User’s Manual

Flash Self-programming Library

FSL - T05

Flash Self-programming Library for V850 Single Voltage Flash Devices

Document No. U20281EE1V0UM00
Date Published: 10.02.2010

Legal Notes

The information in this document is current as of March 2010. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)

(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.

(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

Table of Contents

Chapter 1 Introduction

This user's manual describes the internal structure, the functionality and software interfaces (API) of the NEC V850 Flash Self-Programming Library (FSL) type T05, designed for V850 Flash devices based on the UX6LF Flash technology

The device features differ depending on the used Flash implementation and basic technology node. Therefore, pre-compile and run-time configuration options allow adaptation of the library to the device features and to the application needs.

The libraries are delivered in source code. However it has to be considered carefully to do any changes, as not intended behavior and programming faults might be the result.

The development environments of the companies Green Hills (GHS), IAR and NEC are supported. Due to the different compiler and assembler features, especially the assembler files differ between the environments. So, the library and application programs are distributed using an installer tool allowing selecting the appropriate environment.

For support of other development environments, additional development effort may be necessary. Especially, but maybe not only, the calling conventions to the assembler code and compiler dependent section defines differ significantly.

The libraries are delivered together with device dependent application programs, showing the implementation of the libraries and the usage of the library functions.

The different options of setup and usage of the libraries are explained in detail in this document.

Caution:

Please read all chapters of the application note carefully.
Much attention has been put to proper conditions and limitations description. Anyhow, it can never be ensured completely that all not allowed concepts of library implementation into the user application are explicitly forbidden. So, please follow exactly the given sequences and recommendations in this document in order to make full use of the libraries functionality and features and in order to avoid any possible problems caused by libraries misuse.

The Flash Self-Programming Libraries together with application samples, this application note and other device dependent information can be downloaded from the following URL:

http://www.eu.necel.com/updates

1.1 Naming Conventions

Certain terms, required for the description of the Flash Access 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.

These abbreviations shall be explained here:

Abbreviations / Acronyms	Description
Block	Smallest erasable unit of a flash macro
Code Flash	Embedded Flash where the application code is stored
Data Flash	Embedded Flash where mainly the data of the EEPROM emulation are stored. Beside that also code operation might be possible depending on the implementation.
Dual Operation	Dual operation is the capability to fetch code during reprogramming of the flash memory. Current limitation is that dual operation is only available between different flash macros. Within the same flash macro it is not possible! Please refer to the device user manual for Dual Operation support.
Extra Area	A separate area of the Code Flash where protection flags and other internal information are stored. The area is not directly accessible by the application. Only the device internal firmware has access to that area.
FMS	Flash Macro Service
FSL	Flash Self-Programming Library
FSS	Flash Self-Programming System
FSW	Flash Shield Window
FW	Firmware
Flash	Electrical erasable nonvolatile memory. As distinguished from ROM, this type of memory can be re-programmed several times.
Flash Block	A flash block is the smallest erasable unit of the flash memory.
Flash Macro	A flash comprises of the cell array, the sense amplifier and the charge pump (CP). For address decoding and access some additional logic is needed. A certain number of Flash blocks is grouped together in a Flash macro.
RAM	“Random access memory” - volatile memory with random access
ROM	“Read only memory” - nonvolatile memory. The content of that memory can not be changed.
Serial programming	The serial programming mode is used to program the device with an external programmer tool.
Single Voltage	For the reprogramming of single voltage Flashes the voltage needed for erasing and programming are generated onboard of the microcontroller. No external voltage needed like for dual- voltage flash types.

1.2 Flash versus EEPROM

Major difference between Flash and EEPROM (or E^2PROM) is the reprogramming granularity. EEPROM can be reprogrammed wordwise, where the size of one word depends on the organization and interface. It can vary in the wide range between 8 bit and 256 bytes.

Depending on the implementation, Flash may also be programmed wordwise, but the Erase can only be done on a complete block. This is the major limitation of Flash against EEPROM, but due to that the memory hardware effort can be reduced significantly, making the embedded non volatile memory for program code affordable.

1.3 Dual Flash operation

Common for all Flash implementations is, that during Flash modification operations (Erase/Write) a certain amount of Flash memory is not accessible for any read operation (e.g. program execution or data read).

This does not only concern the modified Flash range, but a certain part of the complete Flash system. The amount of not accessible Flash depends on the device architecture.

A standard architectural approach is the separation of the Flash into Code Flash and Data Flash. By that, it is possible to read from the Code Flash (to execute program code or read data) while Data Flash is modified, and vice versa.

If not mentioned otherwise in the device users manuals, UX6LF devices with Data Flash are designed according to this standard approach.

Note:

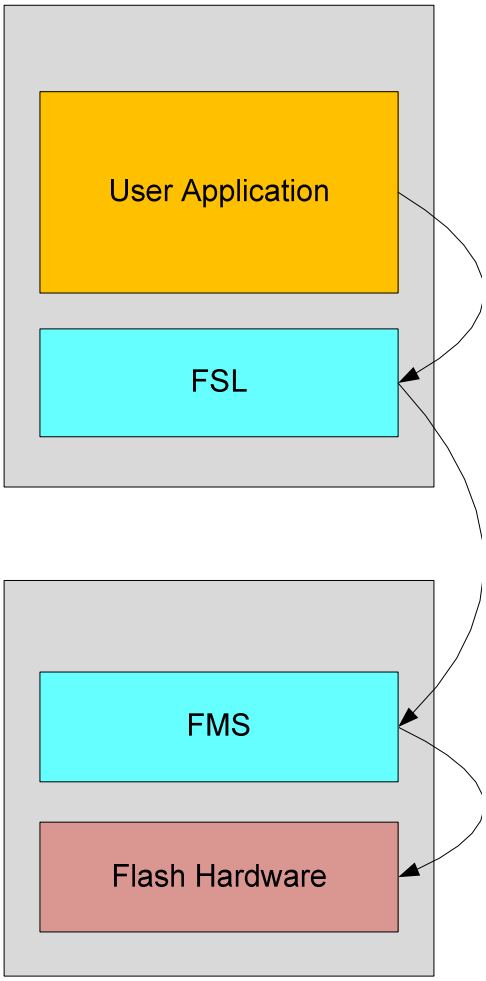
It is not possible to modify Code Flash and Data Flash in parallel

Chapter 2 FSL Architecture

This chapter describes the function of all blocks belonging to the Flash Self-Programming System.

Even though this manual describes the functional block FSL, a short description of all concerned functional blocks and their relationship can be beneficial for the general understanding.

Figure 1 Rough relationship between functional system blocks of the FSS



Application

The functional block “Application” is the user application (including a potential bootloader) provided by the customer.

Flash Self-Programming Library (FSL)

The functional block “Flash Self-Programming Library” offers all functions and commands necessary to reprogram the application in a user friendly C language interface.

Flash Macro Service (FMS)

The “Flash Macro Service” provides the functionality to control the Flash programming hardware (Sequencer). The FMS is part of the device internal firmware.

Chapter 3 FSL Implementation

3.1 File structure

The library is delivered as a complete compilable sample project which contains the FSL and in addition an application sample to show the library implementation and usage in the target application.

The application sample initializes the *FSL* and does some dummy data set *Erase Write* and *Verify* operations.

Differing from former Self-Programming Libraries, this one is realized not as a IDE related specific sample project, but as a standard sample project which is controlled by makefiles.

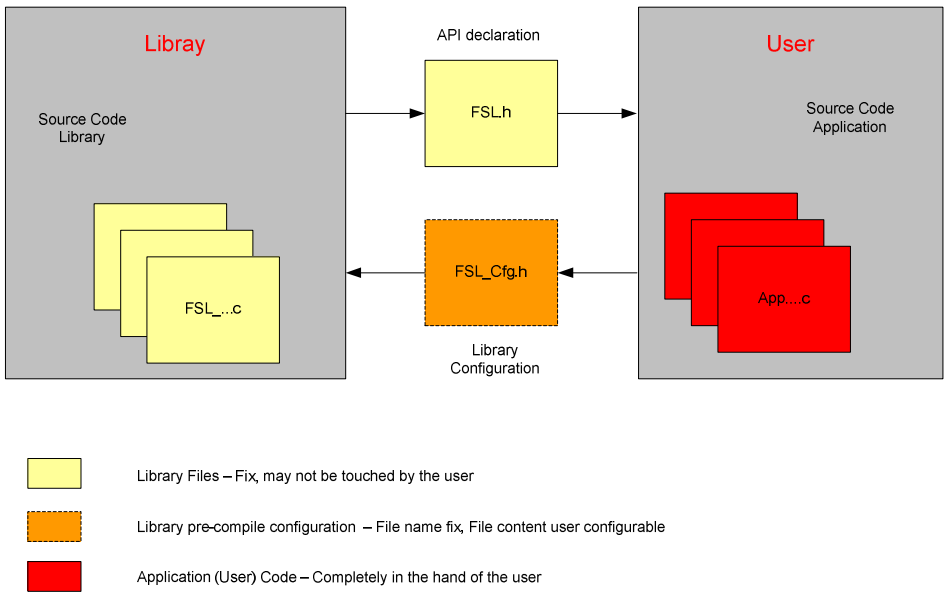
Following that, the sample project can be built in a command line interface and the resulting elf file can be run in the debugger.

The delivery package contains dedicated directories for the library containing the source and the header files.

3.1.1 Overview

The following picture contains the library and application related files:

Figure 2 Library and application file structure



The library code consists of different source files, starting with FSL_... The files may not be touched by the user, independently, if the library is distributed as source code or pre-compiled.

The file FSL.h is the library interface functions header file. It also includes library interface parameters and types.

In case of source code delivery, the library must be configured for compilation. The file FSL_Cfg.h contains defines for that. As it is included by the library source files, the file contents may be modified by the user, but the file name may not.

Note:
Wrong configuration of the FSL might lead to undefined results.

FSL_User.c and FSL_User.h do not belong to the libraries themselves, but to the user application. These files reflect an example, how to activate the Flash environment and handle the FLMD0 pin.

If overtaking the files FSL_User.c/h into the user application, only the file FSL_User.c need to be adapted by the user, while FSL_User.h may remain unchanged.

3.1.2 Delivery package directory structure and files

[root]	
Release.txt	Installer package release notes
[root]\[make]	
GNUPublicLicense.txt	Make utility license file
libiconv2.dll	DLL-File required by make.exe
libintl3.dll	DLL-File required by make.exe
make.exe	Make utility
[root]\[<device name>]\[compiler]	
Build.bat	Batch file to build the application sample
Clean.bat	Batch file to clean the application sample
Makefile	Makefile that controls the build and clean process
[root]\ [<device name>]\[<compiler>]\[sample]	
Main.c	Main source code
target.h	target device and application related definitions
... device header files ...	(GHS: df<device number>.h, io_macros.h, ...
	IAR: io_70f3xxx.h
	NEC: -)
... startup file ...	(GHS: Startup_df<dev. num.>.850
	IAR: DF3xxx_HWInit.s85
	NEC: tbd)
... linker directive file ...	(GHS: Df<device number>.ld
	IAR: lnk70f3xxx.xcl
	NEC: tbd)
[root]\[<device name>]\[<compiler>]\[sample]\[FSL]	
FSL_Cfg.h	Header file with definitions for library setup at compile time
FSL.h	Header file containing function prototypes, error and status codes

FSL_User.h	User file header including Flash environment activation / deactivation and FLMD0 handling. To be edited by the user.
FSL_User.c	User file including Flash environment activation / deactivation and FLMD0 handling. Maybe modified by the user.

[root]\[<device name>]\[<compiler>]\[sample]\[FSL]\[lib]

FSL_Global.h	Library internal defines, function prototypes and variables
FSL_UserIF_Init.c	Source code for the *FSL* initialization
FSL_UserIF_Operation.c	Source code for the normal *FSL* operations
FSL_FirmwareIF.c	Interface to the firmware
FSL_BasicFct.c	Source code of basic functions used during self-programming
FSL_BasicFct_Asm.s	Assembler code of basic functions used during self-programming

3.2 FSL Linker sections

The following sections are Flash Self-Programming Library related:

FSL data sections

- FSL_DATA
  This section contains the variables required for FSL. It can be located either in internal or in external RAM.

FSL code sections

- FSL_CODE_ROM
  This section contains the code executed at the beginning of self-programming. This code is executed at the original location, e.g. internal Flash. The library initialization is part of this section.
- FSL_CODE_ROMRAM
  The section contains the user interface. Depending on the library configuration, code from this section need to be executed in RAM or may be executed in Flash.
- FSL_CODE_RAM
  This section contains the firmware interface and so need to be executed outside the reprogrammed Flash area.
- FSL_CODE_RAM_USRINT
  This section contains the interrupt functions.
- FSL_CODE_RAM_USR
  This section contains user functions to be executed in RAM. User functions may contain code for the self-programming control flow, e.g.
- FSL_CODE_RAM_EX_PROT
  Dummy section to avoid prefetch errors at the borders of the copied sections during RAM execution.

Note:

It is not allowed to place any section in between the FSL code sections. A violation of that rule or a reorder of the sections will cause a crash of the library.

3.3 MISRA Compliance

The FSL has been tested regarding MISRA compliance.

The used tool is the QAC Source Code Analyzer which tests against the MISRA 2004 standard rules.

All MISRA related rules have been enabled. Findings are commented in the code while the QAC checker machine is set to silent mode in the concerning code lines.

Chapter 4 FSL Usage

4.1 Flash Security

4.1.1 Strategy

In most cases a application software contains important intellectual property and/or data, that may not be distributed to others or manipulated by others. In order to ensure Flash data integrity and to prevent unintended data read-out, NEC implements a set of features and mechanisms into Flash devices.

As these mechanisms may also limit the flexibility required for the application and the programming or reprogramming, it has to be decided carefully what level of protection is intended.

Two major items to be considered in the protection concept are:

- Illegal read-out of Flash contents
- Illegal reprogramming of the Flash

In the following the strategies regarding these items are described in detail:

Illegal read-out of Flash contents

Read-out, legal and illegal, can be done on different ways. The following describes major ways and the appropriate counter measures against illegal operations:

- *Direct read-out via on-chip debug interface*

  Some devices contain the N-Wire / Nexus interface. This is basically a superset of the well known JTAG debug interface and allows full control over all data stored in the device. It can be protected by a password. As the protection is not directly a Flash feature, it is just mentioned for reference. Please refer to the device user manual or the N-Wire tools description for details.

- *Direct read-out via programming interface*

  The standard programming interface (e.g. PG-FP5) supports a command to read out the Flash contents on all current devices. This feature helps a lot in the developing and debugging phase and for failure analysis. This command can be disabled by a protection flag (see chapter 4.1.2, “Configuration options” for details)

- *Direct read-out by the application itself (via any interface)*

  E.g. a debug command in the application to dump memory. Please ensure that this possibility is not implemented or at least protected in your application.

- *Indirect read-out by spy software, programmed into the internal Flash*

  Software can be programmed into Flash in two different ways:

  - By the application itself using self-programming

    Please ensure, that this possibility is not implemented or at least protected in your application.

  - By the programmer interface

    In order to disable this feature, the commands Flash Write and Flash Block Erase can be disabled (see chapter 4.1.2, “Configuration options” for details). By doing so, Flash writing via this interface is only possible after erasing the complete Flash, but then no more data to be read is in the Flash any more.

Illegal or accidental reprogramming of Flash

For many applications protection against the illegal Flash read-out is already sufficient. In other cases reprogramming the device either completely or partly must be disabled. V850 devices provide features even for that:

- Partly reprogramming by the programmer interface
  See "**Illegal read-out of Flash contents**"
- Complete reprogramming by the programmer interface
  If also the complete erasing and reprogramming by this interface shall be disabled, in addition to Flash Write and Flash Block Erase commands also the Chip Erase command can be disabled (see chapter 4.1.2, "Configuration options" for details). By doing so the reprogramming via programmer interface is no longer possible, neither by unauthorized nor by authorized use. Reprogramming by the application using self-programming is still possible.
- Reprogramming by the application using self-programming
  It is also possible to disable reprogramming parts or the complete Flash via the application (see chapter 4.1.2, "Configuration options" for details). This protection is block wise organized, starting from 0x00000000. So it is possible to protect e.g. a Bootloader or more code and data up to the complete application.
  In addition a Flash Shield Window is able to protect parts of the Flash. This Window is configurable via self-programming. Only the Flash blocks covered by the FSW can be reprogrammed via self-programming.
  Hardware protection of the complete Flash is also possible. Please refer to 4.2.1, "Hardware Protection" for details.

Note:

When disabling reprogramming of blocks via the application, the secured part can no longer be reprogrammed in any way any more.

4.1.2 Configuration options

This chapter explains the protection relevant settings (stored in the Extra Area) and mechanisms, implemented in UX6LF based Flash devices.

For the usage of these settings and the protection strategy, please refer to section 4.1.1, "Strategy".

The protection configuration can be set by the dedicated Flash programmers, like PG-FP5 or via self-programming.

Note:

If set once, resetting is only possible by the Chip-Erase command using a dedicated Flash programmer.

The following flags and settings are available:

- Read command disable (Programmer interface)
  Reading the Flash contents via the programming interface is disabled. It does not affect self-programming (see 5.3.3.7, "FSL_SetReadProtectFlag").
- Program command disable (Programmer interface)
  Writing to Flash via programming interface is disabled. It does not affect self-programming (see 5.3.3.5, "FSL_SetWriteProtectFlag").

- Block Erase command disable (Programmer interface)
  Erasing single blocks via programming interface is disabled. It does not affect Chip Erase or self-programming. The Flag is valid for the complete Flash (see 5.3.3.4, “FSL_SetBlockEraseProtectFlag”).
- Chip Erase command disable (Programmer interface)
  Erasing the complete device via programming interface is disabled. It does not affect self-programming (see 5.3.3.3, “FSL_SetChipEraseProtectFlag”).
- Boot Cluster Protection
  If set, erasing and writing on the Flash by the application using the self-programming is disabled for the boot cluster (see 5.3.3.6, “FSL_SetBootClusterProtectFlag”).

Flash access via N-Wire / Nexus interface can be secured via an internal ID. The ID is stored in the Extra Area and has to match the configure ID in the N-Wire interface configuration of the debugger to allow Flash access. For details about the ID, please refer to chapter 5.3.4.12, “FSL_SetID”.

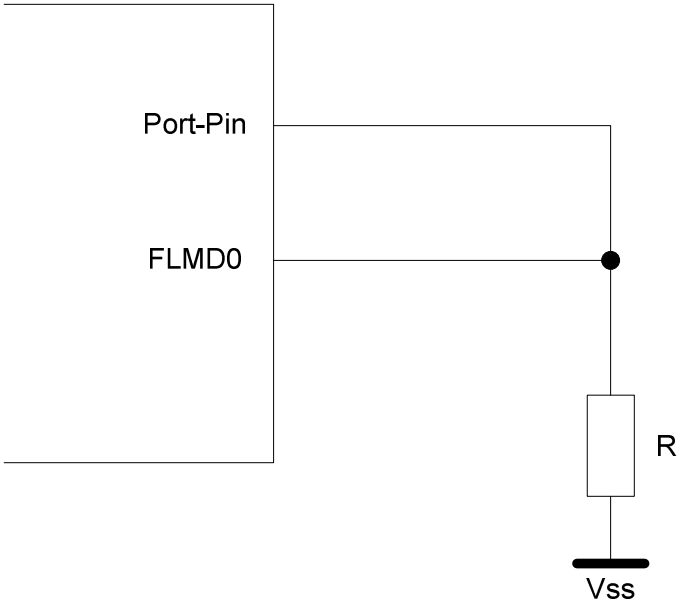
4.2 Flash Safety

All UX6LF based Flash devices are equipped with dedicated security features. The features have to be separated for normal operation, where data retention is important and for reprogramming, where secure reprogramming in case of power fail or other problems is important.

4.2.1 Hardware Protection

Device Reprogramming is disabled if FLMD0 Pin is low. By using a port pin or an external logic FLMD0 must be set to “1” to allow self-programming. Additionally reprogramming can be enabled by a register if supported by the device. Please refer to the device user manual for further details.

Figure 3 FLMD0 Sample circuit



In the sample circuit, the port pin is input on reset. By that FLMD0 is held to V_{SS} on reset. During self-programming the port is set to output and to the value "1". Then the FLMD0 pin is set to $V_{DD.}$

4.2.2 Normal operation (Error Correction Circuit – ECC)

UX6LF based Flash devices contain Error Correction Circuits (ECC) to provide correct Flash data. Beside the Flash data redundant ECC data is written into additional Flash cells to correct detected Flash errors if possible. ECC is an on-line method. That means from user point of view ECC has no impact on the data read performance.

4.2.3 Secure reprogramming using self-programming

When talking about secure self-programming, this naming needs to be exactly defined, as several different ways of understanding are possible.

Basic idea of secure self-programming is that if anything during reprogramming process goes wrong, it must be possible to keep basic application functionality alive. Usually it is solved by separation of the application into the application that is updated and therefore temporarily not valid during reprogramming, and a specific bootloader that must always be executable somehow again after power up or reset.

Two major options with different advantages and disadvantages have to be considered. Depending on the application and bootloader the appropriate solution has to be selected:

- Secure self-programming without bootloader update
- Secure self-programming with bootloader update

4.2.3.1 Secure self-programming without bootloader update

The easiest way of secure self-programming is to occupy one or more complete Flash blocks for the bootloader and do not reprogram them again. By that it never happens, that an interruption of the reprogramming (e.g. power fail) causes an invalid bootloader.

This method is only possible if data, that needs to be reprogrammed using self-programming, is not located on one of the bootloader Flash blocks. This is because as soon as data needs to be reprogrammed, the Flash block needs to be erased and so part of the bootloader is temporarily not available.

Although this method might waste some space if the bootloader does not occupy a complete Flash block, the handling of reprogramming is easy.

Furthermore, increased safety by protection against reprogramming the bootloader due to program failures is possible. The block protection feature can be used to protect the bootloader forever against any reprogramming. In that case, please consider that the block cannot be reprogrammed in any way any more.

4.2.3.2 Secure self-programming with bootloader update

Bootloader block update might be necessary due to the following items:

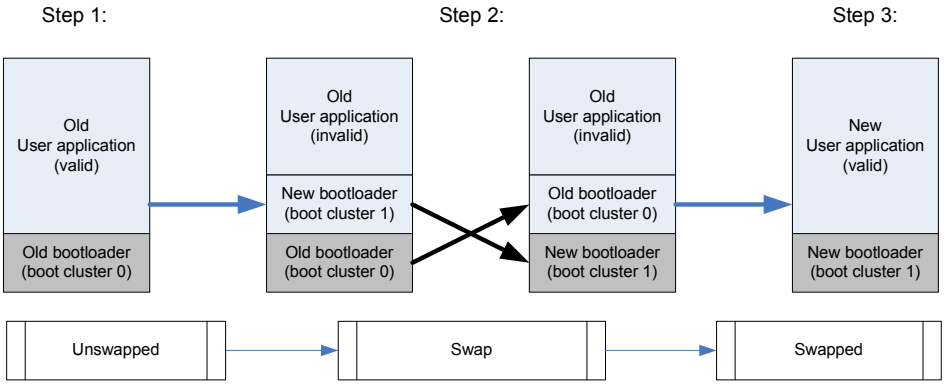
- Keep the option to fix bootloader bugs.
- Application code/data, that needs to be updated, is stored in the same block as the bootloader.

If the bootloader has to be updated, it needs to be ensured, that always a working version of the bootloader is available, even during the update procedure. Furthermore, in case of a power failure the valid bootloader needs

to be detected and the program has to be started there. To fulfil these requirements, the Boot Swap functionality is implemented.

Boot swap means, that two Flash blocks or cluster of blocks can be swapped in the address range. This swapping is done depending on Boot Swap bits, set in the corresponding Flash Extra Areas. When a valid bootloader is contained in the corresponding block and the bits are set accordingly, the block is automatically swapped to the address 0x00000000 on device start-up. By that and by the correct reprogramming sequence can be ensured, that even a block containing a bootloader can be updated securely.

Figure 4 Secure bootloader update



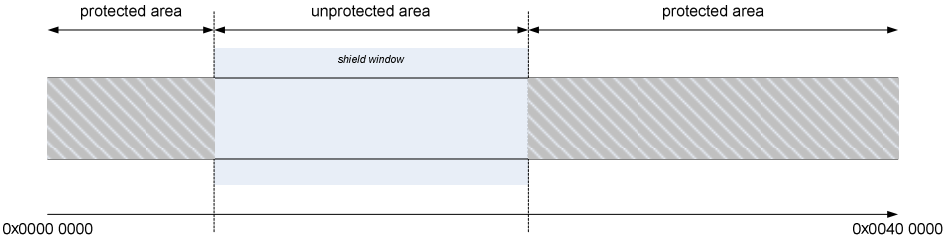
Two methods are implemented in the Library to swap the boot cluster. The first method only temporary swaps the boot cluster, but the cluster will be unswapped again after a device reset. For details, please refer to chapter 5.3.4.6, “FSL_ChangeSwapState”.

The second method inverts the boot flag. Therefore the boot cluster is changed after a reset. An additional parameter forces the device to swap the boot cluster immediately in addition to changing the swap flag. Please refer to 5.3.4.8, “FSL_ChangeSwapFlag”.

4.2.4 Flash Shield Window

Beside hardware protection by a dedicated pin, internal Flash is additionally protected from accidental reprogramming by a shield window. This window is configurable during runtime. It allows to program or to erase all Flash blocks covered by the window and denies destructive access to all other blocks. Per default all Flash blocks are covered by the Flash Shield Window. For details how to configure the Flash Shield Window, please refer to chapter 5.3.3.9, “FSL_SetFSW”.

Figure 5 Flash Shield Window



4.3 Code execution in RAM

The application, including the control program and the FSL are usually located in the internal flash. As the memory location of the application is not permanently available during self-programming, parts of the program need to be copied to a “save” location, where they can be executed. This may be the internal RAM, but also external RAM, if available, is acceptable.

To copy necessary code parts into available RAM, three different methods are possible:

- C-Startup:

The code is linked to the destination address. The compiler startup routines copy the code from a ROM image to the RAM. Please refer to the compiler documentation, “ROM linker section attribute”, for details.

- FSL_CopySections

By calling FSL_CopySections all specified sections are copied to the destination address.

- User specific

In case of a user specific implementation, the user is responsible for the correct location of the sections.

Note:

During RAM execution as well as during ROM execution, the device tries to speed up execution time by a code prefetch mechanism. This prefetch mechanism is responsible for ECC errors in case of uninitialized RAM areas. Therefore the user has to initialize 32 Bytes at the end of the RAM placed code in case of a user specific implementation.

Depending on the configured mode (see section 5.1, “Pre-compile configuration”) following linker sections need to be copied to RAM:

User mode

- FSL_CODE_RAM_USRINT
- FSL_CODE_RAM_USR
- FSL_CODE_RAM
- FSL_CODE_ROMRAM
- FSL_CODE_EX_PROT

Internal mode

- FSL_CODE_RAM_USRINT
- FSL_CODE_RAM_USR
- FSL_CODE_RAM
- FSL_CODE_EX_PROT

For further information regarding the linker sections please refer to chapter 3.2, “FSL Linker sections”.

Note:

Beside the mentioned sections, the Flash Self-Programming Library needs additional 4kByte of RAM located on the top of the RAM. These

RAM addresses are reserved for the Flash Macro Services and the internal firmware. The FSL will destroy the RAM content on these addresses during self-programming.

4.4 User code execution during self-programming

The activation and deactivation of the Self-Programming Environment can be handled by the FSL automatically or by the user application.

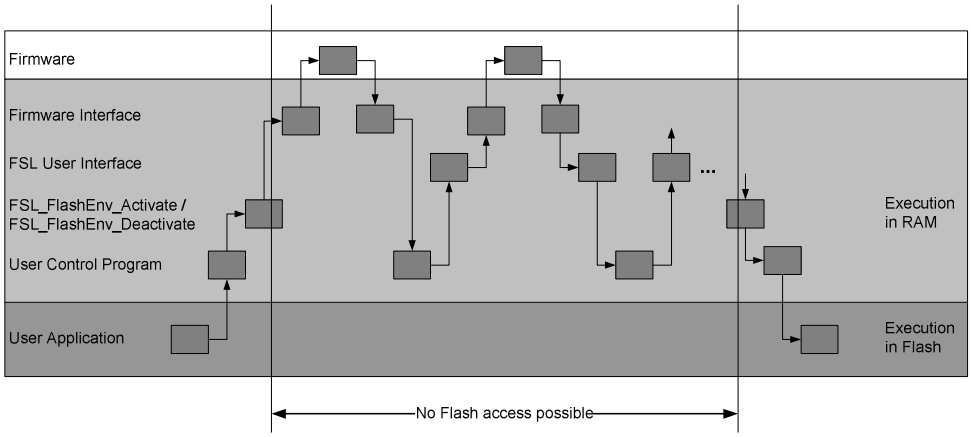
Especially activation is time consuming. In order to achieve fast reprogramming the environment should be kept activated during the whole reprogramming. On the other hand, during activated environment the program execution cannot be done from Flash. So other memory like internal RAM or external memory is required. If not sufficient memory is available, sequential activation and deactivation is necessary and only small code parts are executed from internal RAM.

Following two major scenarios can be considered for self-programming, that are reflected by the library modes:

- User mode

  Most parts of the Self-Programming Library are executed in the internal RAM, additionally the reprogramming control functions and other user code to be executed during self-programming. In order to realise fast reprogramming the activation/deactivation sequence is done only once for the complete reprogramming. Every code to be executed between activation and deactivation needs to be executed outside the Flash.

Figure 6 Reprogramming sequence in user mode



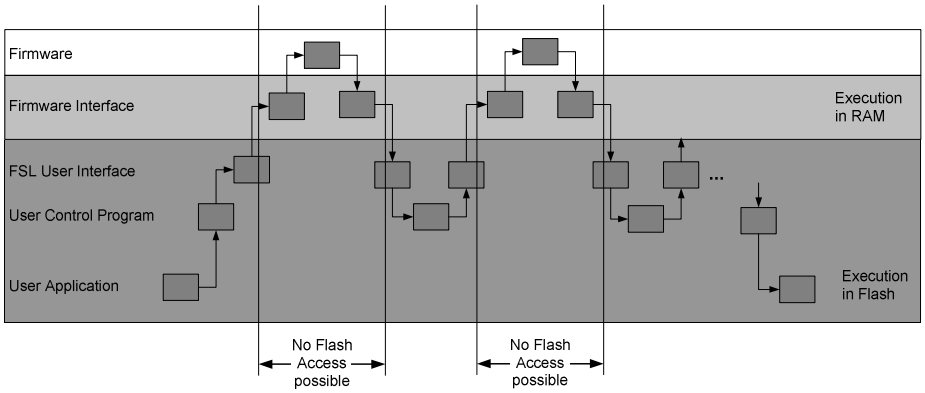
  This sequence is best for devices with sufficient internal RAM. User code execution is always possible during self-programming, because a started FSL command returns to the user application right after the command execution started. The user has to poll the command status via the status check function. Interrupt as well as user code execution is possible if all related functions are located in RAM.

  To enable this mode, the library must be configured to use the user mode (see 5.1, “Pre-compile configuration”).

- Internal mode

  Only small parts of the library are executed in RAM, the rest is executed in the Code Flash. Frequent activation and deactivation of the Flash Environment is necessary and therefore programming time will increase.

Figure 7 Basic RAM saving reprogramming sequence



Less internal RAM is used as only the device firmware interface need to be executed in RAM. On the other hand, user code execution during self-programming is impossible, because a FSL function starting a command does not return until the operation is finished. Therefore only interrupts are possible during self-programming.

To enable this mode, the library must be configured to use the internal mode (see 5.1, “Pre-compile configuration”).

4.5 Interrupts in RAM

As mentioned before, Code Flash is not accessible during self-programming. Therefore the interrupt vector table as well as interrupt handler routines, which are normally located in the flash, are also not accessible. Interrupt acknowledges have to re re-routed to external or internal RAM.

Two methods exist to execute interrupts from RAM:

- Single interrupt vector

  All interrupts are mapped to the single interrupt vector of interrupt channel 0. Based on this interrupt, the interrupt handler routine has to handle all pending interrupts.

- Interrupt table mapped to RAM

  The base address of the interrupt vector table is mapped to a different location in RAM. In this case the offset of the different channels is added to the new base address.

Regardless which method is used, interrupt service routines have to be executed from and therefore copied to RAM. For details how to copy the routines to RAM, please refer to chapter 4.3, “Code execution in RAM”.

Note:

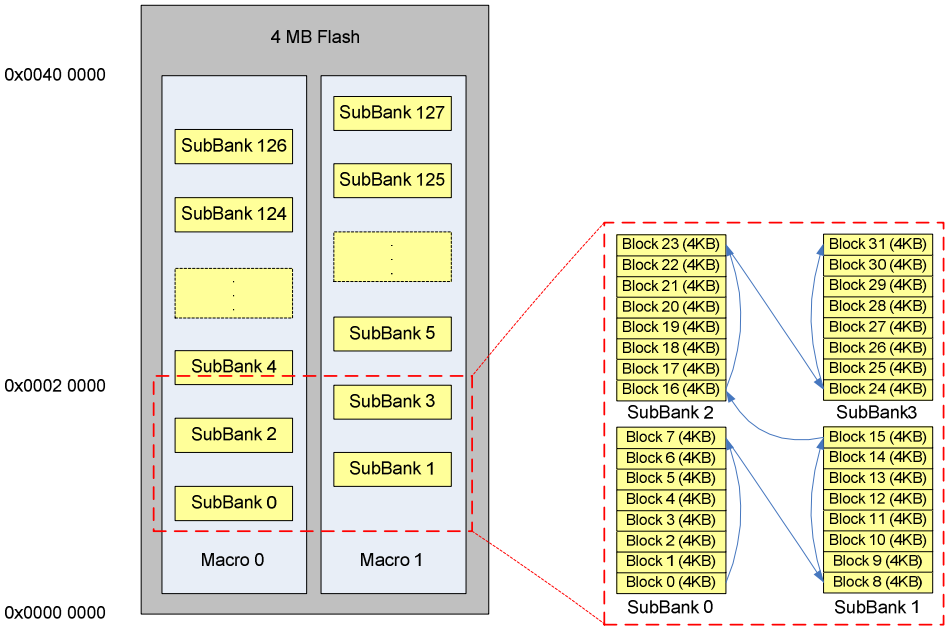
Further information about interrupt handling from RAM can be found in the device user manual and in the CPU architecture description (see “*V850E2R-V3 Architecture*”).

4.6 Self-programming performance

Depending on the specification, design and technology UX6LF Code Flash based microcontrollers contain a certain number of Flash macros with a certain size.

Each Flash macro consists of several sub-banks. Each sub-bank is subdivided into several Flash blocks.

Figure 8 Flash structure example



During self-programming it is possible to access both macros simultaneously. This feature is used to accelerate the standard sequence erase - write.

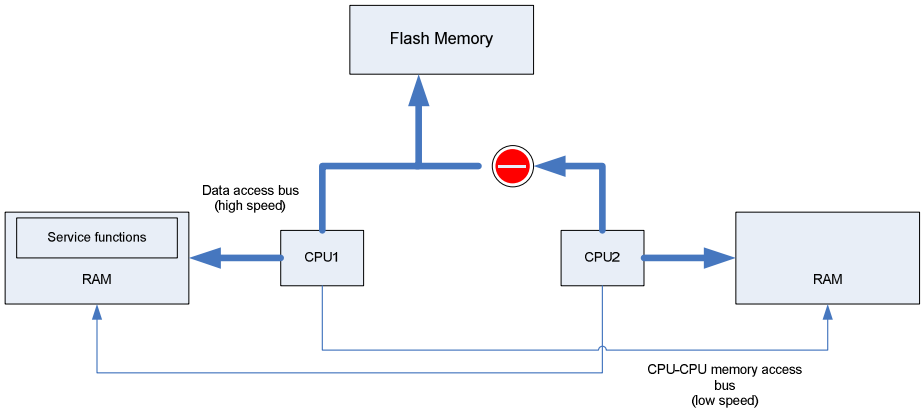
E.g. block 0 to 31 shall be reprogrammed during an application update. The internal firmware erases SubBank 0 first. After that it erases SubBank1 and rewrites SubBank 0 in parallel. Then it erases SubBank2 and writes SubBank 1 in parallel and then SubBank 3 and SubBank 2. The last step is to program SubBank 3. Please refer to chapter 5.3.2.4, “FSL_EraseWrite” and 5.3.2.5, “FSL_EraseWriteCont” for a detailed API description.

4.7 Misc

4.7.1 Dual CPU operation

In case of a dual CPU device the usage of the FSL is not limited to one CPU. The Flash memory can be controlled by each CPU.

Figure 9 Dual CPU operation



Dual CPU operation causes some smaller restrictions. The service functions are always located in RAM area of CPU1. Therefore the function response time will increase in case of control by CPU2.

A second restriction is access control in general. To provide a fail safe mechanism, only access by one CPU at a time is allowed. Simultaneous access by the other CPU is prohibited. Access rights are controlled automatically by the library.

4.7.2 Option Byte

The Extra Area contains user specific configuration data called Option Byte. These configuration settings are adjustable via self-programming. Depending on the used device, size of the Option Byte various between 4Byte and 36Byte. For details about possible configuration settings please refer to the device user manual.

Chapter 5 User Interface (API)

5.1 Pre-compile configuration

The pre-compile configuration of the FSL is located in the FSL_cfg.h. The user has to configure all parameters and attributes by adapting the related constant definition in that header-file.

The configuration contains the following elements:

FSL_STATUS_CHECK
defines whether the status check should be performed by the firmware or by the user to allow execution of user code in between the status checks.

```
#define FSL_STATUS_CHECK          FSL_STATUS_CHECK_INETRNAL
```

Following configuration options are possible:

- FSL_STATUS_CHECK_INTERNAL
- FSL_STATUS_CHECK_USER

As described in the previous chapter the library behavior changes depending on the configure mode. Summarizing the following behavior is possible:

User mode (FSL_STATUS_CHECK_USER)

Advantages:

- less CPU time
- less activation / deactivation time
- user code execution during self-programming

Disadvantages:

- more RAM consumption
- polling necessary

Internal mode (FSL_STATUS_CHECK_INTERNAL)

Advantages:

- no polling necessary
- less RAM consumption

Disadvantages:

- more activation / deactivation time
- 100% CPU time during self-programming
- user code execution during self-programming not possible

For details please refer to chapter 4.4, “User code execution during self-programming”.

5.2 Data Types

Figure 10 FSL status and error codes

Error	Value	Explanation	FSL Impact
FSL_ERR_FLMD0	0x01	The FLMD0-Pin is not at a High level.	Current command rejected
FSL_ERR_PARAMETER	0x05	A new operation should be initiated, but an error in the given parameter occurred.	Current command rejected
FSL_ERR_PROTECTION	0x10	A new operation should be initiated although this operation is forbidden due to a security feature.	Current command rejected
FSL_ERR_ERASE	0x1A	The current operation stopped due to an error while erasing.	Current command aborted.
FSL_ERR_BLANKCHECK	0x1B	The current operation stopped due to an error while blank checking.	Current command aborted.
FSL_ERR_IVERIFY	0x1B	The current operation stopped due to an error while verifying.	Current command aborted.
FSL_ERR_WRITE	0x1C	The current operation stopped due to an error while writing.	Current command aborted.
FSL_ERR_FLOW	0x1F	A new operation should be initiated although the state machine is still busy	Current command rejected
FSL_OK	0X00	The operation finished successfully	Correct result
FSL_IDLE	0x30	No operation is ongoing	A new function call is possible
FSL_REQ_UPDBUF	0x40	The data buffer needs to be updated.	The current erase and write flow is suspended. After the update of the buffer the flow can be resumed.
FSL_BUSY	0XFF	The operation has been started successfully and is still running.	Correct result

5.3 Functions

Functions represent the application interface to the FSL which the user SW can use.

5.3.1 Initialization

5.3.1.1 FSL_Init

Description

Function is executed before any execution other FSL function. It initializes internal self-programming environment and internal variables.

Interface

```
void FSL_Init( void )
```

Arguments

None

Return types/values

None

Pre-conditions

None

Post-conditions

None

Example

```
/* Initialze and start Self-Programming Library */
FSL_Init( );
```

5.3.1.2 FSL_CopySections

Description

Function is executed after FSL_Init and before execution of any other FSL function. It copies FSL functions to a specific destination address. This function is not neccessary if the FSL code is copied to RAM by a user defined copy function or by the C-Startup. For details please refer to 4.3, “Code execution in RAM”.

Interface

```
void FSL_CopySections( fsl_u32 addDest_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	addDest_u32	Destination address of Self-Programming Library in RAM

Return types/values

None

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details).

Post-conditions

None

Example

```
/* Copy FSL to internal RAM address 0xffff7000 */
FSL_CopySections( 0xffff7000 );
```

5.3.1.3 FSL_FlashEnv_Activate

Description

Function initializes the Flash control macro and activates and prepares the Flash environment. This function is only available in the user mode.

In internal mode, it is executed in the library, transparent for the user.

Interface

```
fsl_status_t FSL_FlashEnv_Activate( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_FLOW

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details).

Post-conditions

None

Example

```
/* Enable Flash environment */
fsl_status_t  status_enu;

status_enu = FSL_FlashEnv_Activate( );

/* Error treatment */
...
```

5.3.1.4 FSL_FlashEnv_Deactivate

Description

Function deactivates the Flash environment after termination of all ongoing Flash operations. This function is only available in the user mode.

In internal mode, it is executed in the library, transparent for the user.

Interface

```
fsl_status_t FSL_FlashEnv_Deactivate( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_FLOW

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, ”FSL_CopySections” for details).

Post-conditions

None

Example

```
/* Deactivate Flash environment */
fsl_status_t  status_enu;

status_enu = FSL_FlashEnv_Deactivate( );

/* Error treatment */
...
```

5.3.2 Operation

5.3.2.1 FSL_BlankCheck

Description

Function checks wether a range of blocks is erased.

Interface

```
fsl_status_t FSL_BlankCheck( fsl_u32 blockNoStart_u32,
                             fsl_u32 blockNoEnd_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	blockNoStart_u32	Starting block number of checked range
fsl_u32	blockNoEnd_u32	Ending block number of checked range

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK [1] FSL_BUSY [2] FSL_ERR_PARAMETER FSL_ERR_FLOW FSL_ERR_BLANKCHECK [1]

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Blank check block 3 to 20 */
fsl_status_t  status_enu;

status_enu = FSL_BlankCheck( 3, 20 );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.2.2 FSL_Erase

Description

Function erases a range of blocks.

Interface

```
fsl_status_t FSL_Erase( fsl_u32 blockNoStart_u32,
                        fsl_u32 blockNoEnd_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	blockNoStart_u32	First block number to be erased. (It is not the block address, but the number of the Flash block.)
fsl_u32	blockNoEnd_u32	Last block number to be erased. (It is not the block address, but the number of the Flash block.)

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK [1] FSL_BUSY [2] FSL_ERR_PARAMETER FSL_ERR_PROTECTION FSL_ERR_FLOW FSL_ERR_ERASE [1]

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Erase block 3 to 20 */
fsl_status_t  status_enu;

status_enu = FSL_Erase( 3, 20 );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.2.3 FSL_Write

Description

Function writes the specific number of words from a buffer to consecutive Flash addresses starting at the specific adress.

Interface

```
fsl_status_t FSL_Write( fsl_u32 *pAddSrc_pu32,
                        fsl_u32 addDest_u32,
                        fsl_u32 length_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	pAddSrc_pu32	Pointer to source address of data to be written
fsl_u32	addDest_u32	Destination address of data to be written aligned 128 bit
fsl_u32	length_u32	Word length of data aligned 128 bit

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK [1] FSL_BUSY [2] FSL_ERR_PARAMETER FSL_ERR_PROTECTION FSL_ERR_FLOW FSL_ERR_WRITE [1]

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Write 64 words of data to address 0x00000000 onwords */
fsl_status_t  status_enu;
fsl_u32       buf_u32[256];

/* fill buffer */
...

status_enu = FSL_Write( buf_u32, 0x00000000, 64 );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.2.4 FSL_EraseWrite

Description

Function erases a range of blocks including all blocks within the range of start and end address. Additionally the function writes the specific number of words from a buffer to consecutive Flash addresses starting at the specific adress. To accelerate, programming functionally of FSL_Erase and FSL_Write is combined in one function and the operations are largely executed in parallel. This function is only available in the user mode.

Interface

```
fsl_status_t FSL_EraseWrite( fsl_u32 *pAddSrc_pu32,
                             fsl_u32 addStart_u32,
                             fsl_u32 addEnd_u32,
                             fsl_u32 bufSize_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	pAddSrc_pu32	Pointer to source address of data to be written
fsl_u32	addStart_u32	Start address for erase and write operation aligned 128 bit
fsl_u32	addEnd_u32	End address for erase and write operation aligned 128 bit
fsl_u32	length_u32	RAM buffer size in bytes aligned 128 bit

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_BUSY FSL_ERR_PARAMETER FSL_ERR_PROTECTION FSL_ERR_FLOW

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

When return value is equal to FSL_REQ_UPDBUF, update the write buffer with data to write next and continue write (see 5.3.2.5, “FSL_EraseWriteCont”).

Example

```
/* Erase and write from address 0x00000000 to address 0x000040000 */
fsl_status_t  status_enu;
fsl_u32       buf_u32[256];

/* fill buffer */
...

status_enu = FSL_EraseWrite( &buf_u32[0], 0x00000000,
                             0x00040000, 256 );
while( ( status_enu == FSL_BUSY ) ||
       ( status_enu == FSL_REQ_UPDBUF ) )
{
       status_enu = FSL_StatusCheck( );
       if( status_enu == FSL_REQ_UPDBUF )
       {
              /* Update buffer */
              ...
              status_enu = FSL_EraseWriteCont( ); /* continue */
       }
}

/* Error treatment */
...
```

5.3.2.5 FSL_EraseWriteCont

Description

Restarts erase and write procedure after updating the data buffer if requested. For detaills of erase and write see chapter 5.3.2.4, “FSL_EraseWrite”5.3.2.4. This function is only available in the user mode.

Interface

```
fsl_status_t FSL_EraseWriteCont( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_BUSY FSL_ERR_FLOW

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Calling the function is only valid if previous status check (see 5.3.2.8, “FSL_StatusCheck”) of a preceding write (see 5.3.2.3, “FSL_Write”) or a continued write (see 5.3.2.5, “FSL_EraseWriteCont”) ended with FSL_REQ_UPDBUF.

Post-conditions

See chapter 5.3.2.4, “FSL_EraseWrite”5.3.2.4.

Example

See chapter 5.3.2.4, “FSL_EraseWrite”5.3.2.4.

5.3.2.6 FSL_IVerify

Description

The function compares the Flash contents on erase level against the write level.

Note:

The Internal Verify is a check for higher reliability of the programming, but no manipulation or operation on Flash cells is performed. It is not mandatory but recommended on Code Flash in order to detect possible problems that might occur during programming (e.g. noise, V_{DD} / GND bounce).

Interface

```
fsl_status_t FSL_IVerify( fsl_u32 blockNoStart_u32,
                          fsl_u32 blockNoEnd_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	blockNoStart_u32	First block number to be verified. (It is not the block address, but the number of the Flash block.)
fsl_u32	blockNoEnd_u32	Last block number to be verified. (It is not the block address, but the number of the Flash block.)

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK [1] FSL_BUSY [2] FSL_ERR_PARAMETER FSL_ERR_FLOW FSL_ERR_IVERIFY [1]

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Verify block 3 to 20 */
fsl_status_t  status_enu;

status_enu = FSL_IVerify( 3, 20 );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.2.7 FSL_Read

Description

Function reads the specified number of words from consecutive Flash addresses starting at the specified address and writes it into a buffer.

Interface

```
fsl_status_t FSL_Read( fsl_u32 addSrc_u32,
                       fsl_u32 *pAddDest_pu32
                       fsl_u32 length_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	addSrc_u32	Source address of data to be read
fsl_u32	pAddDest_pu32	Pointer to destination address of data to be read
fsl_u32	length_u32	Word length of data

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK [1] FSL_BUSY [2] FSL_ERR_PARAMETER FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Read 64 words from address 0x00000000 onwords */
fsl_status_t  status_enu;
fsl_u32       buf_u32[256];

status_enu = FSL_Read( 0x00000000, &buf_u32[0], 64 );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.2.8 FSL_StatusCheck

Description

This function handles the complete state machine. It shall be called frequently, by the user application.

Interface

```
void FSL_StatusCheck( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_IDLE FSL_BUSY FSL_ERR_ERASE FSL_ERR_BLANKCHECK FSL_ERR_IVERIFY FSL_ERR_WRITE FSL_ERR_FLOW FSL_REQ_UPDBUF

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

Status check called to check FSL_Erase operation result

```
/* Show FSL_StatusCheck usage */
fsl_status_t status_enu;

/* some FSL operation (e.g. FSL_Erase) */
status_enu = FSL_Erase( 3, 20 );

/* Status Check */
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.3 Security

5.3.3.1 FSL_GetSecurityFlags

Description

Function reads stored security information.

Interface

```
fsl_status_t FSL_GetSecurityFlags( fsl_u32 *pFlags_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	pFlags_pu32	Pointer to destination address of security information to be read

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pFlags_u32	Pointer to buffer filled with bit coded security information: 1xxxx: Read permission 0xxxx: Read prohibition x1xxx: Write permission x0xxx: Write prohibition xx1xx: Chip erase permission xx0xx: Chip erase prohibition xxx1x: Block erase permission xxx0x: Block erase prohibition xxxx1: Permission of boot block cluster programming xxxx0: Prohibition of boot block cluster programming

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read security flags */
fsl_status_t  status_enu;
fsl_u32       flags_u32;

status_enu = FSL_GetSecurityFlags( &flags_u32 );

/* Error treatment */
...
```

5.3.3.2 FSL_ModeCheck

Description

Function checks wether the FLMD0 pin (hardware protection shield) is pulled up or not. In case of pull down, no Flash programming is possible.

Interface

```
fsl_status_t FSL_ModeCheck( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_FLMD0 FSL_ERR_FLOW

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Check level of FLMD0 */
fsl_status_t  status_enu;

status_enu = FSL_ModeCheck( );

/* Error treatment */
...
```

5.3.3.3 FSL_SetChipEraseProtectFlag

Description

Function enables chip erase protection by setting the according protection flag.

Note:

After setting the chip erase protection flag it is neither by the Flash Self-Programming Library nor by an external programmer, e.g. PG-FP5, possible to remove the flag again.

Interface

```
fsl_status_t FSL_SetChipEraseProtectFlag( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Set chip erase protection */
fsl_status_t  status_enu;

status_enu = FSL_SetChipEraseProtectFlag( );

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

/* Error treatment */
...
```

5.3.3.4 FSL_SetBlockEraseProtectFlag

Description

Function enables block erase protection by setting the according protection flag.

Interface

```
fsl_status_t FSL_SetBlockEraseProtectFlag( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Set block erase protection */
fsl_status_t  status_enu;

status_enu = FSL_SetBlockEraseProtectFlag( );

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

/* Error treatment */
...
```

5.3.3.5 FSL_SetWriteProtectFlag

Description

Function enables write protection by setting the according protection flag.

Interface

```
fsl_status_t FSL_SetWriteProtectFlag( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Set write protection */
fsl_status_t  status_enu;

status_enu = FSL_SetWriteProtectFlag( );

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

/* Error treatment */
...
```

5.3.3.6 FSL_SetBootClusterProtectFlag

Description

Function enables boot cluster protection by setting the according protection flag.

Note:

After setting the boot cluster protection flag it is neither by the Flash Self-Programming Library nor by an external programmer, e.g. PG-FP5, possible to remove the flag again.

Interface

```
fsl_status_t FSL_SetBootClusterProtectFlag( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

In addition it is necessary to set the boot cluster size beforehand (please refer to chapter 5.3.4.10, “FSL_SetBootClusterSize” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Set boot cluster protection */
fsl_status_t  status_enu;

status_enu = FSL_SetBootClusterProtectFlag( );

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

/* Error treatment */
...
```

5.3.3.7 FSL_SetReadProtectFlag

Description

Function enables read cluster protection by setting the according protection flag.

Interface

```
fsl_status_t FSL_SetReadProtectFlag( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Set read protection */
fsl_status_t  status_enu;

status_enu = FSL_SetReadProtectFlag( );

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

/* Error treatment */
...
```

5.3.3.8 FSL_GetFSW

Description

Function returns the start and the end block of the actual Flash shield window.

Interface

```
fsl_status_t FSL_GetFSW( fsl_u32 *pBlockNoStart_pu32,
                         fsl_u32 *pBlockNoEnd_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	pBlockNoStart_pu32	Pointer to buffer for starting block number of FSW
fsl_u32	pBlockNoEnd_pu32	Pointer to buffer for ending block number of FSW

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pBlockNoStart_pu32	Buffer containing starting block number of FSW
fsl_u32	pBlockNoEnd_pu32	Buffer containing ending block number of FSW

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read Flash Shield Window range */
fsl_status_t  status_enu;
fsl_u32       blockStart_u32;
fsl_u32       blockEnd_u32;

status_enu = FSL_GetFSW( &blockStart_u32, &blockEnd_u32 );

/* Error treatment */
...
```

5.3.3.9 FSL_SetFSW

Description

Function sets a new Flash shield window to protect the range of blocks from unwanted Flash operations.

Interface

```
fsl_status_t FSL_SetFSW( fsl_u32 blockNoStart_u32,
                         fsl_u32 blockNoEnd_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	blockNoStart_u32	Starting block number of FSW
fsl_u32	blockNoEnd_u32	Ending block number of FSW

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_PARAMETER FSL_ERR_PROTECTION FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Set Flash Shield Window for block 2 up to block 3 */
fsl_status_t  status_enu;

status_enu = FSL_GetFSW( 2, 3 );

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

/* Error treatment */
...
```

5.3.4 Administration

5.3.4.1 FSL_GetVersionString

Description

This function returns a pointer to the library version string. The version string is a zero terminated string identifying the library.

Interface

```
const fsl_u08* FSL_VersionString( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_u08		Pointer to version string. Version string format: "SV850T05xxxxyZabc" with x = supported compiler y = compiler option Z = "E" for engineering version, "V" for final version# abc = Library version number Va.b.c

Pre-conditions

None

Post-conditions

None

Example

```
/* Get library version */
fsl_u08       *vstr_pu08;

vstr_pu08 = FSL_GetVersionString( );
```

5.3.4.2 FSL_GetDevice

Description

Function returns the device number to identify the used device.

Interface

```
fsl_status_t FSL_GetDevice( fsl_u32 *pDeviceNo_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	pDeviceNo_pu32	Pointer to buffer for device number

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pDeviceNo_pu32	Buffer containing device number. Device number format: 0000000000000000xxxxxxxxxxxxxxxx (Example: uPD70F3377 → 3377 = 0x0D31 → xxxxxxxxxxxxxxxx = 0001101001100001)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read device name */
fsl_status_t  status_enu;
fsl_u32       device_u32;

status_enu = FSL_GetDevice( &device_u32 );

/* Error treatment */
...
```

5.3.4.3 FSL_GetBlockCnt

Description

Function returns the number of blocks of the device.

Interface

```
fsl_status_t FSL_GetBlockCnt( fsl_u32 *pBlockCnt_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	pBlockCnt_pu32	Pointer to buffer for block count

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pBlockCnt_pu32	Buffer containing block count

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read block count */
fsl_status_t  status_enu;
fsl_u32       cnt_u32;

status_enu = FSL_GetBlockCnt( &cnt_u32 );

/* Error treatment */
...
```

5.3.4.4 FSL_GetBlockEndAdd

Description

Function returns the end address of the specified block.

Interface

```
fsl_status_t FSL_GetBlockEndAdd( fsl_u32 blockNo_u32,
                                 fsl_u32 *pBlockEndAddr_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	blockNo_u32	Block number
fsl_u32	pBlockEndAddr_pu32	Pointer to buffer for block end address information

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pBlockEndAddr_pu32	Buffer containing block end address

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read block end address of block 3 */
fsl_status_t  status_enu;
fsl_u32       addr_u32;

status_enu = FSL_GetBlockEndAddr( 3, &addr_u32 );

/* Error treatment */
...
```

5.3.4.5 FSL_GetSwapState

Description

Function returns the current swap status.

Interface

```
fsl_status_t FSL_GetSwapState( fsl_u32 *pSwapState_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	pSwapState_pu32	Pointer to buffer for swap status

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pSwapState_pu32	Buffer containing swap status

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read boot cluster */
fsl_status_t  status_enu;
fsl_u32       state_u32;

status_enu = FSL_GetSwapState( &state_u32 );

/* Error treatment */
...
```

5.3.4.6 FSL_ChangeSwapState

Description

Function swaps the boot cluster 0 and boot cluster 1 physically without setting the boot flag. After reset the boot cluster will be activated again as defined by the swap flag.

Interface

```
fsl_status_t FSL_ChangeSwapState( void )
```

Arguments

None

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PROTECTION FSL_ERR_FLOW

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Swap boot cluster */
fsl_status_t  status_enu;

status_enu = FSL_ChangeSwapState( );

/* Error treatment */
...
```

5.3.4.7 FSL_GetSwapFlag

Description

Function reads the current value of the boot swap flag from the Extra Area.

Interface

```
fsl_status_t FSL_GetSwapFlag( fsl_u32 *pSwapFlag_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	pSwapFlag_pu32	Pointer to buffer for swap flag

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pSwapFlag_pu32	Buffer containing swap flag Flag values: 0x00: not swapped 0x01: swapped

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read boot cluster */
fsl_status_t  status_enu;
fsl_u32       flag_u32;

status_enu = FSL_GetSwapFlag( &flag_u32 );

/* Error treatment */
...
```

5.3.4.8 FSL_ChangeSwapFlag

Description

The function inverts the bootswap flag. Depending on the parameter, additionally the current swap state is inverted.

Interface

```
fsl_status_t FSL_ChangeSwapFlag( fsl_u32 immediateSwap_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	immediateSwap_u32	Flag for swap boot cluster immediately: Flag values: 0x00: do not swap boot cluster 0x01: swap boot cluster

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_PROTECTION FSL_ERR_FLOW
fsl_u32	pSwapFlag_pu32	Buffer containing swap flag Flag values: 0x00: not swapped 0x01: swapped

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Swap the boot flag, but do not generate a reset signal */
fsl_status_t  status_enu;

status_enu = FSL_ChangeSwapFlag( 0x00 );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.4.9 FSL_GetBootClusterSize

Description

Function reads current size of protectable boot cluster.

Interface

```
fsl_status_t FSL_GetBootClusterSize ( fsl_u32 *pSize_pu32 )
```

Arguments

Type	Argument	Description
fsl_u32	pSize_pu32	Pointer to buffer for boot cluster size information

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u32	pSize_pu32	Buffer containing boot cluster size information

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read boot cluster size */
fsl_status_t  status_enu;
fsl_u32       size_u32;

status_enu = FSL_GetBootClusterSize( &size_u32 );

/* Error treatment */
...
```

5.3.4.10 FSL_SetBootClusterSize

Description

Function sets protectable boot cluster size and resulting boot swap cluster size. Although the cluster size is variable, swapping is only possible on some addresses.

Size value	Boot cluster size	Boot swap cluster size
0x00	4kByte	4kByte
0x01	8kByte	8kByte
0x02	12kByte	16kByte
0x03	16kByte	16kByte
0x04-0x07	20-32kByte	32kByte
0x08-0x0F	36-64kByte	64kByte
0x10-0x1F	68-128kByte	128kByte
0x20-0xFF	132-1024kByte	256kByte

Interface

```
fsl_status_t FSL_SetBootClusterSize ( fsl_u32 size_u32 )
```

Arguments

Type	Argument	Description
fsl_u32	size_u32	Boot cluster size (range: 0x00-0xFF)

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_PARAMETER FSL_ERR_PROTECTION FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Set boot cluster size to 0x04 */
fsl_status_t  status_enu;

status_enu = FSL_SetBootClusterSize( 0x04 );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.4.11 FSL_GetID

Description

Function reads current ID data information (12 bytes).

Interface

```
fsl_status_t FSL_GetID ( fsl_u08 *pID_pu08 )
```

Arguments

Type	Argument	Description
fsl_u08	pID_pu08	Pointer to buffer for ID information

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u08	pID_pu08	Buffer containing ID information

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read ID */
fsl_status_t  status_enu;
fsl_u08       id_u08[12];

status_enu = FSL_GetID( &id_u08[0] );

/* Error treatment */
...
```

5.3.4.12 FSL_SetID

Description

Function writes new ID settings into the Extra Area (12 bytes).

Interface

```
fsl_status_t FSL_SetID ( fsl_u08 *pID_pu08 )
```

Arguments

Type	Argument	Description
fsl_u08	pID_pu08	Pointer to buffer containing ID information

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_PARAMETER FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Write ID */
fsl_status_t  status_enu;
fsl_u08 id_u08[] =   {
                           0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
                           0x06, 0x07, 0x08, 0x09, 0x0A, 0x8B
                     };    /* OCD on -> bit 95 = 1 */

status_enu = FSL_SetID( & id_u08[0] );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

5.3.4.13 FSL_GetOPB

Description

Function reads current OPB settings from the device. Depending on the device the number of option bytes various. Please see also chapter 4.7.2 Option Byte for details.

Interface

```
fsl_status_t FSL_GetOPB ( fsl_u08 *pOPB_pu08 )
```

Arguments

Type	Argument	Description
fsl_u08	pOPB_pu08	Pointer to buffer for option byte information

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK FSL_ERR_PARAMETER FSL_ERR_FLOW
fsl_u08	pOPB_pu08	Buffer containing option byte information

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

None

Example

```
/* Read option byte */
fsl_status_t  status_enu;
fsl_u08       data_u08[4];

status_enu = FSL_GetOPB( &data_u08[0] );

/* Error treatment */
...
```

5.3.4.14 FSL_SetOPB

Description

Function writes new OPB settings into the Extra Area.

Interface

```
fsl_status_t FSL_SetOPB ( fsl_u08 *pID_pu08 )
```

Arguments

Type	Argument	Description
fsl_u08	pOPB_pu08	Pointer to buffer containing option byte information

Return types/values

Type	Argument	Description
fsl_status_t		Result of the function. Possible values are: FSL_OK[1] FSL_BUSY[2] FSL_ERR_WRITE[1] FSL_ERR_PARAMETER FSL_ERR_FLOW

[1] Status check is performed internally by the firmware (internal mode)
[2] Status check is performed by the user (user mode)

Pre-conditions

Library must be initialized (please refer to 5.3.1.1, “FSL_Init” for details) and copied if necessary (please refer to 5.3.1.2, “FSL_CopySections” for details). In user mode the Flash environment must be additionally activated (please refer to 5.3.1.3, “FSL_FlashEnv_Activate” for details).

Post-conditions

In case of user mode call status check (see 5.3.2.8, “FSL_StatusCheck”) till function return value is different from FSL_BUSY.

Example

```
/* Write option byte */
fsl_status_t  status_enu;
fsl_u08 id_u08[4] =  { 0xFF, 0xFF, 0xFF, 0xFF };

status_enu = FSL_SetOPB( & data_u08[0] );
while( status_enu == FSL_BUSY )
{
       status_enu = FSL_StatusCheck( );
}

/* Error treatment */
...
```

Chapter 6 FSL Implementation into the user application

6.1 First steps

It is very important to have theoretic background about the Code Flash and the FSL in order to successfully implement the library into the user application. Therefore it is important to read this user manual in advance. The best way after initial reading of the user manual will be testing the FSL application sample.

6.1.1 Application sample

After a first compile run, it will be worth playing around with the library in the debugger. By that you will get a feeling for the source code files and the working mechanism of the library.

Note:

Before the first compile run, the compiler path must be configured in the application sample file “makefile”:

Set the variable COMPILER_INSTALL_DIR to the correct compiler directory

Later on, the sample might be reconfigured to use the internal mode to get a feeling of the CPU load and execution time during different modes.

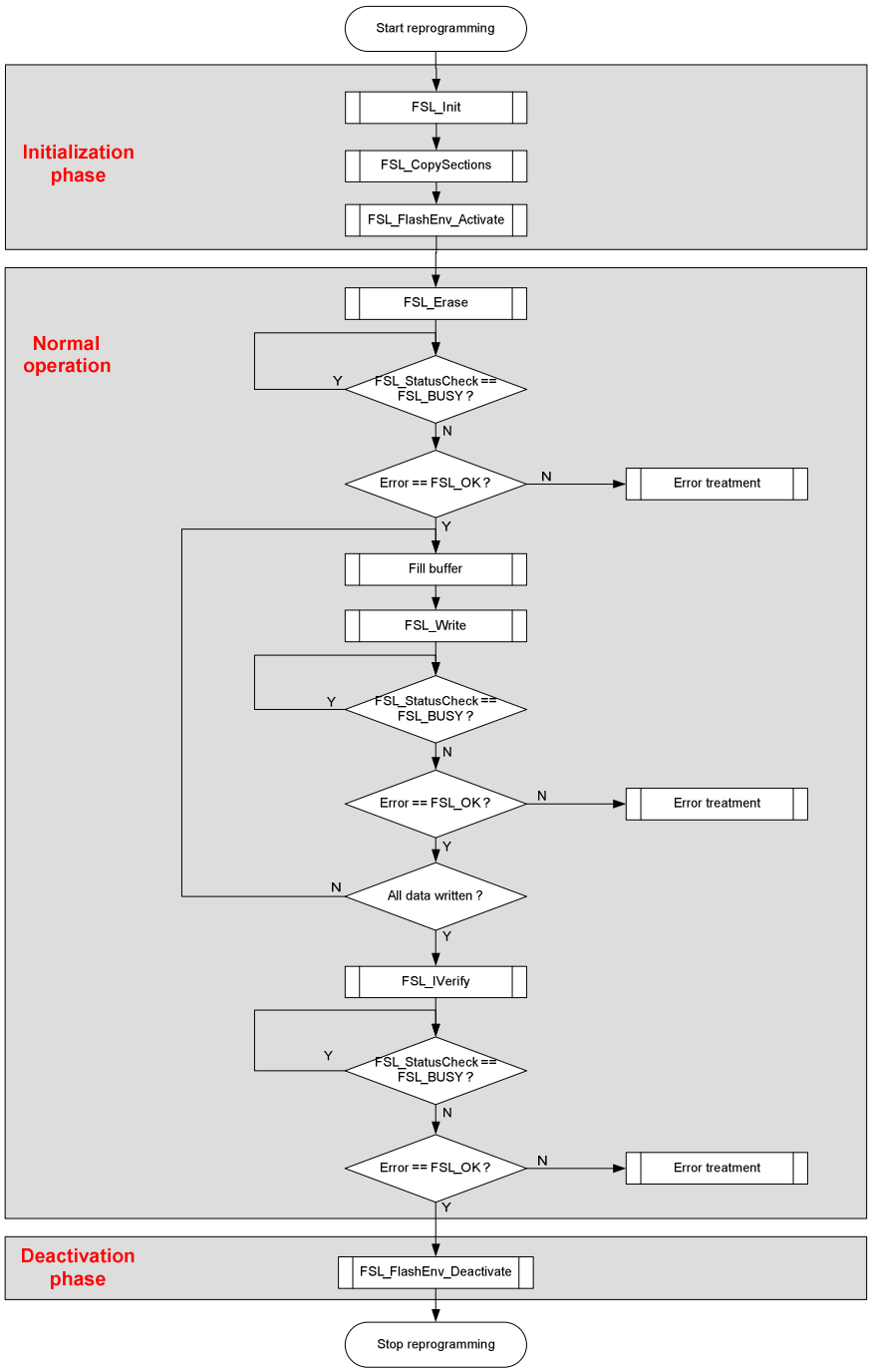
After this exercise it might be easier to understand and follow the recommendations and considerations of this document

6.2 FSL life cycle

The following flow chart represents typical FSL life cycles during device operation including the API functions to be used.

6.2.1 Device reprogramming in user mode using FSL_Erase and FSL_Write

Figure 11 Reprogramming flow - user mode using FSL_Erase and FSL_Write

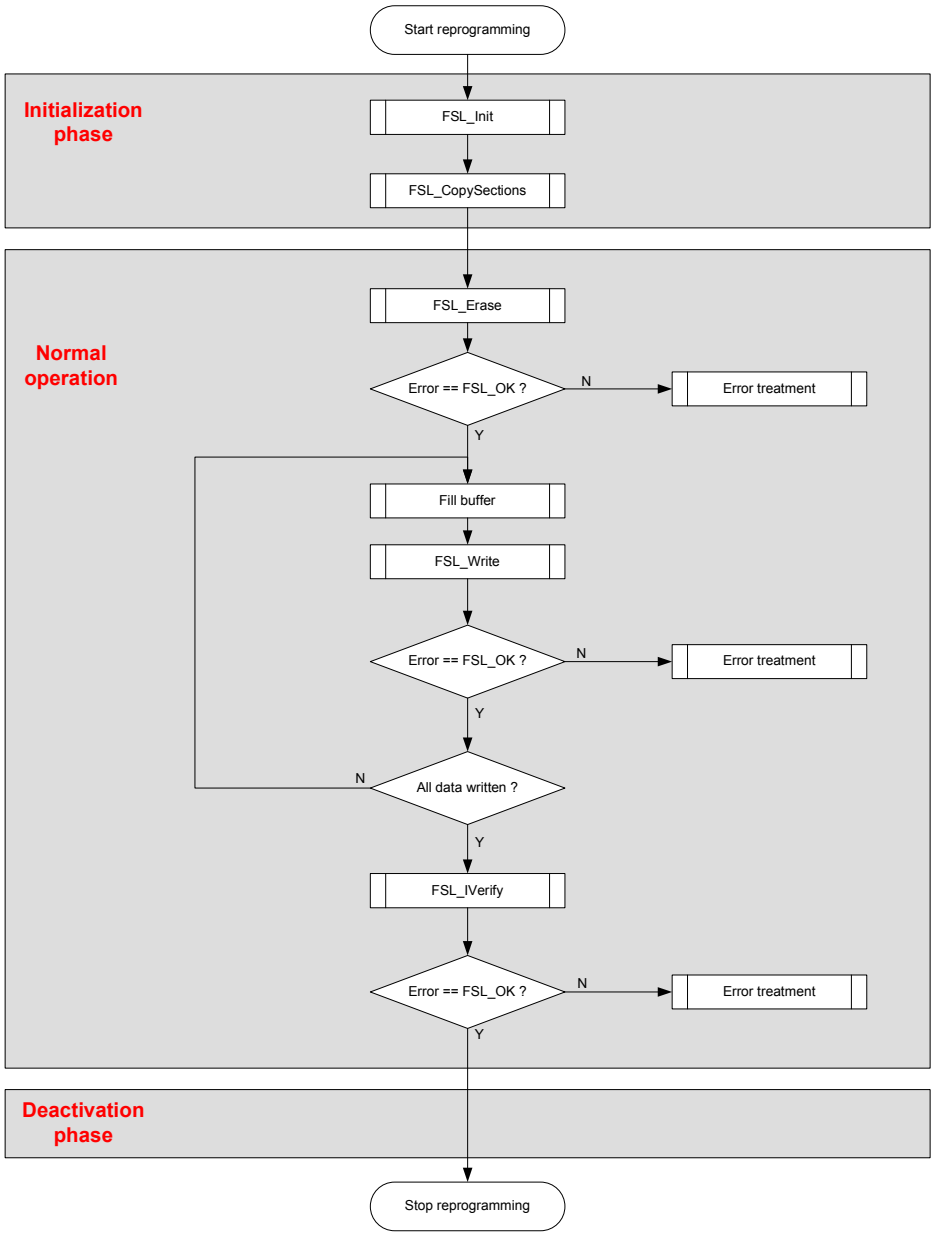


Note:

Error treatment of the FSL function themselves are not detailed described in the flow charts for simplification of the flow charts.

6.2.2 Device reprogramming in internal mode

Figure 12 Reprogramming flow – internal mode

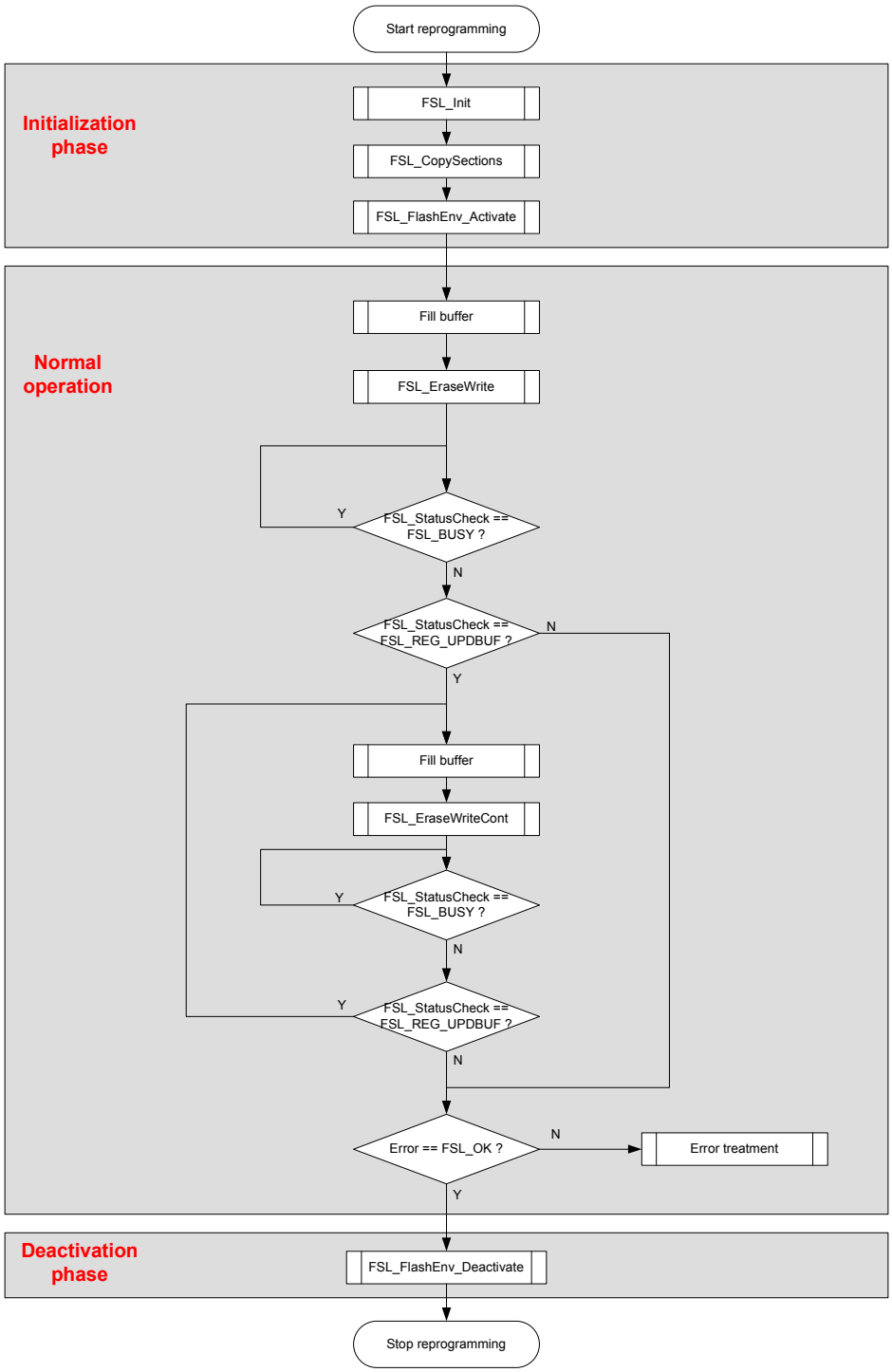


Note:

Error treatment of the FSL function themselves are not detailed described in the flow charts for simplification of the flow charts.

6.2.3 Device reprogramming in user mode using FSL_EraseWrite

Figure 13 Reprogramming flow - user mode, FSL_EraseWrite command



Note:

Error treatment of the FSL function themselves are not detailed described in the flow charts for simplification of the flow charts.

6.3 Special considerations

6.3.1 Library handling by the user application

6.3.1.1 Function re-entrancy

All functions are not reentrant. So, reentrant calls of any FSL functions must be avoided

6.3.1.2 Entering power safe mode

Entering power safe modes is prohibited during self-programming.

6.3.1.3 Code Flash access during self-programming

Code Flash accesses during an active Self-Programming Environment are not possible at all. The user application needs to be executed from other memory during that time. Please refer to chapter 4.4, “User code execution during self-programming” for further information.