus_Canceled` - Operation is canceled.
- `CFls_OperationStatus_Failed` - Operation not attempted due to hardware failure.

Description

This function performs the processing of the flash erase or write job on a flash area of Block#1(16M). When a job has been initiated, the integration environment calls the function `CFls_GetOperationStatus2nd` until the job is finished. This function performs each job operation by Code Flash driver's internal state.

7 Appendix A – API reference

Caveats

- If a job operation in non-blocking mode was ongoing before the call of the function `CFIs_GetOperationStatus2nd` and the job operation continues after the return, the function returns `CFIs_OperationStatus_InProgress`.
- If a job operation in non-blocking mode was ongoing before the call of the function `CFIs_GetOperationStatus2nd` and entire operation is finished successfully, the function returns `CFIs_OperationStatus_Complete`.
- If a job operation in non-blocking mode was ongoing before the call of the function `CFIs_GetOperationStatus2nd` and an error was detected, the function returns `CFIs_OperationStatus_Failed`.
- If the execution time for a job operation in non-blocking mode exceeds a value of the configuration `CFIsCodeflashTimeoutForWait`, the function `CFIs_GetOperationStatus2nd` returns `CFIs_OperationStatus_Failed`.
- If a job operation in non-blocking mode was ongoing before the call of the function `CFIs_GetOperationStatus2nd`, the function `CFIs_Cancel2nd` was called before, and hardware operation is finished, the function `CFIs_GetOperationStatus2nd` returns `CFIs_OperationStatus_Canceled`.
- Only one erase or write job can be accepted at the same time.

7.3.6 CFIs_Cancel

Syntax

```
void CFIs_Cancel  
(  
    void  
)
```

Reentrancy

Non-reentrant to the same code flash area.

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

Cancels an ongoing flash job immediately on a single flash area (other than 16M) or a flash area of Block#0 (16M) (write/erase).

Caveats

- Do not call the function `CFIs_Cancel` during the precedent execution of any API.
- The function `CFIs_Cancel` does not attempt to stop an ongoing hardware operation.

7 Appendix A – API reference

7.3.7 CFls_Cancel2nd

Syntax

```
void CFls_Cancel  
(  
    void  
)
```

Reentrancy

Non-reentrant to the same code flash area.

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

Cancels an ongoing flash job immediately on a flash area of Block#1 (16M) (write/erase).

Caveats

- Do not call the function `CFls_Cancel2nd` during the precedent execution of any API.

The function `CFls_Cancel2nd` does not attempt to stop an ongoing hardware operation.

7.4 Required callback functions

7.4.1 Callout functions

There are some mandatory callback functions that is expected by the Code Flash driver.

7.4.1.1 CFls_WdgClear

Syntax

```
void CFls_WdgClear  
(  
    void  
)
```

Reentrancy

Don't care

Parameters (in)

None

7 Appendix A – API reference

Return value

None

Description

This function is implemented for clearing (triggering) the watchdog timer by user.

7.4.1.2 CFls_GetCounterValue

Syntax

```
uint32 CFls_GetCounterValue  
(  
    void  
)
```

Reentrancy

Don't care

Parameters (in)

None

Return value

A timer count value of a free running timer.

Description

The function is implemented for returning a timer count value of a free running timer by user.

7.4.1.3 CFls_GetElapsedValue

Syntax

```
uint32 CFls_GetElapsedValue  
(  
    uint32 PreviousTime  
)
```

Reentrancy

Don't care

Parameters (in)

- PreviousTime - Previous time.

Return value

A timer count difference since the point in time is provided.

Description

The function is implemented for returning a timer count difference since the point in time that is provided as parameter by user.

8 Appendix B – Access register table

8.1 FLASHC

Table 7 FLASHC access register table

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
FLASH_CTL	31:0	Word (32 bits)	0x00550000 WS[3:0] MAIN_MAP[8:8] MAIN_BANK_MODE[12:12]	Control	CFls_Init CFls_Write CFls_Erase CFls_Erase_Block	0x0077330F	0x00550000 WS[3:0] MAIN_MAP[8:8] MAIN_BANK_MODE[12:12] (After CFls_Init)
FLASH_CMD	31:0	Word (32 bits)	0x00000001	Command	CFls_GetOperationStatus CFls_Cancel	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM0_STATUS	31:0	Word (32 bits)	0x00000001	CM0+ interface status	CFls_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_STATUS	31:0	Word (32 bits)	0x00000001	CM4 interface status	CFls_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_0_STATUSES	31:0	Word (32 bits)	0x00000001	CM7 #0 interface status	CFls_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_1_STATUSES	31:0	Word (32 bits)	0x00000001	CM7 #1 interface status	CFls_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CM7_2_STATU S	31:0	Word (32 bits)	0x00000001	CM7 #2 interface status	CFls_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_3_STATU S	31:0	Word (32 bits)	0x00000001	CM7 #3 interface status	CFls_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.2 FM_CTL_ECT(FLASHC)

Table 8 FM_CTL_ECT access register table

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
MAIN_FLASH_SAFETY	31:0	Word (32 bits)	0x00000001 (Before start of writing, before start of erasing) 0x00000000 (After CFls_Init, after finish of writing, after finish of erasing)	Main (code) flash security enable	CFls_Init CFls_GetOperationStatus CFls_Cancel	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.)

8.3 FLASHC1

Table 9 FLASHC1 access register table

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
FLASH_CTL	31:0	Word (32 bits)	0x00550000 WS[3:0] MAIN_MAP[8:8] MAIN_BANK_MODE[12:12]	Control	CFls_Init CFls_Write CFls_Erase CFls_Erase_Block	0x0077330F	0x00550000 WS[3:0] MAIN_MAP[8:8] MAIN_BANK_MODE[12:12] (After CFls_Init1)
FLASH_CMD	31:0	Word (32 bits)	0x00000001	Command	CFls_GetOperationStatus2nd CFls_Cancel2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM0_STATUS	31:0	Word (32 bits)	0x00000001	CM0+ interface status	CFls_GetOperationStatus2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM4_STATUS	31:0	Word (32 bits)	0x00000001	CM4 interface status	CFls_GetOperationStatus2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_0_STATUS	31:0	Word (32 bits)	0x00000001	CM7 #0 interface status	CFls_GetOperationStatus2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_1_STATUS	31:0	Word (32 bits)	0x00000001	CM7 #1 interface status	CFls_GetOperationStatus2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
CM7_2_STATUS	31:0	Word (32 bits)	0x00000001	CM7 #2 interface status	CFls_GetOperationStatus2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CM7_3_STATUS	31:0	Word (32 bits)	0x00000001	CM7 #3 interface status	CFIs_GetOperationStatus2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.4 FM_CTL_ECT(FLASHC1)

Table 10 FM_CTL_ECT access register table

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
MAIN_FLASH_SAFETY	31:0	Word (32 bits)	0x00000001 (Before start of writing, before start of erasing)	Main (code) flash security enable	CFIs_Init1 CFIs_GetOperationStatus2nd CFIs_Cancel2nd	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
			0x00000000 (After CFIs_Init, after finish of writing, after finish of erasing)				
STATUS	31:0	Word (32 bits)	-	Status read from flash macro	Read only	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.5 FAULT

Table 11 FAULT access register table

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
STATUS	31:0	Word (32 bits)	-	Fault status	Read only	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
MASK1	31:0	Word (32 bits)	0x00070000	Fault mask 1	CFIs_Init	0x00070000	0x00070000 (After CFIs_Init)
MASK2	31:0	Word (32 bits)	0x00007000	Fault mask 2	CFIs_Init1	0x00007000	0x00007000 (After CFIs_Init1)

8.6 IPC

Table 12 IPC access register table

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_NOTIFY	31:0	Word (32 bits)	Notification event (When call of system call)	IPC notification register	CFIs_GetOperationStatus	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)	IPC Data Register 0	CFIs_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
IPC_INTR	31:0	Word (32 bits)	IPC release event clear	IPC interrupt status register	CFIs_GetOperationStatus CFIs_Cancel	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
IPC_INTR_MASK	31:0	Word (32 bits)	0x00000000	IPC interrupt mask register	CFIs_Init CFIs_GetOperationStatus	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)

8.7 CPUSS

Table 13 CPUSS access register table

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.)

Revision history

Revision history

Document revision	Date	Description of changes
**	2019-07-02	Initial release.
*A	2019-10-24	Updated using the Code Flash driver: Updated configuring the Code Flash driver: Updated configuration details: Added notes. Updated adapting your application: Added usage notes. Updated hardware resources: Updated system call (NMI exception): Added and modified description.
*B	2020-11-16	Updated to Infineon template.
*C	2021-03-26	Updated 5.1.1 Writing data to the flash memory Updated 2.2.1.3.2 CFIsSectorNumber (fixed typo)
*D	2021-12-23	Updated to Infineon style.
*E	2023-03-03	Added CM7_2, CM7_3 and 16M to "Abbreviations and definitions" Added description for 16M and Dual support in "2.2 Configuring the Code Flash driver". Added description of Dual bank mode operation in "2.3 Adapting your application". Added an access range Note in "5.1.1 Writing Data to Flash Memory" and description of the API for 16M: CFIs_GetOperationStatus2nd(). Added an access range Note in "5.1.2 Erasing data from the flash memory" and description of the API for 16M: CFIs_GetOperationStatus2nd(). Added description of API for 16M: CFIs_GetOperationStatus2nd() in "5.1.3 Retrieving the status information". Added description of API for 16M: CFIs_Cancel2nd() and CFIs_GetOperationStatus2nd() in "5.1.4 Canceling a job prior to maturity". Changed the description of "5.2 Re-entrancy". Added register settings for Dual Bank in "6.1 Registers". Added description of function for 16M: CFIs_Fault_Handling1() in "6.3 Fault". Added CM7_2 and CM7_3 to "6.4 IPC" Added CM7_2 and CM7_3 to "6.5 System call" Added description of function table for Block#1 of 16M to "6.6 Function table". Added CM7_2 and CM7_3 to "6.7 Memory protection unit (MPU)" Added parameters for Dual bank to "7.2.1 Configuration parameters". Added resource information for 16M to "7.2.2 Resource information".

Revision history

Document revision	Date	Description of changes
		<p>Added configuration information for 16M and configuration information for Dual to "7.2.3 Configuration information".</p> <p>Changed the description of Reentrancy in "7.3.1 CFIs_EraseBlock".</p> <p>Changed the description of Reentrancy in "7.3.2 CFIs_Erase".</p> <p>Changed the description of Reentrancy in "7.3.3 CFIs_Write".</p> <p>Changed the description of Reentrancy in "7.3.4 CFIs_GetOperationStatus " and added the description of 16M.</p> <p>Added function CFIs_GetOperationStatus2nd for 16M to "7.3.5 CFIs_GetOperationStatus2nd ".</p> <p>Changed the description of Reentrancy in "7.3.1 CFIs_EraseBlock".</p> <p>Changed the description of Reentrancy in "7.3.6 CFIs_Cancel" and added the description of 16M.</p> <p>Added function CFIs_GetOperationStatus2nd for 16M to " 7.3.7 CFIs_Cancel2nd".</p> <p>Changed the FLASH_CTL operation in "7.3.7 CFIs_Cancel2nd".</p> <p>To distinguish between FLASHC and FLASHC1 FM_CTL_ECT "8.2 FM_CTL_ECT" was changed to "8.2 FM_CTL_ECT(FLASHC)".</p> <p>Added registers for 16M to "8.3 FLASHC1" and "8.4 FM_CTL_ECT(FLASHC1)".</p> <p>Added MASK2 register to "8.5 FAULT"</p>
*F	2023-12-08	Web release. No content updates.
*G	2024-03-18	Added note to "2.5.1 Memory allocation keyword"

Trademarks

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

Edition 2024-03-18

Published by

Infineon Technologies AG

81726 Munich, Germany

**© 2024 Infineon Technologies AG.
All Rights Reserved.**

**Do you have a question about this
document?**

Email:

erratum@infineon.com

Document reference

002-27310 Rev. *G

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.