User Manual

16

Flash Self-programming Library

Type T01, European Release

16 Bit Single-chip Microcontroller
RL78 Series

Installer: RENESAS_RL78_FSL_T01_xVxx

Renesas Electronics
www.renesas.com

R01US0016ED0120, Rev. 1.20
Dec 27, 2023

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Table of Contents

Chapter 1 Introduction

The RL78 products are equipped with an internal firmware, which allows rewriting of the flash memory without the use of an external programmer. In addition to this Renesas provides the so called Self-programming library. This library offers an easy-to-use interface to the internal firmware functionality. By calling the Self-programming library functions from user program, the contents of the flash memory can easily be rewritten in the field.

The Flash Self-programming library provides APIs for the C and assembly language of the CA78K0R, IAR V1.xx, IAR V2.xx, GNU, CC-RL and LLVM development environments. (APIs for the assembly language are provided by the CA78K0R and CC-RL development environments only.)

The Flash Self-programming library for IAR V2.xx development environments (except linker sample file) can also be used with the IAR V3.xx or later version development environments.

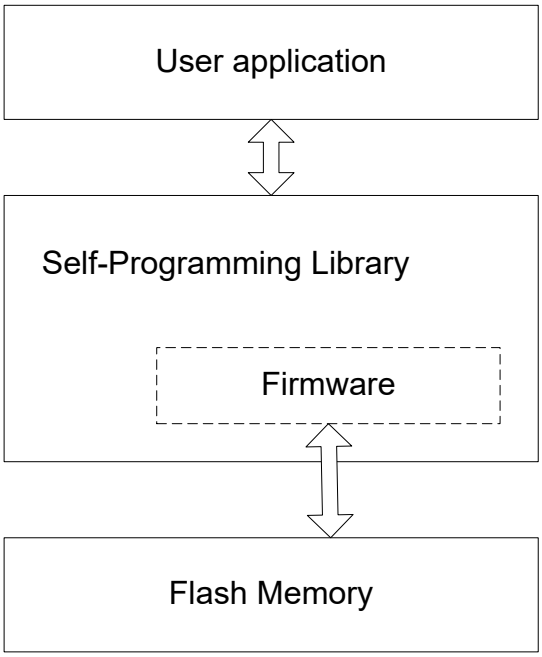

Figure 1-1 Flash Access

CAUTION

- The Self-programming library rewrites the contents of the flash memory by using the CPU, its registers and the internal RAM. Thus the user program cannot be executed while the Self-programming library is in process.
- Use of some RAM areas are prohibited when using the Self-programming. For detailed information please refer to the device Users Manual.

1.1 Naming convention

Certain terms, required for the description of the Flash Self-programming are long and too complicated for good readability of the document. Therefore, special names and abbreviations will be used in the course of this document to improve the readability.

Table 1-1 Used abbreviations and acronyms

Abbreviations / Acronyms	Description
Block	Smallest erasable unit
Code Flash	Embedded Flash where the application code is stored. For devices without Data Flash EEPROM emulation might be implemented on that flash in the so called data area.
Data Flash	Embedded Flash where mainly the data of the EEPROM emulation are stored. Beside that also code operation might be possible.
FSL	Flash Self-programming Library
Flash	“Flash EPROM” - Electrically erasable and programmable nonvolatile memory. The difference to ROM is, that this type of memory can be re-programmed several times.
RAM	“Random access memory” - volatile memory with random access
ROM	“Read only memory” - nonvolatile memory. The content of that memory cannot be changed.
SCI	Status check internal mode
SCU	Status check user mode

Chapter 2 Programming Environment

This chapter describes the software environment which is used to rewrite flash memory by using the Self-programming library. For the hardware environment please refer to the device user manual.

2.1 Software Environment

The Self-programming library allocates its code inside the user area and consumes up to about 1300 bytes of the program memory. The self-programming library itself uses work area in form of entry RAM, application stack and so called data buffer for data exchange with the firmware. The following table lists the required software resources. All values are specified for version V2.20 for IAR V1.xx, CA78K0R and GNU compiler and version V2.21 for IAR V2.xx, CC-RL and LLVM compiler of the Self-programming Library.

Table 2-1 Resource consumption

Item	Description
User RAM	Some RAM areas are prohibited. Please refer to the device users manual for detailed information.
Stack	max. 60 bytes
Data buffer	5 - 256 bytes
Self-programming library	max. 1350 bytes **Note** Code size of the Self-programming library varies depending on their configuration.

CAUTION

- The Self-programming operation is not ensured if the user manipulates the above resources. Do not manipulate these resources during a Self-programming session.Use of some RAM areas are prohibited when using the Self-programming. For detailed information please refer to the device Users Manual.
- The user must release the above resources before calling the Self-programming library.

Table 2-2 Code size of the library depends on the user configuration

	Code size
Max. code size	1350 bytes
Max. code size (without GetInfo, SetInfo, FSL_ForceReset, FSL_PrepareExtFunctions, FSL_StandBy, FSL_WakeUp, FSL_CopySection and FSL_SwapBootCluster)	622 bytes
Max. code size (without GetInfo, SetInfo and FSL_SwapBootCluster) --> FSL_InvertBootFlag, FSL_ForceReset and FSL_GetBootFlag included	905 bytes

2.1.1 Data buffer

The data buffer is used for data-exchange between the firmware and the Self-programming library.

2.1.2 Location of segments

The following tables show the segment location for both execution modes.

Table 2-3 Segments in status check internal mode (SCI)

Segment name	Segment location	Description
FSL_FCD	ROM	Segment will be used for the following functions: FSL_Init, FSL_Close, FSL_Open, FSL_PrepareFunctions, FSL_PrepareExtFunctions, FSL_ChangeInterruptTable, FSL_RestoreInterruptTable, FSL_GetVersionString
FSL_FECD	ROM	Segment will be used for the following functions: FSL_GetBlockEndAddr, FSL_GetFlashShieldWindow, FSL_GetSwapState, FSL_GetBootFlag and FSL_GetSecurityFlags
FSL_BCD	ROM	Must only be placed via linker file in case FSL_Erase, FSL_Write, FSL_IVerify or FSL_BlankCheck functions are used
FSL_BECD	ROM	Must only be placed via linker file in case FSL_SetXXX and FSL_InvertBootFlag functions are used
FSL_RCD	ROM / RAM (*1)	Segment contains the following functions: FSL_ForceReset, FSL_SetXXXXFlag, FSL_StatusCheck, FSL_BlankCheck, FSL_Erase, FSL_IVerify, FSL_Write, FSL_SwapBootCluster, FSL_StandBy, FSL_WakeUp, FSL_SwapActiveBootCluster (*2)

(*1) Mapping the FSL_RCD section into RAM when using the SCI, is supported for IAR V2.xx, CC-RL, CA78K0R and LLVM compiler only.

(*2) FSL_SwapActiveBootCluster is available for IAR V2.xx, CC-RL, CA78K0R and LLVM compiler only.
If used, the segment FSL_RCD must be mapped into RAM.

Note For the mapping of ISRs, please refer to Chapter 4.

Table 2-4 Segments in status check user mode (SCU)

Segment name	Segment location	Description
FSL_FCD	ROM	Segment will be used for the following functions: FSL_Init, FSL_Close, FSL_Open, FSL_PrepareFunctions, FSL_PrepareExtFunctions, FSL_ChangeInterruptTable, FSL_RestoreInterruptTable, FSL_GetVersionString and FSL_CopySection
FSL_FECD	ROM	Segment will be used for the following functions: FSL_GetBlockEndAddr, FSL_GetFlashShieldWindow, FSL_GetSwapState, FSL_GetBootFlag and FSL_GetSecurityFlags
FSL_BCD	ROM	Must only be placed via linker file in case FSL_Erase, FSL_Write, FSL_IVerify or FSL_BlankCheck functions are used
FSL_BECD	ROM	Must only be placed via linker file in case FSL_SetXXX and FSL_InvertBootFlag functions are used
FSL_RCD	RAM	Segment contains the following functions: FSL_ForceReset, FSL_SetXXXXFlag, FSL_StatusCheck, FSL_BlankCheck, FSL_Erase, FSL_IVerify, FSL_Write, FSL_SwapBootCluster, FSL_StandBy, FSL_WakeUp, FSL_SwapActiveBootCluster (*1) For IAR V1.xx and GNU, the content needs to be copied into this RAM segment via the FSL_CopySection(*2) function.
FSL_RCD_ROM (*3)	ROM	This segment contains the following functions: FSL_ForceReset, FSL_SetXXXXFlag, FSL_StatusCheck, FSL_BlankCheck, FSL_Erase, FSL_IVerify, FSL_Write, FSL_SwapBootCluster, FSL_StandBy, FSL_WakeUp However, the execution of these functions is done in the FSL_RCD segment which is located in RAM and not ROM. Therefore, please make sure that the content of the FSL_RCD_ROM segment is copied to the FSL_RCD segment (via FSL_CopySection) before execution of these functions.

(*1) FSL_SwapActiveBootCluster is available for IAR V2.xx, CC-RL, CA78K0R and LLVM compiler only.

(*2) FSL_CopySection is required for IAR V1.xx and GNU compiler only.

(*3) The segment FSL_RCD_ROM is defined for IAR V1.xx compiler only.

Note For the mapping of ISRs, please refer to Chapter 4.

Chapter 3 Execution modes

Self-programming library can be executed in two different ways which will be described here.

3.1 Status check internal mode (from ROM or RAM)

This execution mode allows the user to execute FSL_XXX functions directly from ROM or RAM. The function returns only in case it is successfully finished or an error occurred. Means no status polling is required.

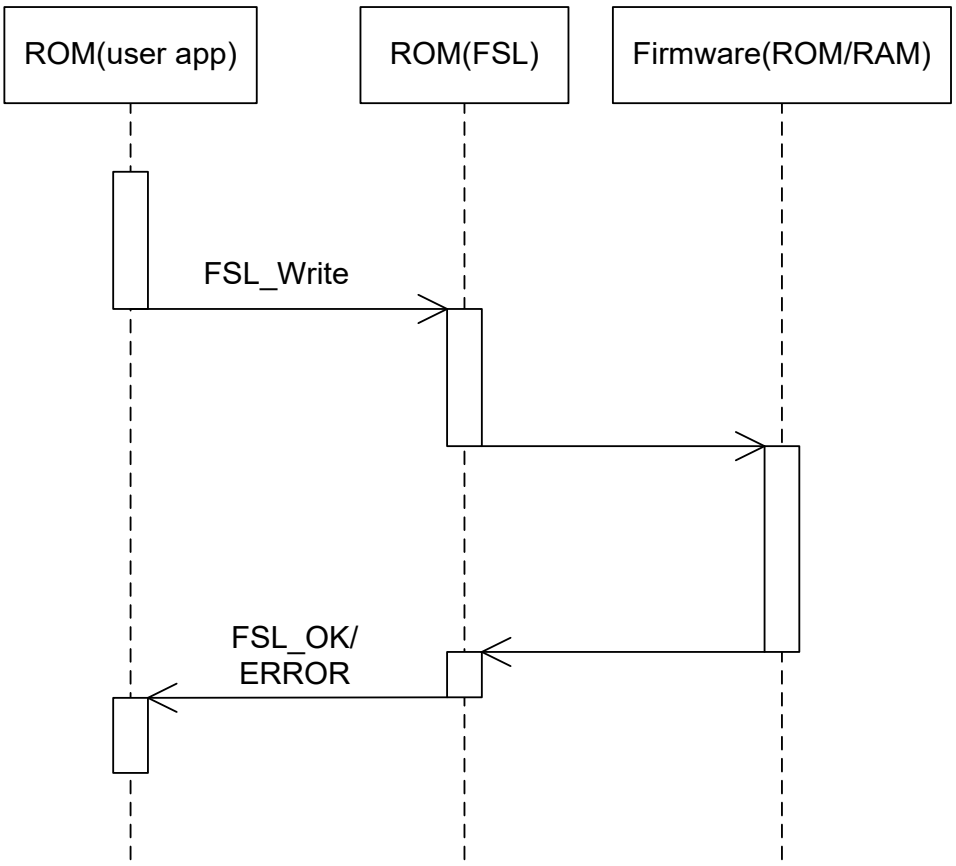

Figure 3-1 Status check internal mode sequence

3.2 Status check user mode (from RAM only)

This execution mode is designed for time critical systems. Means the called FSL_XXX functions return immediately after triggering/checking the hardware to the user for example to reset the watchdog. After that user has to call the FSL_StatusCheck command.

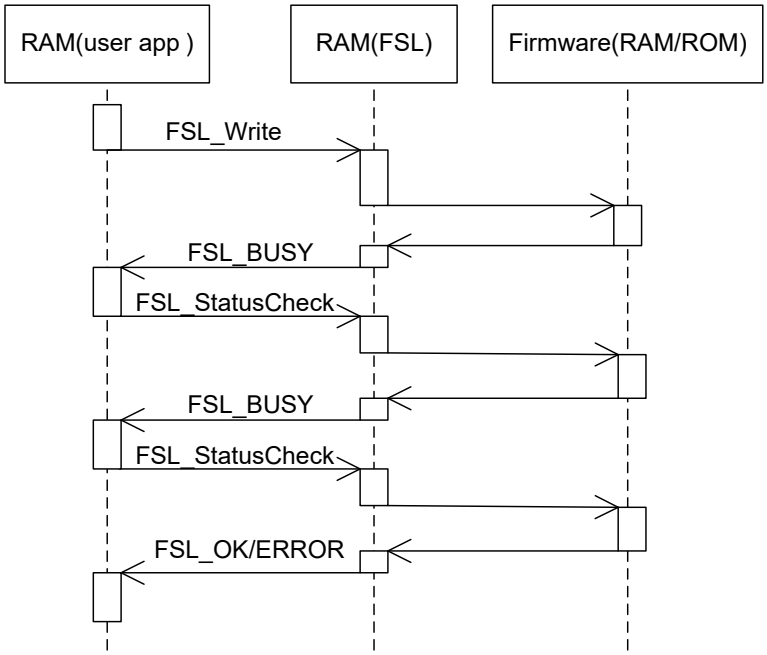

Figure 3-2 Status check user mode sequence

Chapter 4 Interrupt servicing

During execution of Self-programming functions the user cannot use the interrupt service routines located in flash. For that reason the library provides the functions FSL_ChangeInterruptTable and FSL_RestoreInterruptTable which allows the user to execute interrupts from RAM. Means one ISR is located in RAM and all occurred interrupts will be handled by this ISR inside of RAM. The following figure illustrates both scenarios where the interrupts are handled in ROM and in RAM.

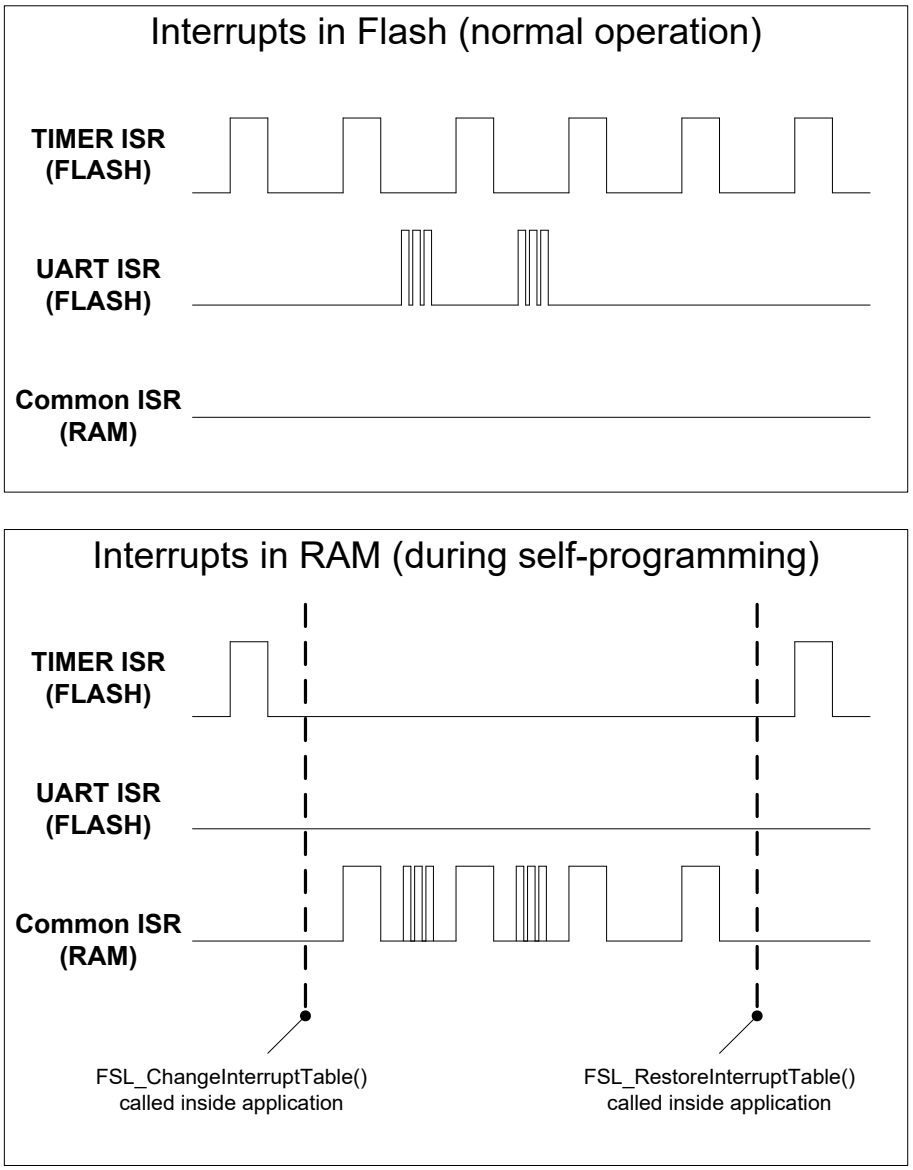

Figure 4-1 Interrupts during normal operation and during Self-programming

As shown in the figure above (Interrupts in RAM) each interrupts will be handled in the RAM ISR. Means the user has to check in the RAM ISR which interrupt source generated the interrupt. This must be done by checking the Interrupt Request Flag Registers.

Note User has to take care that the request flag is cleared before leaving the ISR.

Chapter 5 Boot-swapping

During the re-programming of flash a permanent data loss may occur. This potential risk can be avoided by using the boot swap functionality.

Following events may cause the data loss

- temporary power failure
- externally generated reset

5.1 How to perform a boot-swap?

The FSL library provides a swap function (FSL_InvertBootFlag) which makes it possible to swap the boot cluster.

Before swapping, user program should write the new boot program into boot cluster 1. And then swap the two boot clusters and force a reset. The device will then be restarting from boot cluster 1.

As a result, even if a power failure occurs while the boot program area is being rewritten, the program runs correctly because after reset the circuit starts from boot cluster 1. After that, boot cluster 0 can be erased or written as required.

Note In the following example, a boot cluster size of 4KB is given:

- Boot cluster 0 (0000H to 0FFFH): Original boot program area
- Boot cluster 1 (1000H to 1FFFH): Boot swap target area

The actual boot cluster size is device dependent. Please refer to the device user manual of your target device for details.

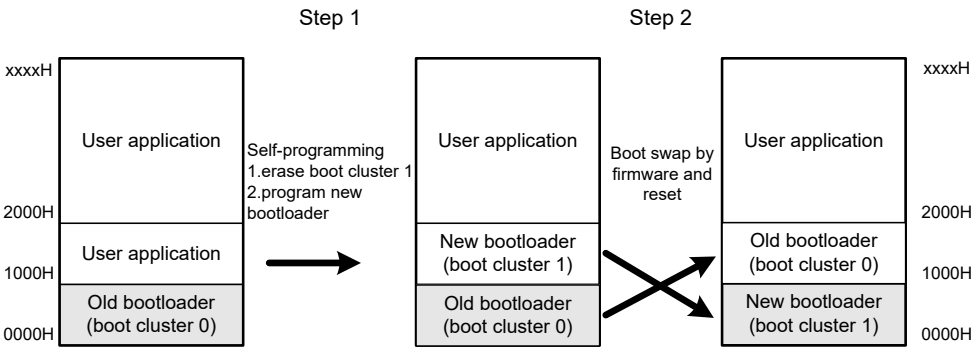

Figure 5-1 Overview of swap procedure

Caution **• To rewrite the flash memory by using a programmer (such as the PG FP5) after boot swapping, please refer to the PG-FP5 users manual.**

• After successful execution of the FSL_InvertBootFlag function it is not allowed to execute any FSL_Setxxx function till hardware reset occurred.

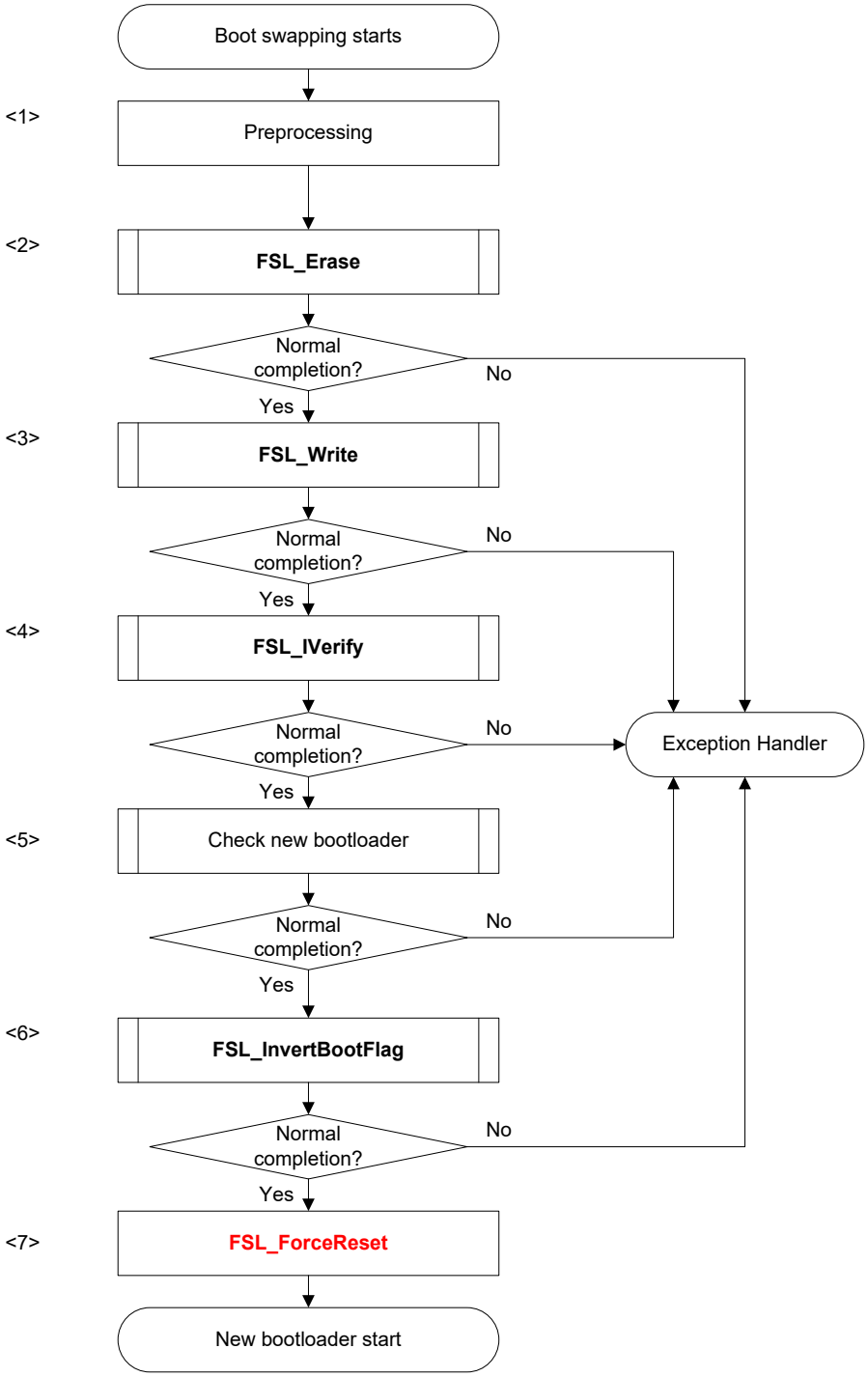

Figure 5-2 Flow of boot swapping

Caution **FSL_ForceReset function generates a software reset (please refer to the device Users Manual for detailed information).**

<1> Preprocessing

The following preprocess of boot swapping is performed.

- Set up software environment
- Set up hardware environment
- Initialize entry RAM

<2> Erasing blocks of boot cluster 1

Call the erase function FSL_Erase to erase blocks 4 to 7.

Note 1 In this example, a boot cluster size of 4KB is given. The actual boot cluster size is device dependent. Please refer to the device user manual of your target device for details.

Note 2 The erase function erases only one block at a time. Call it once for each block.

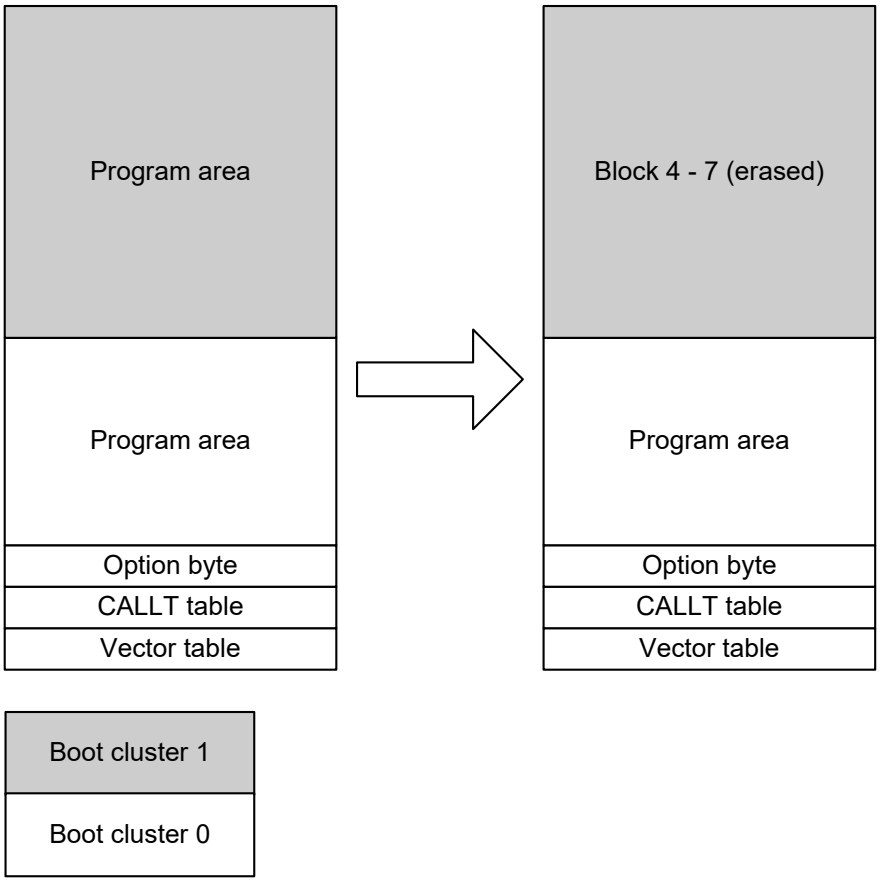

Figure 5-3 Erasing boot cluster 1

<3> Writing new program to boot cluster 1

Use the FSL_Write function to write the new bootloader.

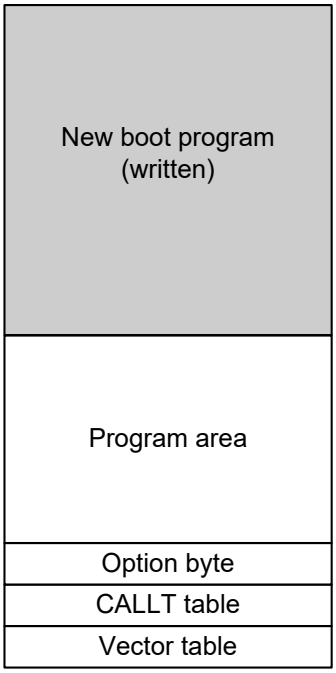

Figure 5-4 Writing new program to boot cluster 1

<4> Verifying blocks of boot cluster 1

Call the verify function FSL_IVerify to verify Blocks 4 to 7.

Note 1 In this example, a boot cluster size of 4KB is given. The actual boot cluster size is device dependent. Please refer to the device user manual of your target device for details.

Note 2 The internal verify function verifies only one block at a time. Call it once for each block.

<5> Checks the new bootloader

e.g. CRC check on the new bootloader.

<6> Setting of boot swap bit

Call the function FSL_InvertBootFlag. The inactive boot cluster with new bootloader becomes active after hardware reset.

<7> Force of reset

Call the FSL_ForceReset function. New bootloader is active after reset.

Chapter 6 User Interface (API)

6.1 Data types

This chapter describes all data definitions used and offered by the FSL.

6.1.1 Library specific simple type definitions

```
typedef unsigned char                   fsl_u08;
typedef unsigned int                    fsl_u16;
typedef unsigned long int               fsl_u32;
```

6.1.2 Structured type “fsl_descriptor_t”

This type defines structure of the FSL descriptor used for FSL_Init function.

Structure member name	Description
fsl_flash_voltage_u08	Setting for the flash memory programming mode: 0 = Full speed mode other = Wide voltage mode
fsl_frequency_u08	***Configured frequency (frequency >= 4MHz)*** Frequency must be rounded up as shown below: descr.fsl_frequency_u08 = 20 for 20000000Hz descr.fsl_frequency_u08 = 24 for 23100000Hz ***Configured frequency (frequency < 4MHz)*** In case the frequency is smaller than 4MHz the only supported physically frequencies are following: descr.fsl_frequency_u08 = 1 for 1000000Hz descr.fsl_frequency_u08 = 2 for 2000000Hz descr.fsl_frequency_u08 = 3 for 3000000Hz
fsl_auto_status_check_u08	Setting for execution mode: 0 = Status check user mode other = Status check internal mode

6.1.3 Structured type “fsl_write_t”

This type defines structure used for FSL_Write function.

Structure member name	Description
fsl_data_buffer_p_u08	Pointer to the data buffer where the data to be written is located.
fsl_destination_address_u32	Destination address: e.g. write.fsl_destination_address_u32 = 0x1000
fsl_word_count_u08	Number of words to be written (1 word = 4 bytes)

6.1.4 Structured type “fsl_getblockendaddr_t”

This type defines structure used for FSL_GetBlockEndAddr function.

Structure member name	Description
fsl_destination_address_u32	Destination address of the block end address info
fsl_block_u16	Block number the end-address is asked for

6.1.5 Structured type “fsl_fsw_t”

This type defines structure of the FSL descriptor used for FSL_SetFlashShieldWindow and FSL_GetFlashShieldWindow function.

Structure member name	Description
fsl_start_block_u16	Start block for the flash shield window
fsl_end_block_u16	End block for the flash shield window

6.2 Functions

This chapter describes all functions provided by the library.

6.2.1 Assembler interface

Following tables describes the parameter passing for the CA78K0R, IAR V1.xx, IAR V2.xx, GNU, CC-RL and LLVM environment.

Table 6-1 Assembler interface for CA78K0R environment

Function	Register used for parameter passing	Register used as return value	Destroyed register after call
FSL_Init	AX(0-15)-C(16-23)	C	n/a
FSL_PrepareFunctions	n/a	n/a	n/a
FSL_PrepareExtFunctions	n/a	n/a	n/a
FSL_ChangeInterruptTable	AX(0-15)	n/a	n/a
FSL_RestoreInterruptTable	n/a	n/a	n/a
FSL_Open	n/a	n/a	n/a
FSL_Close	n/a	n/a	n/a
FSL_BlankCheck	AX(0-15)	C	n/a
FSL_Erase	AX(0-15)	C	n/a
FSL_IVerify	AX(0-15)	C	n/a
FSL_Write	AX(0-15)	C	n/a
FSL_GetSecurityFlags	AX(0-15)	C	n/a
FSL_GetBootFlag	AX(0-15)	C	n/a
FSL_GetSwapState	AX(0-15)	C	n/a
FSL_GetBlockEndAddr	AX(0-15)	C	n/a
FSL_GetFlashShieldWindow	AX(0-15)	C	n/a
FSL_SetBlockEraseProtectFlag	n/a	C	n/a
FSL_SetWriteProtectFlag	n/a	C	n/a
FSL_SetBootClusterProtectFlag	n/a	C	n/a
FSL_InvertBootFlag	n/a	C	n/a
FSL_SetFlashShieldWindow	AX(0-15)	C	n/a
FSL_SwapBootCluster	n/a	C	AX, C, ES, CS
FSL_SwapActiveBootCluster	n/a	C	n/a
FSL_ForceReset	n/a	n/a	n/a
FSL_StatusCheck	n/a	C	n/a
FSL_StandBy	n/a	C	n/a
FSL_WakeUp	n/a	C	n/a
FSL_GetVersionString	n/a	BC(0-15), DE(16-31)	n/a

Table 6-2 Assembler interface for IAR V1.xx environment

Function	Register used for parameter passing	Register used as return value	Destroyed register after call
FSL_Init	Stack	A	n/a
FSL_PrepareFunctions	n/a	n/a	n/a
FSL_PrepareExtFunctions	n/a	n/a	n/a
FSL_ChangeInterruptTable	AX(0-15)	n/a	n/a
FSL_RestoreInterruptTable	n/a	n/a	n/a
FSL_Open	n/a	n/a	n/a
FSL_Close	n/a	n/a	n/a
FSL_BlankCheck	AX(0-15)	A	n/a
FSL_Erase	AX(0-15)	A	n/a
FSL_IVerify	AX(0-15)	A	n/a
FSL_Write	AX(0-15)	A	n/a
FSL_GetSecurityFlags	AX(0-15)	A	n/a
FSL_GetBootFlag	AX(0-15)	A	n/a
FSL_GetSwapState	AX(0-15)	A	n/a
FSL_GetBlockEndAddr	AX(0-15)	A	n/a
FSL_GetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SetBlockEraseProtectFlag	n/a	A	n/a
FSL_SetWriteProtectFlag	n/a	A	n/a
FSL_SetBootClusterProtectFlag	n/a	A	n/a
FSL_InvertBootFlag	n/a	A	n/a
FSL_SetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SwapBootCluster	n/a	A	AX, ES, CS
FSL_ForceReset	n/a	n/a	n/a
FSL_StatusCheck	n/a	A	n/a
FSL_StandBy	n/a	A	n/a
FSL_WakeUp	n/a	A	n/a
FSL_GetVersionString	n/a	HL(0-15), A(16-31)	n/a
FSL_CopySection	n/a	n/a	n/a

Table 6-3 Assembler interface for IAR V2.xx environment

Function	Register used for parameter passing	Register used as return value	Destroyed register after call
FSL_Init	DE(0-15)-A(16-23)	A	n/a
FSL_PrepareFunctions	n/a	n/a	n/a
FSL_PrepareExtFunctions	n/a	n/a	n/a
FSL_ChangeInterruptTable	AX(0-15)	n/a	n/a
FSL_RestoreInterruptTable	n/a	n/a	n/a
FSL_Open	n/a	n/a	n/a
FSL_Close	n/a	n/a	n/a
FSL_BlankCheck	AX(0-15)	A	n/a
FSL_Erase	AX(0-15)	A	n/a
FSL_IVerify	AX(0-15)	A	n/a
FSL_Write	AX(0-15)	A	n/a
FSL_GetSecurityFlags	AX(0-15)	A	n/a
FSL_GetBootFlag	AX(0-15)	A	n/a
FSL_GetSwapState	AX(0-15)	A	n/a
FSL_GetBlockEndAddr	AX(0-15)	A	n/a
FSL_GetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SetBlockEraseProtectFlag	n/a	A	n/a
FSL_SetWriteProtectFlag	n/a	A	n/a
FSL_SetBootClusterProtectFlag	n/a	A	n/a
FSL_InvertBootFlag	n/a	A	n/a
FSL_SetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SwapBootCluster	n/a	A	AX, ES, CS
FSL_SwapActiveBootCluster	n/a	A	n/a
FSL_ForceReset	n/a	n/a	n/a
FSL_StatusCheck	n/a	A	n/a
FSL_StandBy	n/a	A	n/a
FSL_WakeUp	n/a	A	n/a
FSL_GetVersionString	n/a	DE(0-15), A(16-23)	n/a

Table 6-4 Assembler interface for GNU environment

Function	Register used for parameter passing	Register used as return value	Destroyed register after call
FSL_Init	Stack	R8	n/a
FSL_PrepareFunctions	n/a	n/a	n/a
FSL_PrepareExtFunctions	n/a	n/a	n/a
FSL_ChangeInterruptTable	Stack	n/a	n/a
FSL_RestoreInterruptTable	n/a	n/a	n/a
FSL_Open	n/a	n/a	n/a
FSL_Close	n/a	n/a	n/a
FSL_BlankCheck	Stack	R8	n/a
FSL_Erase	Stack	R8	n/a
FSL_IVerify	Stack	R8	n/a
FSL_Write	Stack	R8	n/a
FSL_GetSecurityFlags	Stack	R8	n/a
FSL_GetBootFlag	Stack	R8	n/a
FSL_GetSwapState	Stack	R8	n/a
FSL_GetBlockEndAddr	Stack	R8	n/a
FSL_GetFlashShieldWindow	Stack	R8	n/a
FSL_SetBlockEraseProtectFlag	n/a	R8	n/a
FSL_SetWriteProtectFlag	n/a	R8	n/a
FSL_SetBootClusterProtectFlag	n/a	R8	n/a
FSL_InvertBootFlag	n/a	R8	n/a
FSL_SetFlashShieldWindow	Stack	R8	n/a
FSL_SwapBootCluster	n/a	R8	AX, ES, CS
FSL_ForceReset	n/a	n/a	n/a
FSL_StatusCheck	n/a	R8	n/a
FSL_StandBy	n/a	R8	n/a
FSL_WakeUp	n/a	R8	n/a
FSL_GetVersionString	n/a	R8-R9 (0-15) R10-R11 (16-23)	n/a
FSL_CopySection	n/a	n/a	n/a

R8 = X on register bank 1
R8-R9 = AX on register bank 1
R10-R11 = BC on register bank 1

Table 6-5 Assembler interface for CC-RL environment

Function	Register used for parameter passing	Register used as return value	Destroyed register after call
FSL_Init	DE(0-15)-A(16-23)	A	n/a
FSL_PrepareFunctions	n/a	n/a	n/a
FSL_PrepareExtFunctions	n/a	n/a	n/a
FSL_ChangeInterruptTable	AX(0-15)	n/a	n/a
FSL_RestoreInterruptTable	n/a	n/a	n/a
FSL_Open	n/a	n/a	n/a
FSL_Close	n/a	n/a	n/a
FSL_BlankCheck	AX(0-15)	A	n/a
FSL_Erase	AX(0-15)	A	n/a
FSL_IVerify	AX(0-15)	A	n/a
FSL_Write	AX(0-15)	A	n/a
FSL_GetSecurityFlags	AX(0-15)	A	n/a
FSL_GetBootFlag	AX(0-15)	A	n/a
FSL_GetSwapState	AX(0-15)	A	n/a
FSL_GetBlockEndAddr	AX(0-15)	A	n/a
FSL_GetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SetBlockEraseProtectFlag	n/a	A	n/a
FSL_SetWriteProtectFlag	n/a	A	n/a
FSL_SetBootClusterProtectFlag	n/a	A	n/a
FSL_InvertBootFlag	n/a	A	n/a
FSL_SetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SwapBootCluster	n/a	A	AX, ES, CS
FSL_SwapActiveBootCluster	n/a	A	n/a
FSL_ForceReset	n/a	n/a	n/a
FSL_StatusCheck	n/a	A	n/a
FSL_StandBy	n/a	A	n/a
FSL_WakeUp	n/a	A	n/a
FSL_GetVersionString	n/a	DE(0-15), A(16-23)	n/a

Table 6-6 Assembler interface for LLVM environment

Function	Register used for parameter passing	Register used as return value	Destroyed register after call
FSL_Init	DE(0-15)-A(16-23)	A	n/a
FSL_PrepareFunctions	n/a	n/a	n/a
FSL_PrepareExtFunctions	n/a	n/a	n/a
FSL_ChangeInterruptTable	AX(0-15)	n/a	n/a
FSL_RestoreInterruptTable	n/a	n/a	n/a
FSL_Open	n/a	n/a	n/a
FSL_Close	n/a	n/a	n/a
FSL_BlankCheck	AX(0-15)	A	n/a
FSL_Erase	AX(0-15)	A	n/a
FSL_IVerify	AX(0-15)	A	n/a
FSL_Write	AX(0-15)	A	n/a
FSL_GetSecurityFlags	AX(0-15)	A	n/a
FSL_GetBootFlag	AX(0-15)	A	n/a
FSL_GetSwapState	AX(0-15)	A	n/a
FSL_GetBlockEndAddr	AX(0-15)	A	n/a
FSL_GetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SetBlockEraseProtectFlag	n/a	A	n/a
FSL_SetWriteProtectFlag	n/a	A	n/a
FSL_SetBootClusterProtectFlag	n/a	A	n/a
FSL_InvertBootFlag	n/a	A	n/a
FSL_SetFlashShieldWindow	AX(0-15)	A	n/a
FSL_SwapBootCluster	n/a	A	AX, ES, CS
FSL_SwapActiveBootCluster	n/a	A	n/a
FSL_ForceReset	n/a	n/a	n/a
FSL_StatusCheck	n/a	A	n/a
FSL_StandBy	n/a	A	n/a
FSL_WakeUp	n/a	A	n/a
FSL_GetVersionString	n/a	DE(0-15), A(16-23)	n/a

6.2.2 FSL_CopySection (IAR V1.xx and GNU Compiler only)

As described in chapter 3.2 (status check user mode) the library provides a possibility to execute the user code during the execution of FSL commands (e.g. erase, write, ...). However, it is only possible if the code to be executed (during the FSL functions execution) is located in RAM. To make the copy process quite simple, the library provides this function to copy the code to RAM.

Caution: The code segment (FSL_RCD) placed in the RAM shall be extended by 10bytes to avoid read accesses of uninitialized RAM.

C interface for IAR V1.xx compiler

```
__far_func  void FSL_CopySection(void);
```

C interface for GNU compiler

```
void  FSL_CopySection(void)
                    __attribute__  ((section ("FSL_FCD")));
```

Pre-condition (IAR V1.xx compiler)

Following segments shall be defined:

FSL_RCD = copy-target segment located in RAM + extended by 10 bytes
FSL_RCD_ROM = copy-source segment located in ROM

Example linker file contents:

-DFSL_RCD_WORK=0x0A
-Z(DATA)FSL_RCD+FSL_RCD_WORK=[FF300-FF6FF]/10000
-Z(CODE)FSL_RCD_ROM=[2000-23FF]/10000
-QFSL_RCD=FSL_RCD_ROM

Pre-condition (GNU compiler)

Following segment shall be defined:

FSL_RCD = copy-target segment located in RAM + extended by 10 bytes

Following symbols shall be defined in linker file:

FSL_RCD_RAM_START_ADDR = start address of FSL_RCD segment

FSL_RCD_CP_SIZE = copy size

FSL_RCD_ROM_START_ADDR = start address in flash segment for copy process

FSL_RCD_WORK = fixed bytes count for RAM segment extension

Example linker file contents:

```
FSL_RCD_WORK = 0xA;
FSL_RCD_RAM_ADDR = 0x00FF300;

FSL_RCD FSL_RCD_RAM_ADDR : AT (LOADADDR(<<previous_flash_segment>>) +
SIZEOF(<<previous_flash_segment>>))
{
   FSL_RCD_ROM_START_ADDR = .;
   *(FSL_RCD)
   . = .  + FSL_RCD_WORK;
   FSL_RCD_ROM_END_ADDR = .;
}
_FSL_RCD_WORK = FSL_RCD_WORK ;
_FSL_RCD_RAM_START_ADDR = FSL_RCD_RAM_ADDR ;
_FSL_RCD_RAM_END_ADDR = _FSL_RCD_RAM_START_ADDR + SIZEOF(FSL_RCD);
_FSL_RCD_ROM_START_ADDR = LOADADDR(FSL_RCD) ;
_FSL_RCD_ROM_END_ADDR = _FSL_RCD_ROM_START_ADDR + SIZEOF(FSL_RCD);
_FSL_RCD_CP_SIZE = SIZEOF(FSL_RCD) + FSL_RCD_WORK ;
```

Post-condition

Data are copied from ROM to RAM.

Argument

Argument	Type	Description
None		

Return types/values

Value	Type	Description
None		

Code example:

```
FSL_CopySection();
```

6.2.3 FSL_Init

This function initializes internal Self-programming environment. After initialization the start address of the data-buffer is registered for Self-programming.

C interface for CA78K0R compiler

```
fsl_u08  FSL_Init(__far fsl_descriptor_t* descriptor_pstr);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_Init(const __far fsl_descriptor_t
                                        __far* descriptor_pstr);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_Init(const fsl_descriptor_t
                                       __far * descriptor_pstr);
```

C interface for GNU compiler

```
fsl_u08  FSL_Init(__far fsl_descriptor_t __far *descriptor_pstr)
                          __attribute__ ((section ("FSL_FCD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far FSL_Init(const __far fsl_descriptor_t*
                                       descriptor_pstr);
```

C interface for LLVM compiler

```
fsl_u08 __far FSL_Init(const __far fsl_descriptor_t*
                                        descriptor_pstr)
                          __attribute__ ((section ("FSL_FCD")));
```

Pre-condition

Descriptor must be filled with valid values.
Internal high-speed oscillator is running.

Post-condition

FSL is initialized.

Argument

Argument	Type	Description
descriptor_pstr	fsl_descriptor_t* (far pointer)	This argument must be a pointer to the descriptor.

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK)	fsl_u08	FSL initialized
0x05 (FSL_ERR_PARAMETER)	fsl_u08	wrong parameter passed via descriptor or internal high-speed oscillator isn't started.

Code example (IAR V1.xx, IAR V2.xx, CA78K0R, CC-RL and LLVM compiler):

```
fsl_u08              my_fsl_status;
fsl_descriptor_t     fsl_descr;

fsl_descr.fsl_flash_voltage_u08 = 0x00;
fsl_descr.fsl_frequency_u08 = 0x14;
fsl_descr.fsl_auto_status_check_u08 = 0x01;

my_fsl_status = FSL_Init((__far fsl_descriptor_t*)&fsl_descr);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

Code example (GNU compiler):

```
fsl_u08              my_fsl_status;
fsl_descriptor_t     fsl_descr;

fsl_descr.fsl_flash_voltage_u08 = 0x00;
fsl_descr.fsl_frequency_u08 = 0x14;
fsl_descr.fsl_auto_status_check_u08 = 0x01;

my_fsl_status = FSL_Init((__far fsl_descriptor_t __far *)
                                                  &fsl_descr);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

6.2.4 FSL_Open

This function opens the Self-programming session.

C interface for CA78K0R compiler

```
void  FSL_Open(void);
```

C interface for IAR V1.xx compiler

```
__far_func void  FSL_Open(void);
```

C interface for IAR V2.xx compiler

```
__far_func void  FSL_Open(void);
```

C interface for GNU compiler

```
void  FSL_Open(void) __attribute__ ((section ("FSL_FCD")));
```

C interface for CC-RL compiler

```
void __far FSL_Open(void);
```

C interface for LLVM compiler

```
void __far FSL_Open(void) __attribute__ ((section ("FSL_FCD")));
```

Pre-condition

Library must be initialized via FSL_Init

Post-condition

None

Argument

Argument	Type	Description
none		

Return types/values

Value	Type	Description
None		

Code example:

```
FSL_Open();
```

6.2.5 FSL_Close

This function closes the Self-programming session.

C interface for CA78K0R compiler

```
void FSL_Close(void);
```

C interface for IAR V1.xx compiler

```
__far_func  void  FSL_Close(void);
```

C interface for IAR V2.xx compiler

```
__far_func  void  FSL_Close(void);
```

C interface for GNU compiler

```
void  FSL_Close(void) __attribute__ ((section ("FSL_FCD")));
```

C interface for CC-RL compiler

```
void __far FSL_Close(void);
```

C interface for LLVM compiler

```
void __far FSL_Close(void)
                          __attribute__ ((section ("FSL_FCD")));
```

Pre-condition

None

Post-condition

Self-programming cannot be performed.

Argument

Argument	Type	Description
none		

Return types/values

Value	Type	Description
None		

Code example:

```
FSL_Close();
```

6.2.6 FSL_PrepareFunctions

This function activates a set (FSL_BlankCheck, FSL_Erase, FSL_Write, FSL_IVerify, FSL_StatusCheck, FSL_StandBy, FSL_WakeUp) of FSL functions which can be accessed by the user afterwards.

C interface for CA78K0R compiler

```
void  FSL_PrepareFunctions(void);
```

C interface for IAR V1.xx compiler

```
__far_func void  FSL_PrepareFunctions(void);
```

C interface for IAR V2.xx compiler

```
__far_func void  FSL_PrepareFunctions(void);
```

C interface for GNU compiler

```
void  FSL_PrepareFunctions(void)
                          __attribute__ ((section ("FSL_FCD")));
```

C interface for CC-RL compiler

```
void __far FSL_PrepareFunctions(void);
```

C interface for LLVM compiler

```
void __far FSL_PrepareFunctions(void)
                          __attribute__ ((section ("FSL_FCD")));
```

Pre-condition

Library must be initialized via FSL_Init

Post-condition

Following functions can be used by user:
- FSL_BlankCheck
- FSL_Erase,
- FSL_Write,
- FSL_IVerify,
- FSL_StatusCheck,
- FSL_StandBy,
- FSL_WakeUp

Argument

Argument	Type	Description
none		

Return types/values

Value	Type	Description
None		

Code example:

```
FSL_PrepareFunctions();
```

6.2.7 FSL_PrepareExtFunctions

This function activates a set of functions (FSL_SetXXXProtectFlag, FSL_InvertBootFlag, FSL_SetFlashShieldWinow, FSL_SwapBootCluster, FSL_SwapActiveBootCluster).

C interface for CA78K0R compiler

```
void  FSL_PrepareExtFunctions(void);
```

C interface for IAR V1.xx compiler

```
__far_func void  FSL_PrepareExtFunctions(void);
```

C interface for IAR V2.xx compiler

```
__far_func void  FSL_PrepareExtFunctions(void);
```

C interface for GNU compiler

```
void  FSL_PrepareExtFunctions(void)
                          __attribute__ ((section ("FSL_FCD")));
```

C interface for CC-RL compiler

```
void __far FSL_PrepareExtFunctions(void);
```

C interface for LLVM compiler

```
void _far FSL_PrepareExtFunctions(void)
                          __attribute__ ((section ("FSL_FCD")));
```

Pre-condition

Library must be initialized via FSL_Init

Post-condition

Following functions can be used by user:
- FSL_SetXXXProtectFlag
- FSL_InvertBootFlag,
- FSL_SetFlashShieldWindow,
- FSL_SwapBootCluster,
- FSL_SwapActiveBootCluster (CA78K0R, IAR V2.xx, CC-RL and LLVM Compiler only)

Argument

Argument	Type	Description
none		

Return types/values

Value	Type	Description
None		

Code example:

```
FSL_PrepareExtFunctions();
```

6.2.8 FSL_ChangeInterruptTable

This function deactivates all interrupt vectors and configures only one common ISR located in RAM for all interrupts. Means each interrupt will be handled by only one ISR.

C interface for CA78K0R compiler

```
void FSL_ChangeInterruptTable(fsl_u16
                                  fsl_interrupt_destination_u16);
```

C interface for IAR V1.xx compiler

```
__far_func void  FSL_ChangeInterruptTable(fsl_u16
                                  fsl_interrupt_destination_u16);
```

C interface for IAR V2.xx compiler

```
__far_func void  FSL_ChangeInterruptTable(fsl_u16
                                  fsl_interrupt_destination_u16);
```

C interface for GNU compiler

```
void  FSL_ChangeInterruptTable(fsl_u16
                                   fsl_interrupt_destination_u16)
                           __attribute__ ((section("FSL_FCD")));
```

C interface for CC-RL compiler

```
void __far FSL_ChangeInterruptTable(fsl_u16
                                  fsl_interrupt_destination_u16);
```

C interface for LLVM compiler

```
void __far FSL_ChangeInterruptTable(fsl_u16
                                   fsl_interrupt_destination_u16)
                          __attribute__ ((section("FSL_FCD")));
```

Pre-condition

Library must be initialized via FSL_Init.
Common interrupt service routine must be copied into the RAM.
Interrupts must be disabled during the execution of this function.

Post-condition

All Interrupts will be handled inside one ISR located in RAM. Please be aware of the following consequences:

- Due to the fact that there is only one ISR for all activated interrupts, the interrupt source should be determined by the interrupt request flags.
- Before leaving the ISR function, the related interrupt request flag shall be cleared manually by the user.

Argument

Argument	Type	Description
fsl_interrupt_destination_u16	fsl_u16	The address of the common ISR located in RAM. **Note: The high address of the interrupt vector is fixed to 0x000F means the ISR is located in RAM.**

Return types/values

Value	Type	Description
None		

Code example (IAR V1.xx, IAR V2.xx, CA78K0R and CC-RL Compiler):

```
__interrupt void isr_RAM(void)
{
  ........
  if(TMIF01 == 1){
    ...
    TMIF01=0;
  }
  ........
}

void main(void)
{
  .....
  FSL_ChangeInterruptTable((fsl_u16)&isr_RAM);
  .....
  FSL_XXX(....);
  .....
  FSL_RestoreInterruptTable();
}
```

Code example (GNU Compiler):

```
/* Please refer to chapter 9, Usage of FSL_ChangeInterruptTable
for GNU compiler for a more detailed explanation */
/* File: usr_isr.c */
........
void __attribute__ ((interrupt)) isr_RAM(void)
{
  ........
  if(TMIF01 == 1){
    ...
    TMIF01=0;
  }
  ........
}
........
/* end of file */

/* File: usr_main.c */
extern fsl_u08 __far isr_RAM[];

void main(void)
{
  .....
  FSL_ChangeInterruptTable((fsl_u16)&isr_RAM);
  .....
  FSL_XXX(....);
  .....
  FSL_RestoreInterruptTable();
}
```

Code example (LLVM Compiler):

```
/* Please refer to chapter 9, Usage of FSL_ChangeInterruptTable
for LLVM compiler for a more detailed explanation */
/* File: usr_isr.c */
........
void __far isr_RAM(void) __attribute__ ((interrupt));

void __far isr_RAM(void)
{
  ........
  if(TMIF01 == 1){
    ...
    TMIF01=0;
  }
  ........
}

void main(void)
{
  .....
  FSL_ChangeInterruptTable((fsl_u16)&isr_RAM);
  .....
  FSL_XXX(....);
  .....
  FSL_RestoreInterruptTable();
}
```

6.2.9 FSL_RestoreInterruptTable

This function restores the original interrupt vector table located in flash. Means each interrupt will be handled by its own ISR.

C interface for CA78K0R compiler

```
void FSL_RestoreInterruptTable(void);
```

C interface for IAR V1.xx compiler

```
__far_func void  FSL_RestoreInterruptTable(void);
```

C interface for IAR V2.xx compiler

```
__far_func void  FSL_RestoreInterruptTable(void);
```

C interface for GNU compiler

```
void  FSL_RestoreInterruptTable(void)
                          __attribute__ ((section ("FSL_FCD")));
```

C interface for CC-RL compiler

```
void __far FSL_RestoreInterruptTable(void);
```

C interface for LLVM compiler

```
void __far FSL_RestoreInterruptTable(void)
                          __attribute__ ((section ("FSL_FCD")));
```

Pre-condition

Library must be initialized via FSL_Init.
Interrupt handling is changed by FSL_ChangeInterruptTable.
Interrupts must be disabled during the execution of this function.

Post-condition

All Interrupts will be handled inside by its own ISR located in flash.

Argument

Argument	Type	Description
None		

Return types/values

Value	Type	Description
None		

Code example:

```
FSL_RestoreInterruptTable();
```

6.2.10 FSL_StatusCheck

This function is used by the application to proceed the execution of a command running in the background.

C interface for CA78K0R compiler

```
fsl_u08  FSL_StatusCheck(void);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_StatusCheck(void);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_StatusCheck(void);
```

C interface for GNU compiler

```
fsl_u08  FSL_StatusCheck(void)
                          __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_StatusCheck(void);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_StatusCheck(void)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions and/or FSL_PrepareExtFunctions

Post-condition

Current status of the running command is returned to the caller.

Argument

Argument	Type	Description
None		

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion
0x1A (FSL_ERR_ERASE) (*1)		Erase error Block couldn't be erased.
0x1B (FSL_ERR_BLANKCHECK) (*1)		Blank check error Specified block is not blank
0x1B (FSL_ERR_IVERIFY) (*1)		Verify error Data inside the flash memory is not at a sufficient voltage level
0x1C (FSL_ERR_WRITE) (*1)		Write error Data couldn't be written to the specified address
0x1F (FSL_ERR_FLOW)		Possible errors: - Violates the precondition - FSL is suspending (*1)
0x30 (FSL_IDLE) (*1)		Driver is idle. No command is running.
0xFF (FSL_BUSY) (*1)		Command is running

(*1) For status check user mode only

Code example:

```
/* example for status check user mode */
fsl_u08              my_fsl_status;

my_fsl_status = FSL_Erase(0x0001);

while(my_fsl_status == FSL_BUSY)
{
  my_fsl_status = FSL_StatusCheck();
}

if(my_fsl_status != FSL_OK) MyErrorHandler();
```

6.2.11 FSL_StandBy

This function suspends a running flash command like FSL_Erase, FSL_Write etc. and sets the library in standby mode.

Only FSL_Erase command is really suspended from the Flash hardware point of view. If the FSL_StandBy function is called during execution of other FSL commands than FSL_Erase, then the library enters in standby mode only after the ongoing command is completed.

Caution This function can be used in status check user mode only.

C interface for CA78K0R compiler

```
fsl_u08  FSL_StandBy(void);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_StandBy(void);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_StandBy(void);
```

C interface for GNU compiler

```
fsl_u08  FSL_StandBy(void)
                        __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_StandBy(void);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_StandBy(void)
                        __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions and/or FSL_PrepareExtFunctions

Post-condition

All flash operations were stopped and the FSL is in StandBy mode.

Argument

Argument	Type	Description
None		

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion. FSL is suspended.
0x1A (FSL_ERR_ERASE) (*1)		Running erase is stopped with erase error. Block couldn't be erased. FSL is suspended.
0x1B (FSL_ERR_BLANKCHECK) (*1)		Running blank check is stopped with blank check error. Specified block is not blank (erase operation is not completed). FSL is suspended.
0x1B (FSL_ERR_IVERIFY) (*1)		Running verify is stopped with verify error. Data inside the flash memory is not at a sufficient voltage level. FSL is suspended.
0x1C (FSL_ERR_WRITE) (*1)		Running write is stopped with write error. Data couldn't be written to the specified address. FSL is suspended.
0x1F (FSL_ERR_FLOW)		Possible errors: - Violates the precondition - FSL is already suspended (*1)
0x30 (FSL_IDLE) (*1)		Driver is idle. No command is running.
0x43 (FSL_SUSPEND) (*1)		Previous flash action is suspended.

(*1) For status check user mode only

Code example:

```
/* example for status check user mode */
fsl_u08             my_fsl_status;

my_fsl_status = FSL_Erase(0x0001);
.....
.....
my_fsl_status = FSL_StandBy();

if( (my_fsl_status == FSL_OK) || (my_fsl_status == FSL_SUSPEND))
{
  /* go into STOP mode if necessary*/
}
else
{
  MyErrorHandler();
}
```

6.2.12 FSL_WakeUp

This function resumes a previous suspended flash command (suspended by FSL_StandBy function).

Caution This function can be used in status check user mode only.

C interface for CA78K0R compiler

```
fsl_u08  FSL_WakeUp(void);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_WakeUp(void);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_WakeUp(void);
```

C interface for GNU compiler

```
fsl_u08  FSL_WakeUp(void)
                    __attribute__ ((section("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_WakeUp(void);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_WakeUp(void)
                    __attribute__ ((section("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions

FSL must be suspended by FSL_StandBy function.

Post-condition

FSL is ready for use.

Argument

Argument	Type	Description
None		

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion. FSL is waked up and previous running command is successfully finished.
0x1A (FSL_ERR_ERASE) (*1)		FSL is waked up and previous running erase command is failed. Block couldn't be erased.
0x1F (FSL_ERR_FLOW)		Possible errors: - Violates the precondition - FSL is not suspended (*1)
0xFF (FSL_BUSY) (*1)		Previous flash action is resumed.

(*1) For status check user mode only

Code example:

```
/* example for status check user mode */
fsl_u08             my_fsl_status;

my_fsl_status = FSL_Erase(0x0001);
.....
.....
my_fsl_status = FSL_StandBy();
.....
.....
my_fsl_status = FSL_WakeUp();

if(my_fsl_status == FSL_BUSY)
{
  /* Continue calling FSL_StatusCheck for finishing the erase */
}
.....
```

6.2.13 FSL_BlankCheck

This function checks if a specified block is blank (erased).

C interface for CA78K0R compiler

```
fsl_u08  FSL_BlankCheck(fsl_u16 block_u16);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_BlankCheck(fsl_u16 block_u16);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_BlankCheck(fsl_u16 block_u16);
```

C interface for GNU compiler

```
fsl_u08  FSL_BlankCheck(fsl_u16 block_u16)
                        __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_BlankCheck(fsl_u16 block_u16);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_BlankCheck(fsl_u16 block_u16)
                        __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions

Post-condition

In case of status check user mode:

- Blank check command is running

In case of status check internal mode:

- Blank check command is finished

Argument

Argument	Type	Description
block_u16	fsl_u16	Block number to be checked

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion Specified block is blank (erased)
0x05 (FSL_ERR_PARAMETER)		Parameter error Specified block number is outside the allowed range
0x1B (FSL_ERR_BLANKCHECK)(*1)		Blank check error Specified block is not blank
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished(*2) - Violates the precondition - FSL is suspending (*2)
0xFF (FSL_BUSY) (*2)		- Command is running

(*1) For status check internal mode only

(*2) For status check user mode only

Code example:

```
/* example for status check internal mode */
fsl_u08             my_fsl_status;

my_fsl_status = FSL_BlankCheck(0x0001);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

6.2.14 FSL_IVerify

This function verifies (internal verification) a specified block.

Note - Because only one block is verified at a time, call this function once for each block.
- This internal verification is a function to check if the flash cell levels are such that the full data retention is ensured.
- It is different from a logical verification that just compares data.

C interface for CA78K0R compiler

```
fsl_u08  FSL_IVerify(fsl_u16 block_u16);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_IVerify(fsl_u16 block_u16);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_IVerify(fsl_u16 block_u16);
```

C interface for GNU compiler

```
fsl_u08  FSL_IVerify(fsl_u16 block_u16) __attribute__
                                        ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_IVerify(fsl_u16 block_u16);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_IVerify(fsl_u16 block_u16)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions

Post-condition

In case of status check user:

- Internal verify command is running

In case of status check internal:

- Internal verify command is finished

Argument

Argument	Type	Description
block_u16	fsl_u16	Block number to be checked

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion Specified block is blank (erased)
0x05 (FSL_ERR_PARAMETER)		Parameter error Specified block number is outside the allowed range
0x1B (FSL_ERR_IVERIFY) (*1)		Verify error Data inside the flash memory is not at a sufficient voltage level
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*2) - Violates the precondition - FSL is suspending (*2)
0xFF (FSL_BUSY)(*2)		- Command is running

(*1) For status check internal mode only

(*2) For status check user mode only

Code example:

```
/* example for status check internal mode */
fsl_u08             my_fsl_status;

my_fsl_status = FSL_IVerify(0x0001);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

6.2.15 FSL_Erase

This function erases a specified block.

Note Because only one block is erased at a time, call this function once for each block.

C interface for CA78K0R compiler

```
fsl_u08  FSL_Erase(fsl_u16 block_u16);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_Erase(fsl_u16 block_u16);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_Erase(fsl_u16 block_u16);
```

C interface for GNU compiler

```
fsl_u08  FSL_Erase(fsl_u16 block_u16)
                        __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_Erase(fsl_u16 block_u16);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_Erase(fsl_u16 block_u16)
                        __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions

Post-condition

In case of status check user mode:

- Erase command is running

In case of status check internal mode:

- Erase command is finished

Argument

Argument	Type	Description
block_u16	fsl_u16	Block number to be checked

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion Specified block is blank (erased)
0x10 (FSL_ERR_PROTECTION)		Specified block is included in the boot area and rewriting the boot area is disabled or block is outside the flash shield window.
0x05 (FSL_ERR_PARAMETER)		Parameter error Specified block number is outside the allowed range
0x1A (FSL_ERR_ERASE) (*1)		Erase error Block couldn't be erased.
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*2) - Violates the precondition - FSL is suspending (*2)
0xFF (FSL_BUSY)(*2)		- Command is running

(*1) For status check internal mode only

(*2) For status check user mode only

Code example:

```
/* example for status check internal mode */

fsl_u08              my_fsl_status;

my_fsl_status = FSL_Erase(0x0001);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

6.2.16 FSL_Write

This function writes the specified number of words (each word consists of 4 bytes) to a specified address.

Note - Set a RAM area as a data buffer, containing the data to be written and call this function.
- Data of up to 256 bytes (i.e. 64 words) can be written at one time.
- Call this function as many times as required to write data of more than 256 bytes.

Caution - After writing data, execute verification (internal verification) of the block including the range in which the data has been written. If verification is not executed, the written data is not guaranteed.
- It is not allowed to overwrite data in flash memory.
- Only blank flash cells can be used for the write.

C interface for CA78K0R compiler

```
fsl_u08  FSL_Write(__near fsl_write_t* write_pstr);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_Write(__near fsl_write_t __near*
                                                write_pstr);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_Write(fsl_write_t __near * write_pstr);
```

C interface for GNU compiler

```
fsl_u08  FSL_Write(fsl_write_t* write_pstr)
                          __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_Write(__near fsl_write_t* write_pstr);
```

C interface for LLVM compiler

```
fsl_u08  __far  FSL_Write(__near fsl_write_t* write_pstr)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions

Data buffer must be filled with data to be written.

Post-condition

In case of status check user mode:

- Write command is running

In case of status check internal mode:

- Write command is finished

Argument

Argument	Type	Description
write_pstr	fsl_write_t* (near pointer)	Pointer to the request structure containing necessary information for the write command.

Note - write_pstr.fsl_destination_address_u32 + (write_pstr.fsl_word_count_u08x4) must not straddle over the end address of a single block.

- write_pstr.fsl_destination_address_u32 must be a multiple of 4

- Most significant byte (MSB) of the write_pstr.fsl_destination_address_u32 has to be 0x00. In other words, only 0x00abcdef is a valid flash address.

- word_count*4 has to be less or equal than the size of data buffer. The firmware does not check this.

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion Specified block is blank (erased)
0x05 (FSL_ERR_PARAMETER)		Parameter error Specified address is outside the allowed range
0x10 (FSL_ERR_PROTECTION)		Protection error Specified address is inside of protected boot cluster or outside the flash shield window
0x1C (FSL_ERR_WRITE) (*1)		Write error Data couldn't be written to the specified address
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*2) - Violates the precondition - FSL is suspending (*2)
0xFF (FSL_BUSY)(*2)		- Command is running

(*1) For status check internal mode only

(*2) For status check user mode only

Code example (IAR V1.xx and IAR V2.xx compiler):

```
/* example for status check internal mode */

__near fsl_write_t   my_fsl_write_str;
fsl_u08              my_fsl_status;


my_fsl_write_str.fsl_data_buffer_p_u08 =
                   (fsl_u08 __near*)fsl_data_buffer_u08;
my_fsl_write_str.fsl_word_count_u08 = 0x01;
my_fsl_write_str.fsl_destination_address_u32 = 0x00001000;

my_fsl_status = FSL_Write((fsl_write_t __near*)
&my_fsl_write_str);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

Code example (CA78K0R, CC-RL and LLVM compiler):

```
/* example for status check internal mode */

__near fsl_write_t   my_fsl_write_str;
fsl_u08              my_fsl_status;


my_fsl_write_str.fsl_data_buffer_p_u08 =
                   (__near fsl_u08 *)fsl_data_buffer_u08;
my_fsl_write_str.fsl_word_count_u08 = 0x01;
my_fsl_write_str.fsl_destination_address_u32 = 0x00001000;

my_fsl_status = FSL_Write((__near fsl_write_t*)
&my_fsl_write_str);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

Code example (GNU compiler):

```
/* example for status check internal mode */

fsl_write_t          my_fsl_write_str;
fsl_u08              my_fsl_status;


my_fsl_write_str.fsl_data_buffer_p_u08 =
                   (fsl_u08 *)fsl_data_buffer_u08;
my_fsl_write_str.fsl_word_count_u08 = 0x01;
my_fsl_write_str.fsl_destination_address_u32 = 0x00001000;

my_fsl_status = FSL_Write((fsl_write_t*) &my_fsl_write_str);
if(my_fsl_status != FSL_OK) MyErrorHandler();
```

6.2.17 FSL_GetSecurityFlags

This function reads the security (write-/erase-protection) information.

C interface for CA78K0R compiler

```
fsl_u08  FSL_GetSecurityFlags(fsl_u08 __near
                                *data_destination_pu08);
```

C interface for IAR V1.xx compiler

```
___far_func fsl_u08  FSL_GetSecurityFlags(__near fsl_u08 __near*
                                data_destination_pu08);
```

C interface for IAR V2.xx compiler

```
___far_func fsl_u08  FSL_GetSecurityFlags(fsl_u08 __near *
                                data_destination_pu08);
```

C interface for GNU compiler

```
fsl_u08  FSL_GetSecurityFlags(fsl_u08 *data_destination_pu08)
                        __attribute__ ((section ("FSL_FECD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_GetSecurityFlags(fsl_u08 __near
                                *data_destination_pu08);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_GetSecurityFlags(fsl_u08 __near
                                *data_destination_pu08)
                        __attribute__ ((section ("FSL_FECD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open

Post-condition

- None

Argument

Argument	Type	Description
data_destination_pu08	fsl_u08* (near pointer)	Pointer to the variable.

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK)	fsl_u08	Normal completion Security information is written to the variable
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*1) - Violates the precondition - FSL is suspending (*1)

(*1) For status check user mode only

Change in the destination address.

Security flag will be written in the destination address.

Meaning of each bit of security flag:

Bit 0: 1
Bit 1: Boot area overwrite protection (0: protected, 1: not protected)
Bit 2: Block erase protection (0: protected, 1: not protected)
Bit 3: 1
Bit 4: Write protection (0: protected, 1: not protected)
Bit 5: 1
Bit 6: 1
Bit 7: 1

Code example

```
/* get security informations                              */
my_status_u08 =
FSL_GetSecurityFlags( (fsl_u08*)&my_security_dest_u08);

if( my_status_u08 != 0x00 )
   my_error_handler();

if(my_security_dest_u08 & 0x0002)
{
  myPrintFkt("Boot area overwrite protection disabled!");
}
else{ myPrintFkt("Boot area overwrite protection enabled!");
```

6.2.18 FSL_GetBootFlag

This function reads the current value of the boot flag.

C interface for CA78K0R compiler

```
fsl_u08  FSL_GetBootFlag(fsl_u08 __near *data_destination_pu08);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_GetBootFlag(__near fsl_u08 __near*
                                    data_destination_pu08);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_GetBootFlag(fsl_u08 __near *
                                    data_destination_pu08);
```

C interface for GNU compiler

```
fsl_u08  FSL_GetBootFlag(fsl_u08 *data_destination_pu08)
                         __attribute__ ((section ("FSL_FECD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_GetBootFlag(fsl_u08 __near
                                *data_destination_pu08);
```

C interface for LLVM compiler

```
fsl_u08  __far  FSL_GetBootFlag(fsl_u08 __near
                                *data_destination_pu08)
                        __attribute__ ((section ("FSL_FECD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open

Post-condition

- None

Argument

Argument	Type	Description
data_destination_pu08	fsl_u08* (near pointer)	Pointer to the variable.

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK)	fsl_u08	Normal completion Security information is located in the passed variable
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*1) - Violates the precondition - FSL is suspending (*1)

(*1) For status check user mode only

Change in the destination address.
Boot flag will be written in the destination address.

00H: Boot area will be not swapped after reset
01H: Boot area will be swapped after reset

Code example

```
/* get boot-swap flag                                    */
my_status_u08= FSL_GetBootFlag((fsl_u08*)&my_bootflag_dest_u08);

if( my_status_u08 != 0x00 )
   my_error_handler();

if(my_bootflag_dest_u08){ myPrintFkt("Boot area is swapped!"); }
else{ myPrintFkt("Boot area is not swapped!"); }
```

6.2.19 FSL_GetSwapState

This function reads the current physical state of the boot clusters.

C interface for CA78K0R compiler

```
fsl_u08  FSL_GetSwapState (fsl_u08 __near
                                    *data_destination_pu08);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_GetSwapState(__near fsl_u08 __near*
                                     data_destination_pu08);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_GetSwapState(fsl_u08 __near *
                                     data_destination_pu08);
```

C interface for GNU compiler

```
fsl_u08  FSL_GetSwapState(fsl_u08 *data_destination_pu08)
                          __attribute__ ((section ("FSL_FECD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_GetSwapState(fsl_u08 __near
                                    *data_destination_pu08);
```

C interface for LLVM compiler

```
fsl_u08  __far  FSL_GetSwapState(fsl_u08 __near
                                    *data_destination_pu08)
                          __attribute__ ((section ("FSL_FECD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open

Post-condition

- None

Argument

Argument	Type	Description
data_destination_pu08	fsl_u08* (near pointer)	Pointer to the variable.

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK)	fsl_u08	Normal completion Security information is located in the passed variable
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*1) - Violates the precondition - FSL is suspending (*1)

(*1) For status check user mode only

Change in the destination address.
Boot flag will be written in the destination address.

00H: Boot cluster 0 starts with address 0x00000000
01H: Boot cluster 1 starts with address 0x00000000

Code example

```
/* get boot-swap flag                                    */
my_status_u08= FSL_GetSwapState((fsl_u08*)&my_bstate_dest_u08);

if( my_status_u08 != 0x00 )
   my_error_handler();

if(my_bstate_dest_u08){ myPrintFkt("Boot area is swapped!"); }
else{ myPrintFkt("Boot area is not swapped!"); }
```

6.2.20 FSL_GetBlockEndAddr

This function returns the end address of passed block.

Note This function may be used to secure the write function FSL_Write. If write operation exceeds the end address of a block, the written data is not guaranteed. Use this function to check whether the (write address + word number * 4) exceeds the end address of a block before calling the write function.

C interface for CA78K0R compiler

```
fsl_u08  FSL_GetBlockEndAddr(__near fsl_getblockendaddr_t*
                                        getblockendaddr_pstr);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_GetBlockEndAddr(__near
            fsl_getblockendaddr_t __near* getblockendaddr_pstr);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_GetBlockEndAddr(fsl_getblockendaddr_t
                                    __near * getblockendaddr_pstr);
```

C interface for GNU compiler

```
fsl_u08  FSL_GetBlockEndAddr(
                  fsl_getblockendaddr_t* getblockendaddr_pstr)
                        __attribute__ ((section ("FSL_FECD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_GetBlockEndAddr(__near fsl_getblockendaddr_t*
                                        getblockendaddr_pstr);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_GetBlockEndAddr(__near fsl_getblockendaddr_t*
                                        getblockendaddr_pstr)
                          __attribute__ ((section ("FSL_FECD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open

Post-condition

- None

Argument

Argument	Type	Description
getblockendaddr_pstr	fsl_getblockendaddr_t* (near pointer)	Pointer to the variable.

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK)	fsl_u08	Normal completion Address information is located in the passed variable
0x05 (FSL_ERR_PARAMETER)		Parameter error Wrong block number passed
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*1) - Violates the precondition - FSL is suspending (*1)

(*1) For status check user mode only

Change in the destination address.

Block end address will be written into the passed structure.

Code example

```
__near fsl_getblockendaddr_t      my_blk_str;

my_blk_str.fsl_block_u16 = 0x0001;
my_blk_str.fsl_destination_address_u32 = 0x00000000;
 /* get end adress of the block                          */
my_status_u08 = FSL_GetBlockEndAddr((fsl_getblockendaddr_t*)&
my_blk_str);

if( my_status_u08 != 0x00 )
   my_error_handler();

/* ##### ANALYSE my_blk_str.fsl_destination_address_u32 ##### */
```

6.2.21 FSL_GetFlashShieldWindow

This function reads the stored flash shield window. The flash shield window is a mechanism to protect the flash content against unwanted overwrite or erase defines. It can be reprogrammed by the application at any time by using the function FSL_SetFlashShieldWindow.

Example:

Flash shield window start block = 0x04

Flash shield window end block = 0x05

This configuration of the flash shield window prohibits the user to write e.g. into the block …..0x00, 0x01,0x02,0x03, 0x06…..0xFF

C interface for CA78K0R compiler

```
fsl_u08 FSL_GetFlashShieldWindow(__near fsl_fsw_t* getfsw_pstr);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_GetFlashShieldWindow(__near fsl_fsw_t
                                             __near* getfsw_pstr);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_GetFlashShieldWindow(fsl_fsw_t
                                           __near * getfsw_pstr);
```

C interface for GNU compiler

```
fsl_u08  FSL_GetFlashShieldWindow(fsl_fsw_t* getfsw_pstr)
                          __attribute__ ((section ("FSL_FECD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_GetFlashShieldWindow(__near fsl_fsw_t*
                                                getfsw_pstr);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_GetFlashShieldWindow(__near fsl_fsw_t*
                                                 getfsw_pstr)
                          __attribute__ ((section ("FSL_FECD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open

Post-condition

- None

Argument

Argument	Type	Description
getfsw_pstr	fsl_fsw_t* (near pointer)	Pointer to the variable.

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK)	fsl_u08	Normal completion Flash shield window is located in the passed variable
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*1) - Violates the precondition - FSL is suspending (*1)

(*1) For status check user mode only

Change in the destination address.

Flash Shield Window information will be written into the passed structure.

Code example

```
__near fsl_fsw_t     my_fsw_info_str;

 /* get FSW info                                          */
my_status_u08 = FSL_GetFlashShieldWindow((fsl_fsw_t*)&
                                              my_fsw_info_str);

if( my_status_u08 != 0x00 )
   my_error_handler();

/* ##### ANALYSE FSW stored in my_fsw_info_str ##### */
```

6.2.22 FSL_SetFlashShieldWindow

This function sets the new flash shield window. The flash shield window is a mechanism to protect the flash content against unwanted overwrite or erase defines.

Example:

Flash shield window start block = 0x04

Flash shield window end block = 0x05

This configuration of the flash shield window prohibits the user to write e.g. into the block …..0x00, 0x01,0x02, 0x03, 0x06…..0xFF

C interface for CA78K0R compiler

```
fsl_u08 FSL_SetFlashShieldWindow(__near fsl_fsw_t* setfsw_pstr);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_SetFlashShieldWindow(__near fsl_fsw_t
                                             __near* setfsw_pstr);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_SetFlashShieldWindow(fsl_fsw_t
                                           __near * setfsw_pstr);
```

C interface for GNU compiler

```
fsl_u08  FSL_SetFlashShieldWindow(fsl_fsw_t* setfsw_pstr)
                          __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_SetFlashShieldWindow(__near fsl_fsw_t*
                                                setfsw_pstr);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_SetFlashShieldWindow(__near fsl_fsw_t*
                                                  setfsw_pstr)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions
4. FSL_PrepareExtFunctions

Post-condition

- None

Argument

Argument	Type	Description
setfsw_pstr	fsl_fsw_t* (near pointer)	Information of the new shield window

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion New Flash Shield Window is set
0x05 (FSL_ERR_PARAMETER)		Parameter error Specified block number is outside the allowed range
0x1A (FSL_ERR_ERASE) (*1)		Erase error Block couldn't be erased.
0x1B (FSL_ERR_IVERIFY) (*1)		Verify error Data inside the flash memory is not at a sufficient voltage level
0x1C (FSL_ERR_WRITE) (*1)		Write error Data couldn't be written.
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*2) - Violates the precondition - FSL is suspending (*2)
0xFF (FSL_BUSY)(*2)		Command is running

(*1) For status check internal mode only

(*2) For status check user mode only

Code example

```
/* Example for status check internal mode only */
__near fsl_fsw_t     my_new_fsw_str;

my_new_fsw_str.fsl_start_block_u16 = 0x0000;
my_new_fsw_str.fsl_end_block_u16 = 0x0003;

/* set new fsw */
my_status_u08 = FSL_SetFlashShieldWindow((fsl_fsw_t*)
                                          &my_new_fsw_str);

if( my_status_u08 != 0x00 )
   my_error_handler();
```

6.2.23 FSL_SetXXX and FSL_InvertBootFlag

The Self-programming library has 4 functions for setting security bits. Each dedicated function sets a corresponding security flag.

These functions are listed below.

Function name	Outline
FSL_SetBlockEraseProtectFlag	Sets the block-erase-protection flag.
FSL_SetWriteProtectFlag	Sets the write-protection flag.
FSL_SetBootClusterProtectFlag	Sets the bootcluster-update-protection flag.
FSL_InvertBootFlag	Inverts the current value of the boot flag.

Caution **1. Boot-cluster protection cannot be reset by external programmer (e.g. PG-FP5).**

2. After successful execution of the FSL_InvertBootFlag function it is not allowed to execute any FSL_Setxxx function till hardware reset is occurred.

3. After RESET the other boot-cluster is activated. Please ensure a valid boot-loader inside the area, before calling the function.

4. Each security flag can be written by the application only once until next reset.

C interface for CA78K0R compiler

```
fsl_u08  FSL_SetBlockEraseProtectFlag(void);
fsl_u08  FSL_SetWriteProtectFlag(void);
fsl_u08  FSL_SetBootClusterProtectFlag(void);
fsl_u08  FSL_InvertBootFlag(void);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_SetBlockEraseProtectFlag(void);
__far_func fsl_u08  FSL_SetWriteProtectFlag(void);
__far_func fsl_u08  FSL_SetBootClusterProtectFlag(void);
__far_func fsl_u08  FSL_InvertBootFlag(void);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_SetBlockEraseProtectFlag(void);
__far_func fsl_u08  FSL_SetWriteProtectFlag(void);
__far_func fsl_u08  FSL_SetBootClusterProtectFlag(void);
__far_func fsl_u08  FSL_InvertBootFlag(void);
```

C interface for GNU compiler

```
fsl_u08  FSL_SetBlockEraseProtectFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
fsl_u08  FSL_SetWriteProtectFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
fsl_u08  FSL_SetBootClusterProtectFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
fsl_u08  FSL_InvertBootFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_SetBlockEraseProtectFlag(void);
fsl_u08 __far  FSL_SetWriteProtectFlag(void);
fsl_u08 __far  FSL_SetBootClusterProtectFlag(void);
fsl_u08 __far  FSL_InvertBootFlag(void);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_SetBlockEraseProtectFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
fsl_u08 __far  FSL_SetWriteProtectFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
fsl_u08 __far  FSL_SetBootClusterProtectFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
fsl_u08 __far  FSL_InvertBootFlag(void)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions
4. FSL_PrepareExtFunctions

Post-condition

- Security flag is set.

Argument

Argument	Type	Description
None		

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion
0x10 (FSL_ERR_PROTECTION) (FSL_InvertBootFlag only)		Protection error.
0x1A (FSL_ERR_ERASE) (*1)		Erase error Block couldn't be erased.
0x1B (FSL_ERR_IVERIFY) (*1)		Verify error Data inside the flash memory is not at a sufficient voltage level
0x1C (FSL_ERR_WRITE) (*1)		Write error Data couldn't be written.
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*2) - Violates the precondition - FSL is suspending (*2)
0xFF (FSL_BUSY)(*2)		Command is running

(*1) For status check internal mode only

(*2) For status check user mode only

Code example

```
/* Example for status check internal mode only */
my_status_u08 = FSL_SetBlockEraseProtectFlag();

if( my_status_u08 != 0x00 )
   my_error_handler();
```

6.2.24 FSL_SwapBootCluster

This function performs the physical swap of the bootclusters (0 and 1) without touching the boot flag. After the physically swap the PC (program counter) will be set regarding the reset vector from the boot cluster 1.

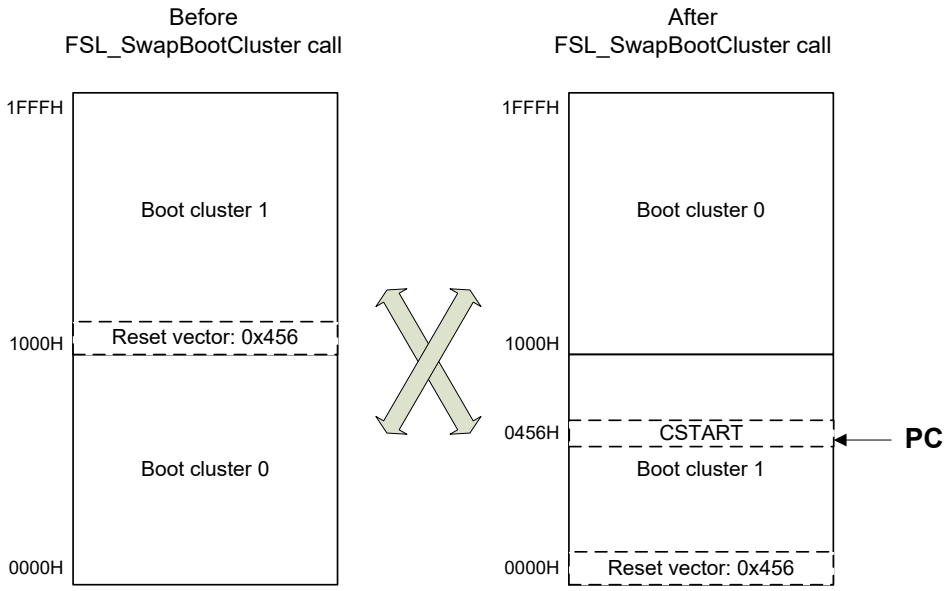

Figure 6-1 Overview of swap procedure

Note 1 **After the execution of this function boot cluster 1 is located from the address 0x0000 to 0x0FFF and PSW.IE bit is cleared! After reset the boot clusters will be switched regarding the boot swap flag.**

Note 2 **After successful execution of the FSL_InvertBootFlag function it is not allowed to execute any FSL_Setxxx function till reset is occurred.**

Note 3 In this example, a boot cluster size of 4KB is given. The actual boot cluster size is device dependent. Please refer to the device user manual of your target device for details.

C interface for CA78K0R compiler

```
fsl_u08  FSL_SwapBootCluster(void);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08  FSL_SwapBootCluster(void);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08  FSL_SwapBootCluster(void);
```

C interface for GNU compiler

```
fsl_u08  FSL_SwapBootCluster(void)
                          __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
fsl_u08  __far  FSL_SwapBootCluster(void);
```

C interface for LLVM compiler

```
fsl_u08  __far  FSL_SwapBootCluster(void)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions
4. FSL_PrepareExtFunctions

Post-condition

- Bootcluster 0 and 1 were swapped. Boot flag is not touched.

Argument

Argument	Type	Description
None		

Return types/values

Value (Definition)	Type	Description
0x10 (FSL_ERR_PROTECTION)	fsl_u08	Protection error.
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*1) - Violates the precondition - FSL is suspending (*1)

(*1) For status check user mode only

Code example

```
/* Example for status check internal mode only */
my_status_u08 = FSL_SwapBootCluster();

if( my_status_u08 != 0x00 )
   my_error_handler();
```

6.2.25 FSL_ForceReset

This function generates a software reset. For detailed information please refer to the device users manual.

C interface for CA78K0R compiler

```
void  FSL_ForceReset(void);
```

C interface for IAR V1.xx compiler

```
__far_func void  FSL_ForceReset(void);
```

C interface for IAR V2.xx compiler

```
__far_func void  FSL_ForceReset(void);
```

C interface for GNU compiler

```
void  FSL_ForceReset(void)
                          __attribute__ ((section ("FSL_RCD")));
```

C interface for CC-RL compiler

```
void  __far  FSL_ForceReset(void);
```

C interface for LLVM compiler

```
void  __far  FSL_ForceReset(void)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

None

Post-condition

None

Argument

Argument	Type	Description
None		

Return types/values

Value	Type	Description
None		

Code example

```
/* Generate a reset */
FSL_ForceReset();
```

6.2.26 FSL_GetVersionString

This function returns the pointer to the version string provided by the library.

C interface for CA78K0R compiler

```
fsl_u08 __far* FSL_GetVersionString(void);
```

C interface for IAR V1.xx compiler

```
__far_func fsl_u08 __far* FSL_GetVersionString(void);
```

C interface for IAR V2.xx compiler

```
__far_func fsl_u08 __far * FSL_GetVersionString(void);
```

C interface for GNU compiler

```
fsl_u08 __far* FSL_GetVersionString(void)
                          __attribute__ ((section ("FSL_FCD")));
```

C interface for CC-RL compiler

```
__far fsl_u08* __far  FSL_GetVersionString(void);
```

C interface for LLVM compiler

```
__far fsl_u08* __far  FSL_GetVersionString(void)
                          __attribute__ ((section ("FSL_FCD")));
```

Pre-condition

None

Post-condition

None

Argument

Argument	Type	Description
None		

Return types/values

Value	Type	Description
Pointer to the version string	fsl_u08 __far*	This is the pointer to the version string.

Description

For version control at runtime the developer can use this function to find the starting character of the library version string (ASCII format).

The version string is a zero-terminated string constant that covers library-specific information and is based on the following structure: NMMMMTTTCCCCCGVVV..V, where:

- N : library type specifier (here ‘S’ for FSL)
- MMMM : series name of microcontroller (here ‘RL78’)
- TTT : type number (here ‘T01’)
- CCCCC : compiler information (4 or 5 characters)
  - ‘Rxyy’ for CA78K0R compiler version x.yy
  - ‘Ixyy’ for IAR V1.xx compiler version x.yy
  - ‘Uxxyy’ for GNU compiler version xx.yy
  - ‘Lxyyz’ for CC-RL compiler version x.yy.0z

  Note: The version string of IAR V2.xx and LLVM indicates that the supported compiler is CC-RL because the library files for these are identical to the one for CC-RL.
- G : all memory models (here ‘G’ for general)
- VVV..V : library version
  - ‘Vxyy’ for release version x.yy
  - ‘Exyyy’ for engineering version x.yyy

Examples:

The version string of the FSL V2.20 for the CA78K0R compiler version 1.10 is: "SRL78T01R110GV220"

The version string of the FSL V2.20 for the IAR V1.xx compiler version 1.20 is: "SRL78T01I120GV220"

The version string of the FSL V2.20 for the GNU compiler version 13.01 is: "SRL78T01U1301GV220"

The version string of the FSL V2.21 for the CC-RL compiler version 1.23.04 is: "SRL78T01L1234GV221"

Code example

```
fsl_u08 __far* my_pointer_version_str;

my_pointer_version_str = FSL_GetVersionString();
```

6.2.27 FSL_SwapActiveBootCluster (CA78K0R, IAR V2.xx, CC-RL and LLVM Compiler only)

This function inverts the current value of the boot and swaps the bootcluster 0 and 1 physically.

Caution **After execution of this function the boot clusters were swapped.**

C interface for CA78K0R compiler

```
fsl_u08  FSL_SwapActiveBootCluster(void);
```

C interface for IAR V2.xx compiler

```
__far_func  fsl_u08  FSL_SwapActiveBootCluster(void);
```

C interface for CC-RL compiler

```
fsl_u08 __far  FSL_SwapActiveBootCluster(void);
```

C interface for LLVM compiler

```
fsl_u08 __far  FSL_ SwapActiveBootCluster (void)
                          __attribute__ ((section ("FSL_RCD")));
```

Pre-condition

Library must be initialized and started via following sequence:

1. FSL_Init
2. FSL_Open
3. FSL_PrepareFunctions
4. FSL_PrepareExtFunctions

Post-condition

- Boot flag is inverted
- Bootcluster were swapped

Argument

Argument	Type	Description
None		

Return types/values

Value (Definition)	Type	Description
0x00 (FSL_OK) (*1)	fsl_u08	Normal completion Boot flag is inverted and bootcluster were swapped
0x10 (FSL_ERR_PROTECTION)		Protection error.
0x1A (FSL_ERR_ERASE) (*1)		Erase error Block couldn't be erased.
0x1B (FSL_ERR_IVERIFY) (*1)		Verify error Data inside the flash memory is not at a sufficient voltage level
0x1C (FSL_ERR_WRITE) (*1)		Write error Data couldn't be written.
0x1F (FSL_ERR_FLOW)		Possible errors: - Last operation has not finished (*2) - Violates the precondition - FSL is suspending (*2)
0xFF (FSL_BUSY)(*2)		Command is running

(*1) For status check internal mode only

(*2) For status check user mode only

Code example

```
my_status_u08 = FSL_SwapActiveBootCluster();

if( my_status_u08 != 0x00 )
   my_error_handler();
```

Chapter 7 Operation

This chapter describes the operation of the libary as well as the integration.

7.1 Basic workflow

To be able to use the FSL (execute commands) in a proper way the user has to follow a specific procedure. Following sub-chapters will help you to understand the different workflows.

7.1.1 FSL_Write, FSL_Erase ... in status check internal mode

The status check internal mode can be executed from ROM or RAM. The following figure illustrates the flash access flow.

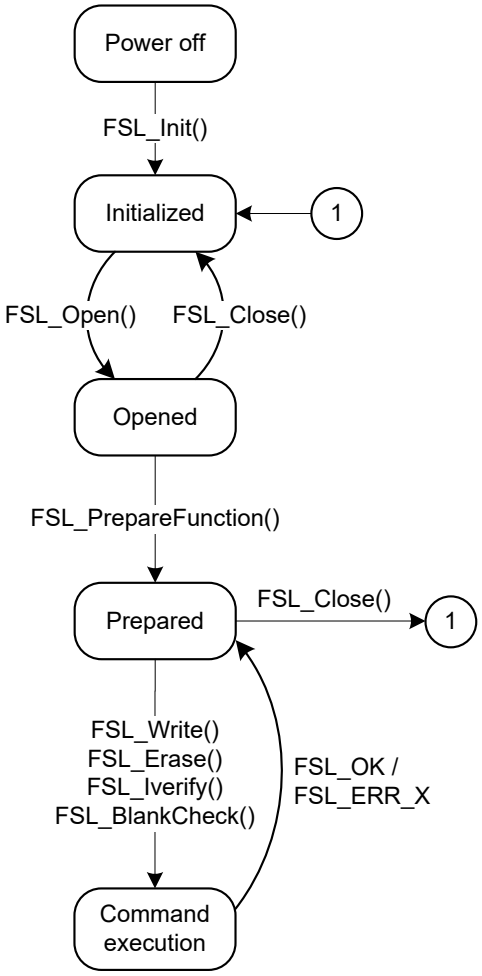

Figure 7-1 Status check internal mode (flash access)

7.1.2 FSL_Write, FSL_Erase ... in status check user mode

As shown in the figure below on status check mode the user has to copy the FSL parts into RAM before using them for status check.

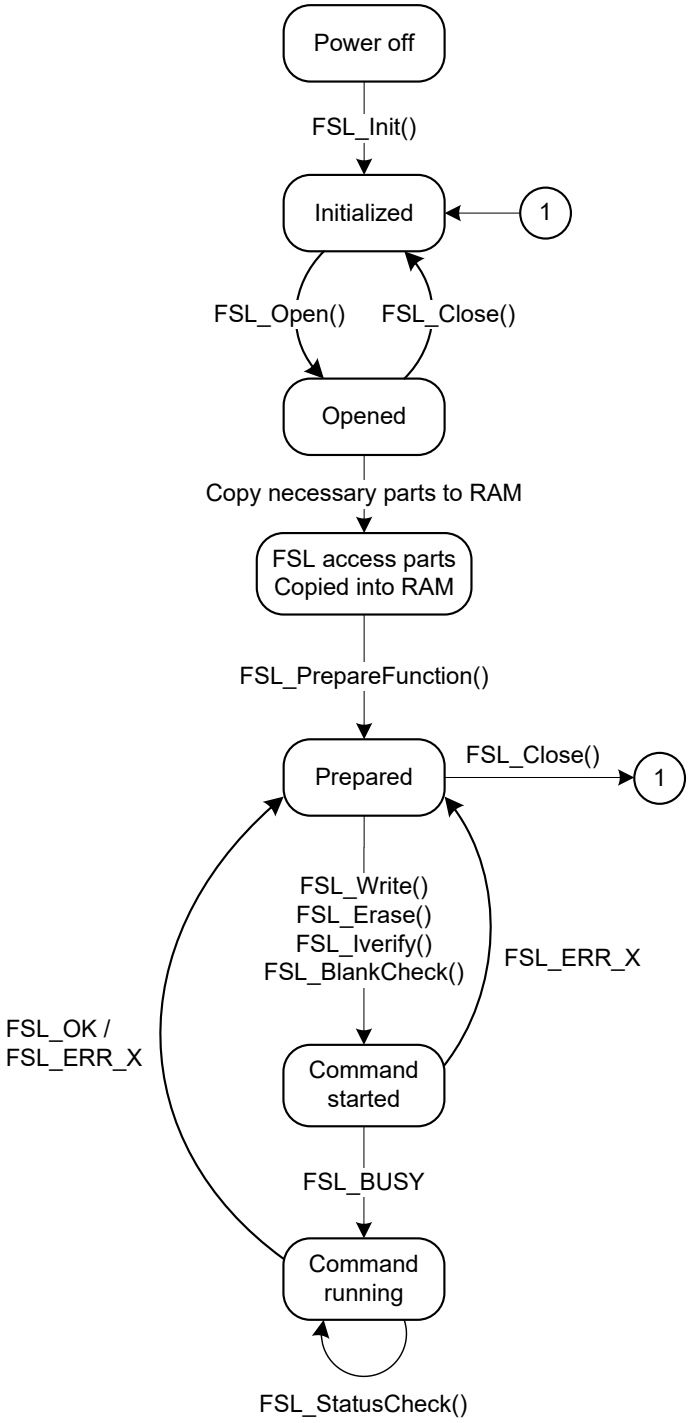

Figure 7-2 Status check user mode (flash access)

7.1.3 FSL_SetXXX, FSL_InvertBootFlag ... in status check user mode

The following figure illustrates the flow where the FSL_SetXXX functions are used. It is basically the same flow except that the FSL_PrepareExtFunction function must be called.

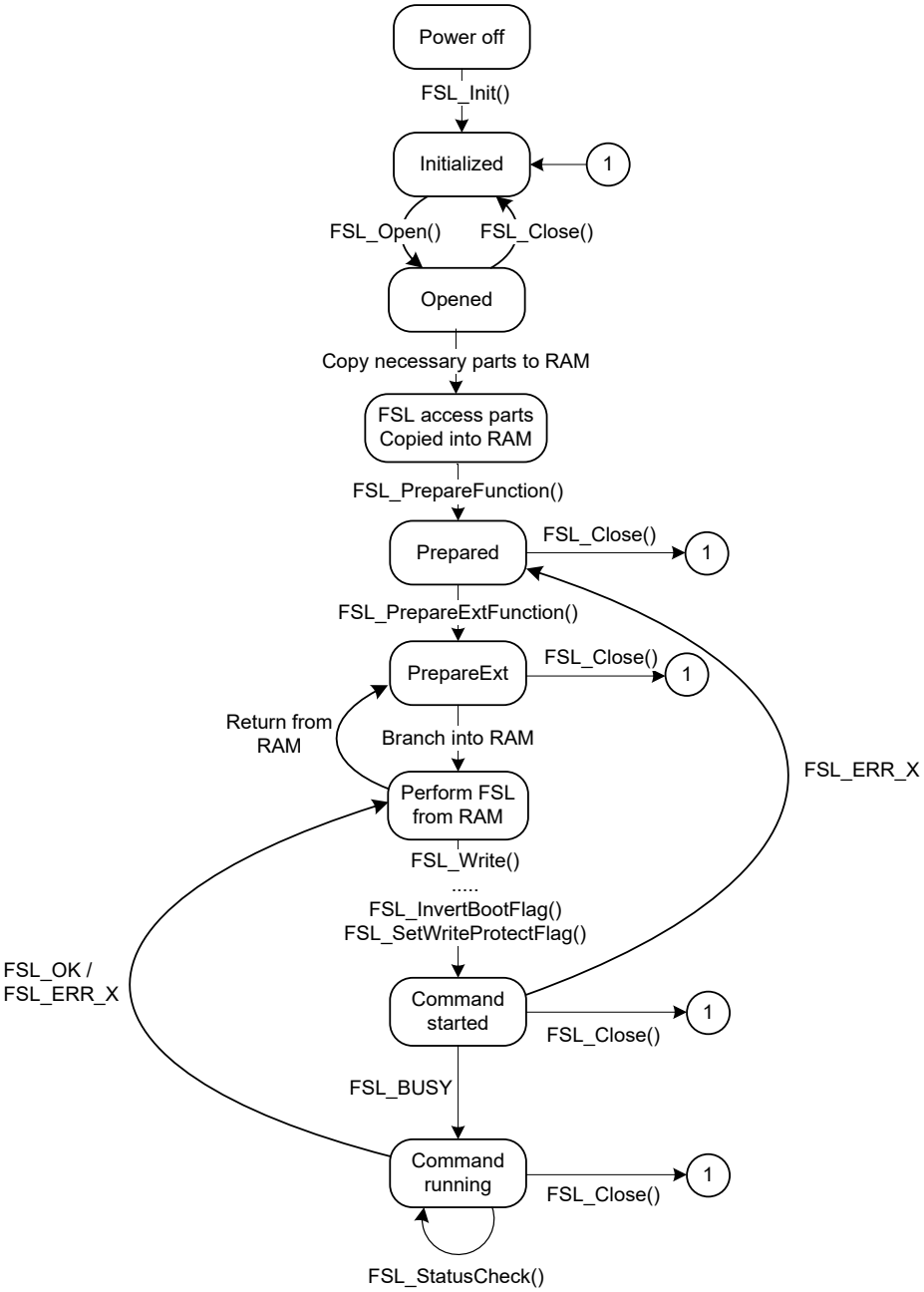

Figure 7-3 Status check user mode (flash access + security setting)

7.1.4 Interrupts during Self-programming

As described in the chapter “Interrupts servicing” the interrupts will always be handled in RAM during flash access. The following flow shows the general procedure on how to use interrupts.

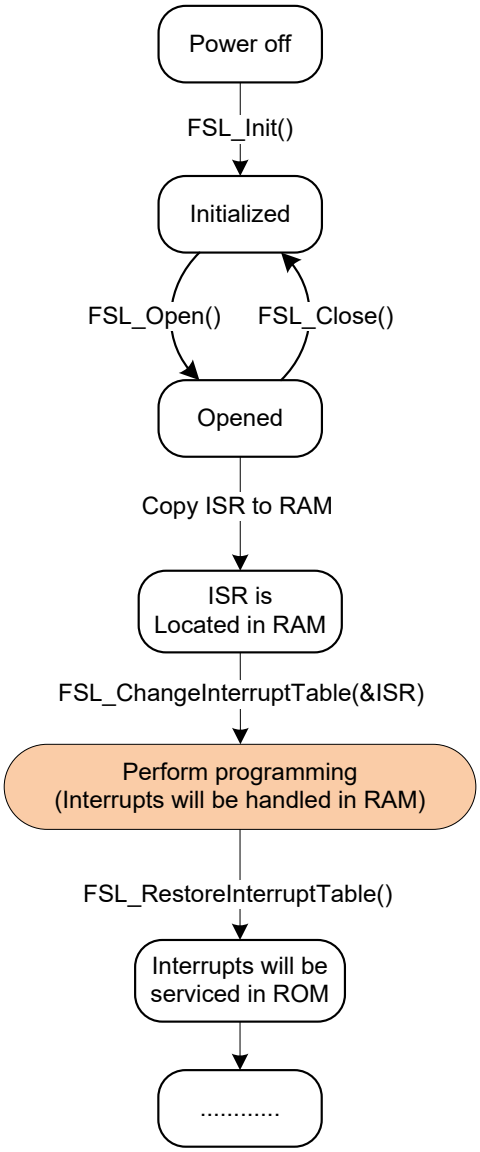

Figure 7-4 Interrupts during Self-programming

7.2 Integration

- Copy all the files into your project subdirectory
- add all fsl*.* files into your project (workbench or make-file)
- adapt the linker file due to your requirements (at least segments FSL_FCD, FSL_FECD, FSL_BCD, FSL_BECD, FSL_RCD and FSL_RCD_ROM(IAR V1.xx only) should be defined). For detailed segment description please refer to chapter 2.1.2.
- re-compile the project

Chapter 8 Characteristics

This chapter includes the timing information's of each function depending on the user configuration. The following timings are based on the library version V2.20.

8.1 Function response time

8.1.1 Status check user mode

Please refer to the following figure for the function response time definition.

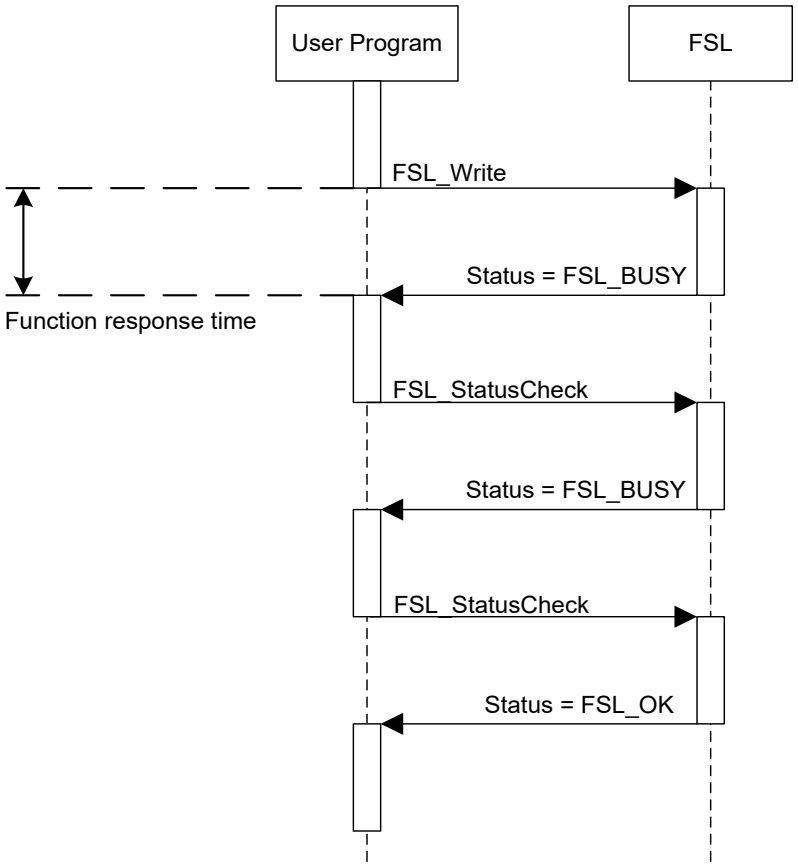

Figure 8-1 Function response time definition in case of SCU

Table 8-1 Full speed mode

Function	Max
FSL_Init	5021/fclk
FSL_PrepareFunctions	2484/fclk
FSL_PrepareExtFunctions	1259/fclk
FSL_ChangeInterruptTable	253/fclk
FSL_RestoreInterruptTable	229/fclk
FSL_Open	10/fclk
FSL_Close	10/fclk
FSL_BlankCheck	2069/fclk + 30 us
FSL_Erase	2192/fclk + 30 us
FSL_IVerify	2097/fclk + 30 us
FSL_Write	2451/fclk + 30 us
FSL_GetSecurityFlags	331/fclk + 0 us
FSL_GetBootFlag	328/fclk + 0 us
FSL_GetSwapState	206/fclk + 0 us
FSL_GetBlockEndAddr	368/fclk + 0 us
FSL_GetFlashShieldWindow	307/fclk + 0 us
FSL_SetBlockEraseProtectFlag	2347/fclk + 30 us
FSL_SetWriteProtectFlag	2346/fclk + 30 us
FSL_SetBootClusterProtectFlag	2347/fclk + 30 us
FSL_InvertBootFlag	2341/fclk + 30 us
FSL_SetFlashShieldWindow	2141/fclk + 30 us
FSL_SwapBootCluster	419/fclk + 32 us
FSL_SwapActiveBootCluster	2316/fclk + 30 us
FSL_ForceReset	-
FSL_StatusCheck	1135/fclk + 50 us
FSL_StandBy	
(case: Erase)	935/fclk + 31 us
(case: except Erase) FSL_SetXXX are supported	140367/fclk + 513844 us
(case: except Erase) FSL_SetXXX are not supported	76101/fclk + 35952 us
FSL_WakeUp	
(case: suspended erase)	2144/fclk + 30 us
(case: except erase)	148/fclk + 0 us
FSL_GetVersionString	10/fclk

Remark fclk: CPU operating frequency
(For example, when using a 20 MHz clock, fclk is 20.)

Table 8-2 Wide voltage mode

Function	Max
FSL_Init	5021/fclk
FSL_PrepareFunctions	2484/fclk
FSL_PrepareExtFunctions	1259/fclk
FSL_ChangeInterruptTable	253/fclk
FSL_RestoreInterruptTable	229/fclk
FSL_Open	10/fclk
FSL_Close	10/fclk
FSL_BlankCheck	2068/fclk + 30 us
FSL_Erase	2192/fclk + 30 us
FSL_IVerify	2097/fclk + 30 us
FSL_Write	2451/fclk + 30 us
FSL_GetSecurityFlags	331/fclk + 0 us
FSL_GetBootFlag	328/fclk + 0 us
FSL_GetSwapState	206/fclk + 0 us
FSL_GetBlockEndAddr	368/fclk + 0 us
FSL_GetFlashShieldWindow	307/fclk + 0 us
FSL_SetBlockEraseProtectFlag	2347/fclk + 30 us
FSL_SetWriteProtectFlag	2346/fclk + 30 us
FSL_SetBootClusterProtectFlag	2347/fclk + 30 us
FSL_InvertBootFlag	2341/fclk + 30 us
FSL_SetFlashShieldWindow	2141/fclk + 30 us
FSL_SwapBootCluster	419/fclk + 32 us
FSL_SwapActiveBootCluster	2316/fclk + 30 us
FSL_ForceReset	-
FSL_StatusCheck	1135/fclk + 50 us
FSL_StandBy	
(case: Erase)	935/fclk + 44 us
(case: except Erase) FSL_SetXXX are supported	123274/fclk + 538046 us
(case: except Erase) FSL_SetXXX are not supported	73221/fclk + 69488 us
FSL_WakeUp	
(case: suspended erase)	2144/fclk + 30 us
(case: except erase)	148/fclk + 0 us
FSL_GetVersionString	10/fclk

Remark fclk: CPU operating frequency
(For example, when using a 20 MHz clock, fclk is 20.)

8.1.2 Status check internal mode

Please refer to the following figure for the function response time definition in case of SCI.

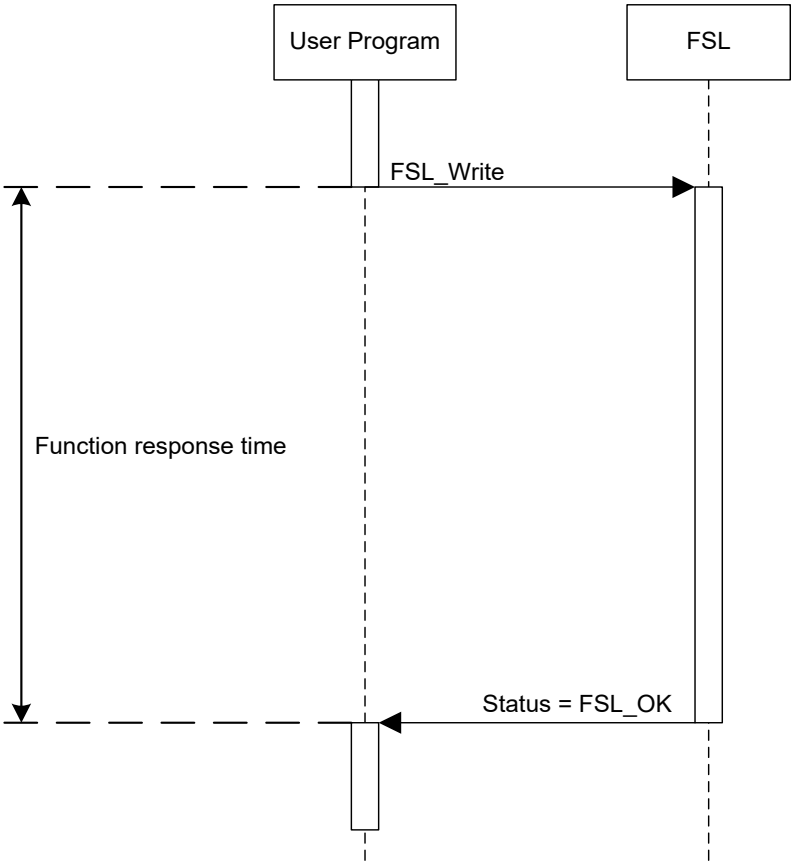

Figure 8-2 Function response time definition in case of SCI

Table 8-3 Full speed mode

Function	Min	Max
FSL_Init	3348/fclk	5021/fclk
FSL_PrepareFunctions	1656/fclk	2484/fclk
FSL_PrepareExtFunctions	839/fclk	1259/fclk
FSL_ChangeInterruptTable	168/fclk	253/fclk
FSL_RestoreInterruptTable	152/fclk	229/fclk
FSL_Open	6/fclk	10/fclk
FSL_Close	6/fclk	10/fclk
FSL_BlankCheck	2201/fclk + 32 us	4833/fclk + 164us
FSL_Erase	3251/fclk + 108 us	73339/fclk + 255366 us
FSL_IVerify	6982/fclk + 738 us	10474/fclk + 1107 us
FSL_Write	2080/fclk + 44 us +(396/fclk+40us)*W	3121/fclk + 66 us +(1153/fclk + 561us)*W
FSL_GetSecurityFlags	220/fclk + 0 us	331/fclk + 0 us
FSL_GetBootFlag	218/fclk + 0 us	328/fclk + 0 us
FSL_GetSwapState	137/fclk + 0 us	206/fclk + 0 us
FSL_GetBlockEndAddr	245/fclk + 0 us	368/fclk + 0 us
FSL_GetFlashShieldWindow	204/fclk + 0 us	307/fclk + 0 us
FSL_SetBlockEraseProtectFlag	1047/fclk+ 12us	140946/fclk+ 513830us
FSL_SetWriteProtectFlag	1046/fclk+ 12us	140945/fclk+ 513830us
FSL_SetBootClusterProtectFlag	1047/fclk+ 12us	140946/fclk+ 513830us
FSL_InvertBootFlag	1043/fclk+ 12us	140940/fclk+ 513830us
FSL_SetFlashShieldWindow	904/fclk+ 12us	140739/fclk+ 513830us
FSL_SwapBootCluster	279/fclk+ 21us	419/fclk+ 32us
FSL_SwapActiveBootCluster	1292/fclk+ 33us	141314/fclk+ 513862us
FSL_ForceReset	n/a	n/a
FSL_StatusCheck	n/a	n/a
FSL_StandBy	n/a	n/a
FSL_WakeUp	n/a	n/a
FSL_GetVersionString	6/fclk	10/fclk

Remarks 1. fclk: CPU operating frequency
(For example, when using a 20 MHz clock, fclk is 20.)
2. W: The number of words to be written (1 word = 4 bytes)
(For example, when specifying 2 words = 8 bytes, W is 2.)

Table 8-4 Wide voltage mode

Function	Min	Max
FSL_Init	3347/fclk	5021/fclk
FSL_PrepareFunctions	1656/fclk	2484/fclk
FSL_PrepareExtFunctions	839/fclk	1259/fclk
FSL_ChangeInterruptTable	168/fclk	253/fclk
FSL_RestoreInterruptTable	152/fclk	229/fclk
FSL_Open	6/fclk	10/fclk
FSL_Close	6/fclk	10/fclk
FSL_BlankCheck	2198/fclk + 82us	4574/fclk + 401us
FSL_Erase	3116/fclk + 267us	64468/fclk + 266193 us
FSL_IVerify	5106/fclk + 5022us	7659/fclk + 7534us
FSL_Write	2080/fclk + 44 us +(394/fclk+74us)*W	3121/fclk + 66us +(1108/fclk+1085us)*W
FSL_GetSecurityFlags	220/fclk + 0 us	331/fclk + 0 us
FSL_GetBootFlag	218/fclk + 0 us	328/fclk + 0 us
FSL_GetSwapState	137/fclk + 0 us	206/fclk + 0 us
FSL_GetBlockEndAddr	245/fclk + 0 us	368/fclk + 0 us
FSL_GetFlashShieldWindow	204/fclk + 0 us	307/fclk + 0 us
FSL_SetBlockEraseProtectFlag	1047/fclk+ 12us	123853/fclk+ 538032us
FSL_SetWriteProtectFlag	1046/fclk+ 12us	123852/fclk+ 538032us
FSL_SetBootClusterProtectFlag	1047/fclk+ 12us	123853/fclk+ 538032us
FSL_InvertBootFlag	1043/fclk+ 12us	123847/fclk+ 538032us
FSL_SetFlashShieldWindow	904/fclk+ 12us	123646/fclk+ 538032us
FSL_SwapBootCluster	279/fclk+ 21us	419/fclk+ 32us
FSL_SwapActiveBootCluster	1292/fclk+ 33us	124221/fclk+ 538064us
FSL_ForceReset	n/a	n/a
FSL_StatusCheck	n/a	n/a
FSL_StandBy	n/a	n/a
FSL_WakeUp	n/a	n/a
FSL_GetVersionString	6/fclk	10/fclk

Remarks 1. fclk: CPU operating frequency
(For example, when using a 20 MHz clock, fclk is 20.)
2. W: The number of words to be written (1 word = 4 bytes)
(For example, when specifying 2 words = 8 bytes, W is 2.)

Chapter 9 General cautions

The following cautions must be considered before developing an application using the RL78 FSL T01:

- Library code and constants must be located completely in the same 64k flash page.
- The library segments FSL_BCD and FSL_BECD must not be located at the end of any boundary of a 64KB flash page (at least 2 Bytes of space are needed at the end of a 64KB flash page) if the used library version is V2.20 or earlier.
- The library segments FSL_BCD and FSL_BECD must not be located at an odd addresses.
- All functions are not re-entrant. That means don't call FSL functions inside the ISRs while any FSL function is already running.
- Task switches, context changes and synchronization between FSL functions:

  All FSL functions depend on FSL global available information and are able to modify this. In order to avoid synchronization problems, it is necessary that at any time only one FSL function is executed. So, it is not allowed to start an FSL function, then switch to another task context and execute another FSL function while the last one has not finished.
- It is not possible to modify the Data Flash parallel to modification of the Code Flash. This also means that it is not allowed to use the FSL in parallel to any data flash or EEPROM emulation library. If these libraries need to be used, please ensure to end the FSL operation via FSL_Close first.
- Do not enter the STOP or HALT mode during self-programming. If you are going to enter the STOP or HALT mode, the flash self-programming needs to be halted\stopped via the FSL_StandBy or FSL_Close function. In case of FSL_Close, please finish the running command before calling this function.
- The code flash cannot be read during FSL operations accessing the code flash.
- Some RAM areas are prohibited during execution of FSL library. Please refer to the device user manual for detailed information.
- It is not allowed to locate the data buffer over the 0xFFE20 address.
- When using the data transfer controller (DTC) during the execution of flash self-programming, do not allocate the RAM area used by the DTC to the prohibited RAM area or an address over 0xFFE20.
- Initialize all arguments to be used by the FSL functions. When corresponding RAM area is not initialized, a RAM parity error is detected and the RL78 microcontroller might be reset. For a more detailed description of the RAM parity error feature, please refer to the user manual of the target RL78 microcontroller.
- The watchdog timer must be configured for period which is longer than the execution time of FSL_SetXXX and FSL_SwapBootCluster in case of using the status check internal mode.
- FSL_RCD segment size must be at least 10 bytes larger than the size of FSL_RCD_ROM segment when using FSL_CopySection.

- Internal high-speed oscillator must be started before using of the FSL.
- During the execution of the FSL_ChangeInterruptTable and the FSL_RestoreInterruptTable function, Interrupts shall be disabled.
- When a reset is done after the interrupt destination is changed with the FSL_ChangeInterruptTable function, the system starts up with the recovered original interrupt destination.
- In case of usage of RAM ISR:
  1. It is not allowed to locate the RAM ISR over the 0xFFE20 address.
  2. It is not allowed to access flash (read/write) during execution of RAM ISR.
  3. User has to take care that the request flag is cleared before leaving the ISR.
  4. Interrupt processing on the RAM increases by up to 20 clocks compared to the normal interrupt response time.
- Not all RL78 microcontrollers support an interrupt during the execution of flash self-programming. Please refer to the user manual of the target RL78 microcontroller for device-specific restrictions during self-programming.
- Not all RL78 microcontrollers support the boot-swap function. Please refer to the user manual of the target RL78 microcontroller to see whether the boot swap function is supported.
- Not all RL78 microcontrollers support the security-setting function by the FSL. Please refer to the user manual of the target RL78 microcontroller for device-specific security settings and features.
- Usage of FSL_ChangeInterruptTable for GNU compiler:

  The GNU compiler may utilize a so-called plt section storing a branch table in the Code Flash when obtaining the function entry addresses by means of the address operator '&'. This can lead to unpredictable behaviour in case the Code Flash is not available during FSL operation, but a previously stored function pointer is used to call the function. This needs to be specifically considered when using FSL_ChangeInterruptTable as it requires to provide a function pointer to the RAM as argument. In order to obtain the actual function address in the RAM for operation, using the GNU compiler, please consider the following workaround:

  1. Have the function definition in a different source file than the routines that need the function entry address.
  2. The public declaration of the function shall be an array of fsl_u08 rather than a function (e.g. extern fsl_u08 __far my_func[];). The name of that array must be same as the name of the function.

  If both of the conditions above are fulfilled, than the label of the function is visible to the compiler as an array. For an array, the compiler will not use the plt table. Thereby, it should be safe to obtain a direct pointer to the function address in RAM by means of the address operator '&'.

- Usage of FSL_ChangeInterruptTable for LLVM compiler:

  Using the LLVM compiler, please consider avoiding that the plt entry for the interrupt function is generated.

  The interrupt function shall be specified with memory allocation area "__far". For a function explicitly located far area with "__far" keyword (e.g. void __far my_func(void)), the compiler will not generate the plt entry for that function. Thereby, a direct pointer to the function address in RAM can be obtained by means of the address operator '&'.

- When using an assembler of the CC-RL compiler from Renesas Electronics, the hexadecimal prefix representation (0x..) cannot be mixed together with the suffix representation (..H). Specify the representation method by editing the symbol definition in fsl.inc to match the user environment.

  fsl.inc

  ```
  ; __FSL_INC_BASE_NUMBER_SUFFIX .SET 1
  ```

  When symbol "FSL_INC_BASE_NUMBER_SUFFIX" is not defined (initial state), the prefix representation will be selected.

  fsl.inc

  ```
  __FSL_INC_BASE_NUMBER_SUFFIX .SET 1
  ```

  When symbol "FSL_INC_BASE_NUMBER_SUFFIX" is defined, the suffix representation will be selected.

- Additional cautions on using the FSL for IAR V2.xx and LLVM
  - The version string provided by the flash library includes the information on the supported compiler.The string indicates that the supported compiler is CC-RL because the library for IAR V2.xx and LLVM are essentially identical to the one for CC-RL.

Revision history

The following table shows the differences between the current and the previous document version.

Chapter	Page	Description
		Rev .1.02:
8.1	72-77	Timings updated.
		Rev. 1.03:
all	all	GNU support added
2.1.2	10-11	Tables for segment location updated
5, 4.2.24	15-18, 65	Adding note for device-dependent boot cluster sizes
6.2.8	32	Adding additional explanations to pre- and post-condition of FSL_ChangeInterruptTable
6.2.9	34	Extending pre-condition of FSL_RestoreInterruptTable
8.1	76-81	Timings updated for library V2.20
9	82	List of cautions extended
		Rev. 1.04:
all	all	CC-RL support added
all	all	Renesas (REN) Compiler renamed to CA78K0R
		Rev. 1.05:
6.2.8	36	Updated GNU code example
9	89	New caution added related to usage of FSL_ChangeInterruptTable for GNU compiler
		New cautions added related to FSL_BCD and FSL_BECD sections mapping
		Rev. 1.10
all	all	IAR V2.xx compiler support added
9	88	New caution added related to usage of assembler for CC-RL compiler
		Rev.1.20
all	all	LLVM compiler support added
9	92	New caution added related to usage of FSL_ChangeInterruptTable for LLVM compiler

Flash Self-programming Library