NEC

Application Note

78K0R/Kx3

16-Bit Single-Chip Microcontrollers

Flash Memory Self Programming

Document No. U19236EE1V0AN00
Date published April 2008

Printed in Germany

Legal Notes

- **The information contained in this document is being issued in advance of the production cycle for the product. The parameters for the product may change before final production or NEC Electronics Corporation, at its own discretion, may withdraw the product prior to its production.**

- 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 products 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).

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives anddistributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and otherlegal issues may also vary from country to country.

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668, Japan
Tel: 044 4355111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554,
U.S.A.
Tel: 408 5886000
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211 65030
http://www.eu.necel.com/

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908 691133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01 30675800

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 6387200

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02 667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven,
The Netherlands
Tel: 040 2654010

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27
ZhiChunLu Haidian District,
Beijing 100083, P.R.China
Tel: 010 82351155
http://www.cn.necel.com/

NEC Electronics Shanghai Ltd.
Room 2511-2512, Bank of China
Tower,
200 Yincheng Road Central,
Pudong New Area,
Shanghai 200120, P.R. China
Tel: 021 58885400
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
12/F., Cityplaza 4,
12 Taikoo Wan Road, Hong Kong
Tel: 2886 9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R.O.C.
Tel: 02 27192377

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253 8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku, Seoul,
135-080, Korea Tel: 02-558-3737
http://www.kr.necel.com/

Table of Contents

Chapter 1 General-Information 6
1.1 Overview 6
1.2 Work Flow 8
1.3 Memory organization 10
1.4 Library processing time 11
Chapter 2 Programming environment 12
2.1 Hardware environment 12
2.2 Software environment 12
2.2.1 Stack and data-buffer 14
Chapter 3 Interrupt servicing 15
3.1 Interrupt response time and suspension delay 20
3.2 Restrictions during interrupt servicing 21
Chapter 4 Boot-swapping 22
Chapter 5 Library for NEC Compiler 26
Chapter 6 Library for IAR Compiler 27
6.1 Library function prototypes 27
6.2 Library explanation 28
6.2.1 FSL_Open 29
6.2.2 FSL_Close 31
6.2.3 FSL_Init 32
6.2.4 FSL_Init_cont 33
6.2.5 FSL_ModeCheck 34
6.2.6 FSL_BlankCheck 35
6.2.7 FSL_Erase 36
6.2.8 FSL_IVerify 37
6.2.9 FSL_Write 38
6.2.10 FSL_EEPROMWrite 40
6.2.11 FSL_GetSecurityFlags 42
6.2.12 FSL_GetActiveBootCluster 44
6.2.13 FSL_GetBlockEndAddress 45
6.2.14 FSL_GetFlashShieldWindow 47
6.2.15 FSL_SetFlashShieldWindow 49
6.2.16 FSL_SetXXX and FSL_InvertBootFlag 51
6.2.17 FSL_SwapBootCluster 53
6.2.18 FSL_ForceReset 54
6.2.19 FSL_SetInterruptMode 55
6.3 Sample linker file 56
6.4 How to integrate the library into an application 58
Chapter 7 Sample code 59

Chapter 1 General-Information

1.1 Overview

The 78K0R/Kx3 series products are equipped with an internal firmware, which allows to rewrite the flash memory without the use of an external programmer. In addition to this internal firmware NEC provides the socalled 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.

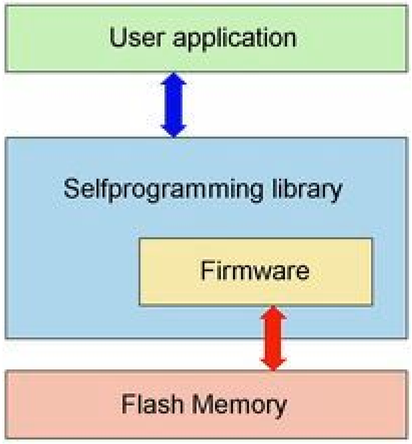


Figure 1-1 Flash Access

Caution

- In the 78K0R/Kx3 series products, 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.
- The self programming library uses the CPU (register bank 3). Use of some RAM areas are prohibited when using the self-programming. For detailed information please refer to the device Users Manual.

Operation Modes There are three operation modes during self-programming.

Mode	Description
Normal Mode	- execute user application - after RESET operation starts in this mode
Mode A1	- After FSL_XXX function call
Mode A2	- used by the firmware only to perform the command - not visible to the user

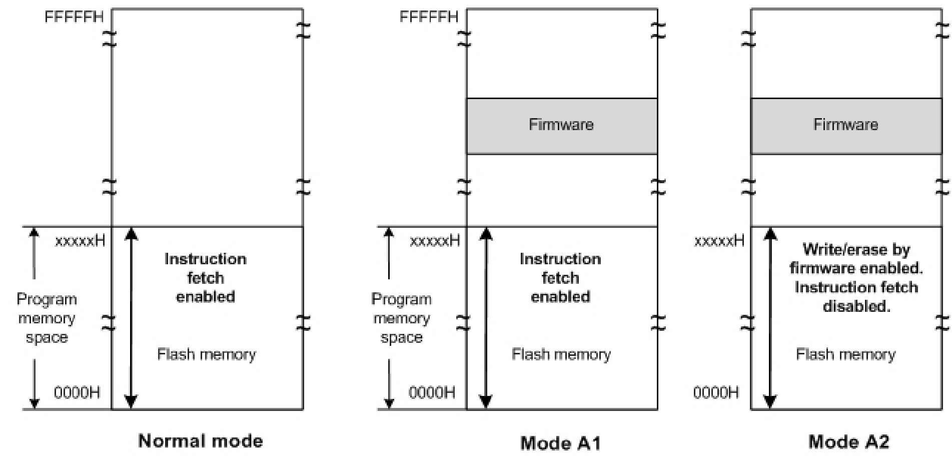


Figure 1-2 Operation Modes

1.2 Work Flow

The self-programming library can be used by an user program written in either C- or assembly language.

The following flowchart illustrates a sample procedure of rewriting the flash memory by using the self programming library.

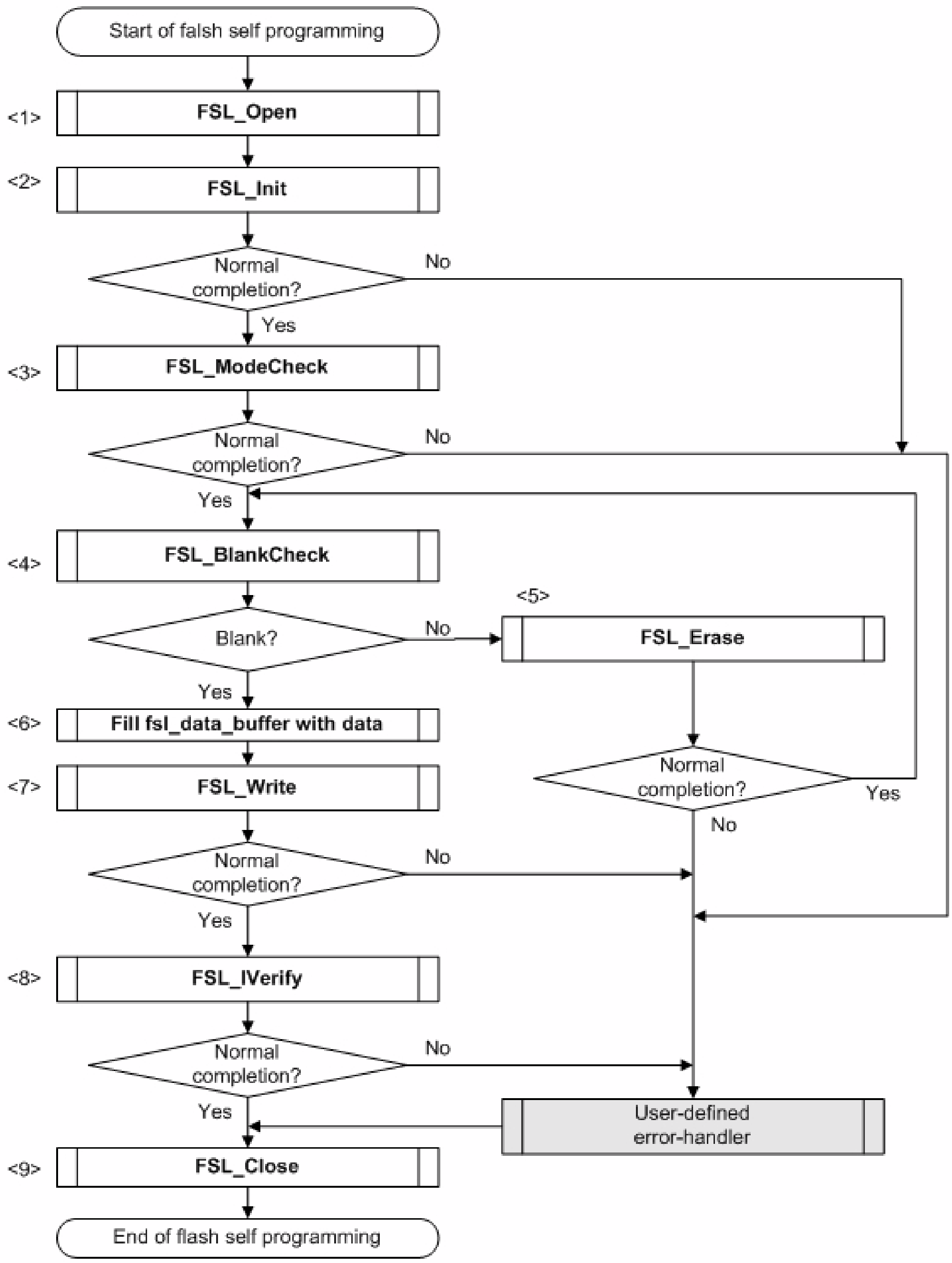


Figure 1-3 Flow of self-programming (rewriting contents of flash memory)

Flow Explanation

1. Call the function **FSL_Open**.
   - Preservation and configuration of the interrupt controller for self-programming. (optional)
   - Set FLMD0 to HIGH.
   - any other customizable preparation measures(i.e. activation of the communication channel)
2. Call the function **FSL_Init** to initialize the self-programming environment.
3. Call the mode check function **FSL_ModeCheck** to examine the FLMD0 voltage level.
4. Call the block blank check function **FSL_BlankCheck** to prove if the specified block is blank.
5. Call the block erase function **FSL_Erase** to erase the data of a specified block.
6. Fill the data buffer with data has to be written into the flash.
7. Call the word write function **FSL_Write** to update 1 to 64 words (each word equals 4 bytes) of data to a specified address.
8. Call the block verify function **FSL_IVerify** to verify a specified block (internal verification).
9. Postprocessing, call the close function **FSL_Close**.
   - Set FLMD0 is LOW.
   - Retrieve preserved interrupt masks. (optional)
   - any other customizable post-processing measures(i.e. deactivation of the communication channel)

1.3 Memory organization

The flash memory of all 78K0R/Kx3 devices is divided into blocks of 2KByte. Each block can be erased/verified and blankchecked individually. The following table shows the start- and end-addresses of each block.

Bl. nr	Block start address	Block end address	Bl. nr	Block start address	Block end address	Bl. nr	Block start address	Block end address
0	00000	007FF	2B	15800	15FFF	56	2B000	2B7FF
1	00800	00FFF	2C	16000	167FF	57	2B800	2BFFF
2	01000	017FF	2D	16800	16FFF	58	2C000	2C7FF
3	01800	01FFF	2E	17000	177FF	59	2C800	2CFFF
4	02000	027FF	2F	17800	17FFF	5A	2D000	2D7FF
5	02800	02FFF	30	18000	187FF	5B	2D800	2DFFF
6	03000	037FF	31	18800	18FFF	5C	2E000	2E7FF
7	03800	03FFF	32	19000	197FF	5D	2E800	2EFFF
8	04000	047FF	33	19800	19FFF	5E	2F000	2F7FF
9	04800	04FFF	34	1A000	1A7FF	5F	2F800	2FFFF
0A	05000	057FF	35	1A800	1AFFF	60	30000	307FF
0B	05800	05FFF	36	1B000	1B7FF	61	30800	30FFF
0C	06000	067FF	37	1B800	1BFFF	62	31000	317FF
0D	06800	06FFF	38	1C000	1C7FF	63	31800	31FFF
0E	07000	077FF	39	1C800	1CFFF	64	32000	327FF
0F	07800	07FFF	3A	1D000	1D7FF	65	32800	32FFF
10	08000	087FF	3B	1D800	1DFFF	66	33000	337FF
11	08800	08FFF	3C	1E000	1E7FF	67	33800	33FFF
12	09000	097FF	3D	1E800	1EFFF	68	34000	347FF
13	09800	09FFF	3E	1F000	1F7FF	69	34800	34FFF
14	0A000	0A7FF	3F	1F800	1FFFF	6A	35000	357FF
15	0A800	0AFFF	40	20000	207FF	6B	35800	35FFF
16	0B000	0B7FF	41	20800	20FFF	6C	36000	367FF
17	0B800	0BFFF	42	21000	217FF	6D	36800	36FFF
18	0C000	0C7FF	43	21800	21FFF	6E	37000	377FF
19	0C800	0CFFF	44	22000	227FF	6F	37800	37FFF
1A	0D000	0D7FF	45	22800	22FFF	70	38000	387FF
1B	0D800	0DFFF	46	23000	237FF	71	38800	38FFF
1C	0E000	0E7FF	47	23800	23FFF	72	39000	397FF
1D	0E800	0EFFF	48	24000	247FF	73	39800	39FFF
1E	0F000	0F7FF	49	24800	24FFF	74	3A000	3A7FF
1F	0F800	0FFFF	4A	25000	257FF	75	3A800	3AFFF
20	10000	107FF	4B	25800	25FFF	76	3B000	3B7FF
21	10800	10FFF	4C	26000	267FF	77	3B800	3BFFF
22	11000	117FF	4D	26800	26FFF	78	3C000	3C7FF
23	11800	11FFF	4E	27000	277FF	79	3C800	3CFFF
24	12000	127FF	4F	27800	27FFF	7A	3D000	3D7FF
25	12800	12FFF	50	28000	287FF	7B	3D800	3DFFF
26	13000	137FF	51	28800	28FFF	7C	3E000	3E7FF
27	13800	13FFF	52	29000	297FF	7D	3E800	3EFFF
28	14000	147FF	53	29800	29FFF	7E	3F000	3F7FF
29	14800	14FFF	54	2A000	2A7FF	7F	3F800	3FFFF
2A	15000	157FF	55	2A800	2AFFF			

1.4 Library processing time

The following figure illustrates the processing time of each library function.

Table 1-1 Processing Time

Function name	Processing Time (Unit: Microseconds)	
	Min	Max
FSL_Init	$31999/f_{CLK}$ + 65	$31999/f_{CLK}$ + 65
FSL_Init_cont	$1099/f_{CLK}$ + 40	$26799/f_{CLK}$ + 45
FSL_Mode Check	$11/f_{CLK}$	$13/f_{CLK}$
FSL_Blank Check	$97798/f_{CLK}$ + 55	$97798/f_{CLK}$ + 55
FSL_Erase	$113223/f_{CLK}$ + 10017	$2036693/f_{CLK}$ + 221085
FSL_IVerify	$201595/f_{CLK}$ + 5200	$201595/f_{CLK}$ + 5200
FSL_Write	(22735 + 500 x W) $/f_{CLK}$ + 65 + 130 x W	(22935+ 3100 x W) $/f_{CLK}$ + 70 + 1350 x W
FSL_EEPROMWrite	(23335+ 1000 x W) $/f_{CLK}$ + 135 + 140 x W	(23435+ 3670 x W) $/f_{CLK}$ + 150 + 1340 x W
FSL_GetSecurityFlags	$1637/f_{CLK}$ + 65	$1637/f_{CLK}$ + 65
FSL_GetActiveBootCluster	$1290/f_{CLK}$	$1290/f_{CLK}$
FSL_GetBlockEndAddr	$322/f_{CLK}$	$322/f_{CLK}$
FSL_GetFlashShieldWindow	$1696/f_{CLK}$ + 60	$1896/f_{CLK}$ + 65
FSL_InvertBootFlag	$14248/f_{CLK}$ + 143	$4433205/f_{CLK}$ + 448195
FSL_SetFlashShieldWindow	$11552/f_{CLK}$ + 83	$4430309/f_{CLK}$ + 448135
FSL_SetChipEraseProtectFlag	$13248/f_{CLK}$ + 143	$4432205/f_{CLK}$ + 448195
FSL_SetBlockEraseProtectFlag		
FSL_SetWriteProtectFlag		
FSL_SetBootClusterProtectFlag		
FSL_SwapBootCluster	$600/f_{CLK}$	$600/f_{CLK}$
FSL_ForceReset	$1/f_{CLK}$	$1/f_{CLK}$
FSL_SetInterruptMode	$70/f_{CLK}$	$70/f_{CLK}$

1. f_{CLK} : CPU frequency
2. W : word count (1 word == 4 bytes)

Caution: The values shown in the table above are estimated values, therefore there is no warranty. Please refer to the device user's manual for detailed timing information.

Chapter 2 Programming environment

This chapter explains the necessary hardware and software environment which is used to rewrite flash memory by using the self-programming library.

2.1 Hardware environment

In the 78K0R/Kx3 series devices, there is a FLMD0 pin controlling flash memory operation mode. To protect the flash memory against unwanted overwriting during normal operation the FLMD0 pin has to be set to LOW level at that time. To be able to update flash memory content the FLMD0 pin should be set to HIGH level.

If the FLMD0 pin is low during self-programming, the firmware can still be executed, but the circuit for rewriting flash memory does not operate. In such a case the self-programming function returns an error code but the content of the flash remains untouched.

FLMD0 controlled via internal pull-down/up resistor

The FLMD0 level can be controlled internally via the BECTL register. When using BECTL for FLMD0 level control, leaving the FLMD0 pin open is recommended.

There are two predefined macros(FSL_FLMD0_LOW and FSL_FLMD0_HIGH) using the BECTL register, which can be found in the **fsl_user.h**.

The self programming open function FSL_Open can switch the FLMD0 pin to high or low, by changing the value of BECTL register vai the macros.

The following is an example circuit that allows to control the voltage level at the FLMD0 pin externally by using a dedicated general purpose I/O port pin. Please refer to the device Users Manual for detailed information.

2.2 Software environment

The self-programming library allocates its code inside the user area and consumes up to about 1002 bytes of the program memory. The self programming library itself uses CPU's register bank 3, 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.

Table 2-1 Software Resources

<table>
<tr><th rowspan="2">Item</th><th rowspan="2">Description</th><th colspan="3">Restriction depending on the implemented RAM size</th></tr>
<tr><th>RAM: 12 KByte</th><th>RAM 30 KByte</th><th>RAM other sizes</th></tr>
<tr><td>CPU</td><td>Register Bank 3</td><td colspan="3">cannot be used by the application</td></tr>
<tr><td>Work area</td><td>Entry RAM: 140 bytes
Used by firmware!</td><td>User RAM
FCF00H-FD6FFH
will be destroyed by
firmware</td><td>User RAM
F8700H-F8EFFH
will be destroyed by
firmware</td><td>User RAM not
touched</td></tr>
<tr><td>Stack</td><td>additional 75 bytes max.
Note
Use the same stack as for the user program</td><td rowspan="3">FFE20H-FFEFFH
and
FCF00H-FD6FFH
prohibited</td><td rowspan="3">FFE20H-FFEFFH
and
F8700H-F8EFFH
prohibited</td><td rowspan="3">FFE20H-FFEFFH
prohibited</td></tr>
<tr><td>Data buffer</td><td>5 to 256 bytes
Note
The size of this buffer varies depending on the writing unit specified by the user program and usage of SetInfo or not.</td></tr>
<tr><td>Self-programming data (FSL_DATA)</td><td>6 bytes internal data usage of FSL</td></tr>
<tr><td>Self-programming library</td><td>xxx-1002 bytes + user part
(9 - 87 bytes)
Note
Code size of the self-programming library varies depending on ther configuration(Please refer to the following table).</td><td colspan="3">The self-programming library must be located inside the internal flash.</td></tr>
</table>

Caution

- The self-programming operation is not guaranteed if the user manipulates the above resources. Do not manipulate these resources during a self programming session.
- 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

	IAR V4.60 (near model)	IAR V4.60 (far model)
Max. code size	949 bytes	1002 bytes
Max. code size (without GetInfo, SetInfo and FSL_SwapBootCluster)	420 bytes	471 bytes
Max. code size (without GetInfo, SetInfo and FSL_SwapBootCluster) --> *FSL_InvertBootFlag and FSL_GetActiveBootCluster included*	731 bytes	786 bytes

Note *** The IAR-Linker excludes this functions automatically, if they are not referenced by the application.

2.2.1 Stack and data-buffer

Stack The stack is used to store data and instruction pointers during self-programming. Please refer to the table above "Software Resources" for the location restrictions of the stack during self-programming.

Data Buffer The data buffer is used for data-exchange between the firmware and the self-programming library.

Note Data to be written to the flash memory must be appropriately set and processed before the word write/SetInfo function is called. The length of the data buffer depends on the user configuration as shown below.

- **min. 50 bytes**: if function FSL_InvertBootFlag or FSL SwapBootCluster is used
- **min. 6 bytes:** if function FSL_InvertBootFlag and FSL SwapBootCluster are not used

Chapter 3 Interrupt servicing

Some FSL functions can be interrupted by an interrupt during the execution. The non-masked interrupts will be checked, whether an interrupt was generated. The following table list the functions, which supports interrupt acknowledgement.

Function name	Interrupt Acknowledgement
FSL_Open	Acknowledged
FSL_Close	
FSL_Init	
FSL_Init_cont	
FSL_ModeCheck	
FSL_BlankCheck	
FSL_Erase	
FSL_IVerify	
FSL_Write	
FSL_EEPROMWrite	
FSL_GetSecurityFlags	Not acknowledged
FSL_GetActiveBootCluster	
FSL_GetBlockEndAddr	
FSL_GetFlashShieldWindow	
FSL_InvertBootFlag	
FSL_SetChipEraseProtectFlag	
FSL_SetBlockEraseProtectFlag	
FSL_SetWriteProtectFlag	
FSL_SetBootClusterProtectFlag	
FSL_SetFlashShieldWindow	
FSL_SwapBootCluster	
FSL_ForceReset	
FSL_SetInterruptMode	

Self-programming without interrupt processing

The following figure illustrates the processing flow without interrupts.

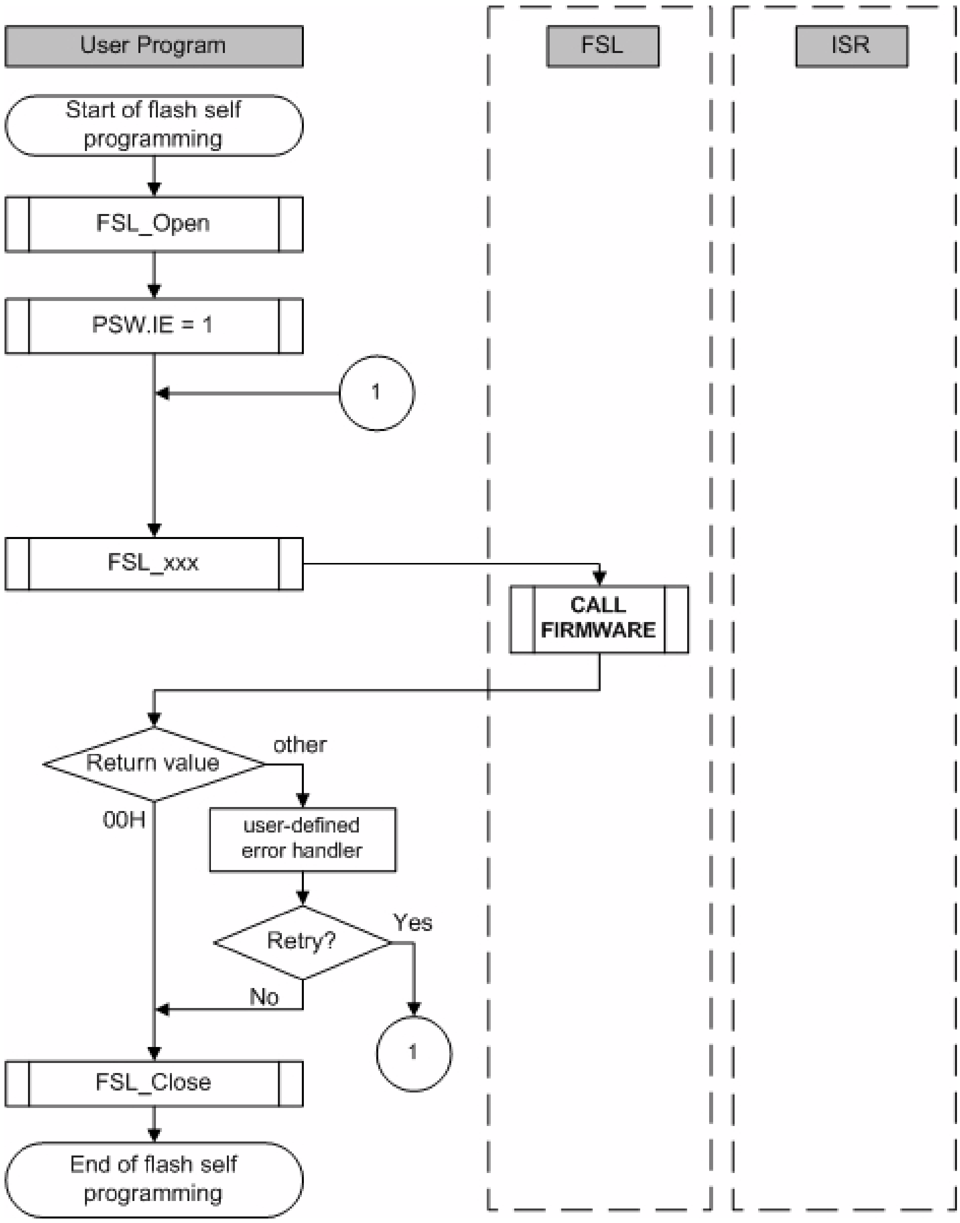


Figure 3-1 Flow of Processing without Interrupt

As shown in the figure above the PSW.IE bit must be cleared for execution without interrupts.

Interrupt handling

Interrupts will be handled in two different ways. If the FSL function was interrupted, the user has a possibility to make a decision (inside ISR), whether to leave the FSL function with 0x1F return value or to continue until it is finished.

Self-programming with interrupt processing only

The following figure illustrates an interrupted FSL function where the ISR decides to continue the function.

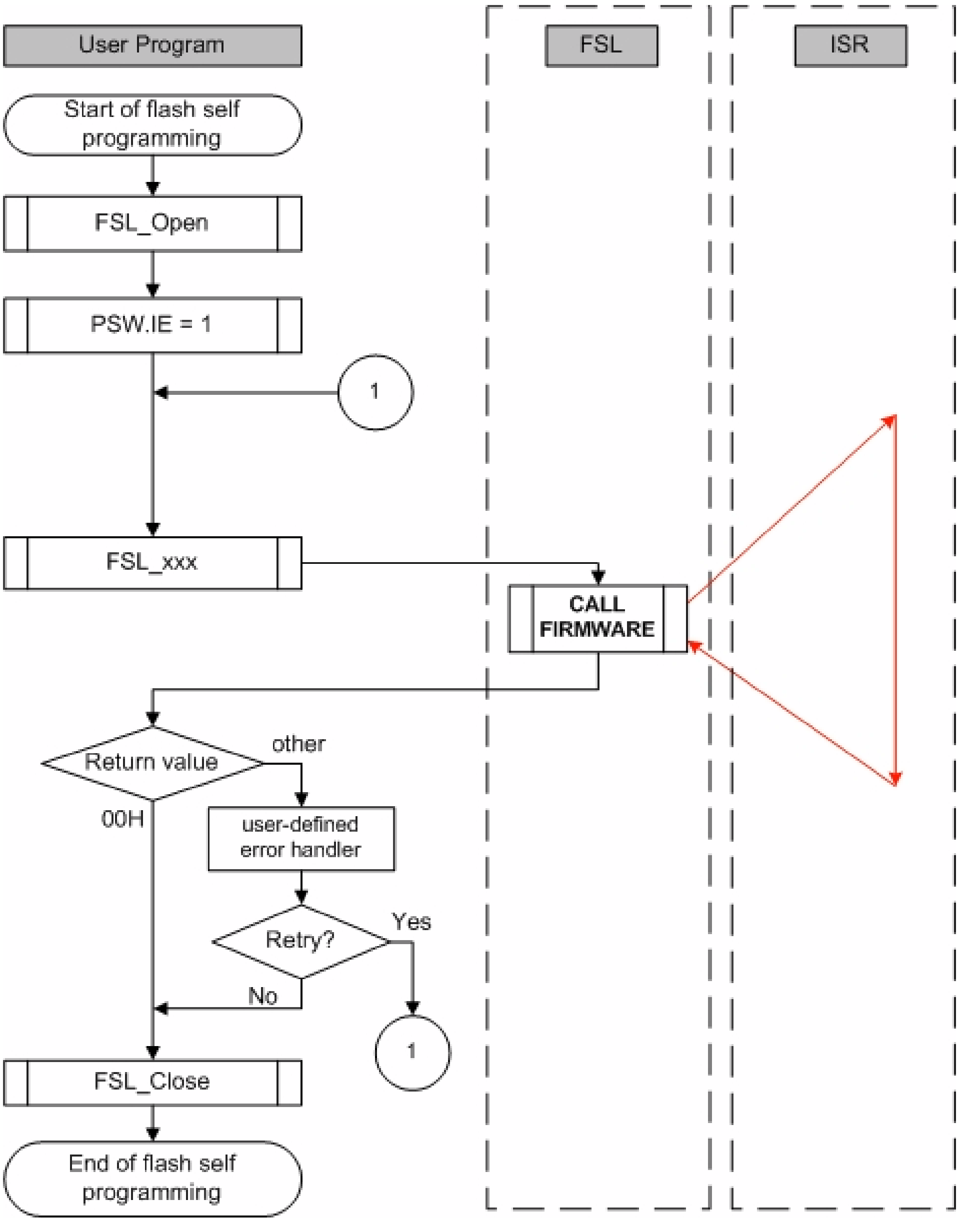


Figure 3-2 Flow of Processing in Case of Interrupt (Mode 0)

As you can see in the figure above, the FSL function will be interrupted by a non-masked interrupt and the ISR will be processed. After ISR processing the FSL will continue the function and will not return to the user application with 0x1F. The other case is, if the user wants to leave the FSL_XXX function as fast as possible. In that case the function FSL_SetInterruptMode must be called inside the ISR. After ISR processing the function will leave the function with 0x1F interrupted status.

Self-programming with interrupt processing followed by subsequent command suspension

The following figure illustrates an interrupted FSL function where the ISR decides to leave the function.

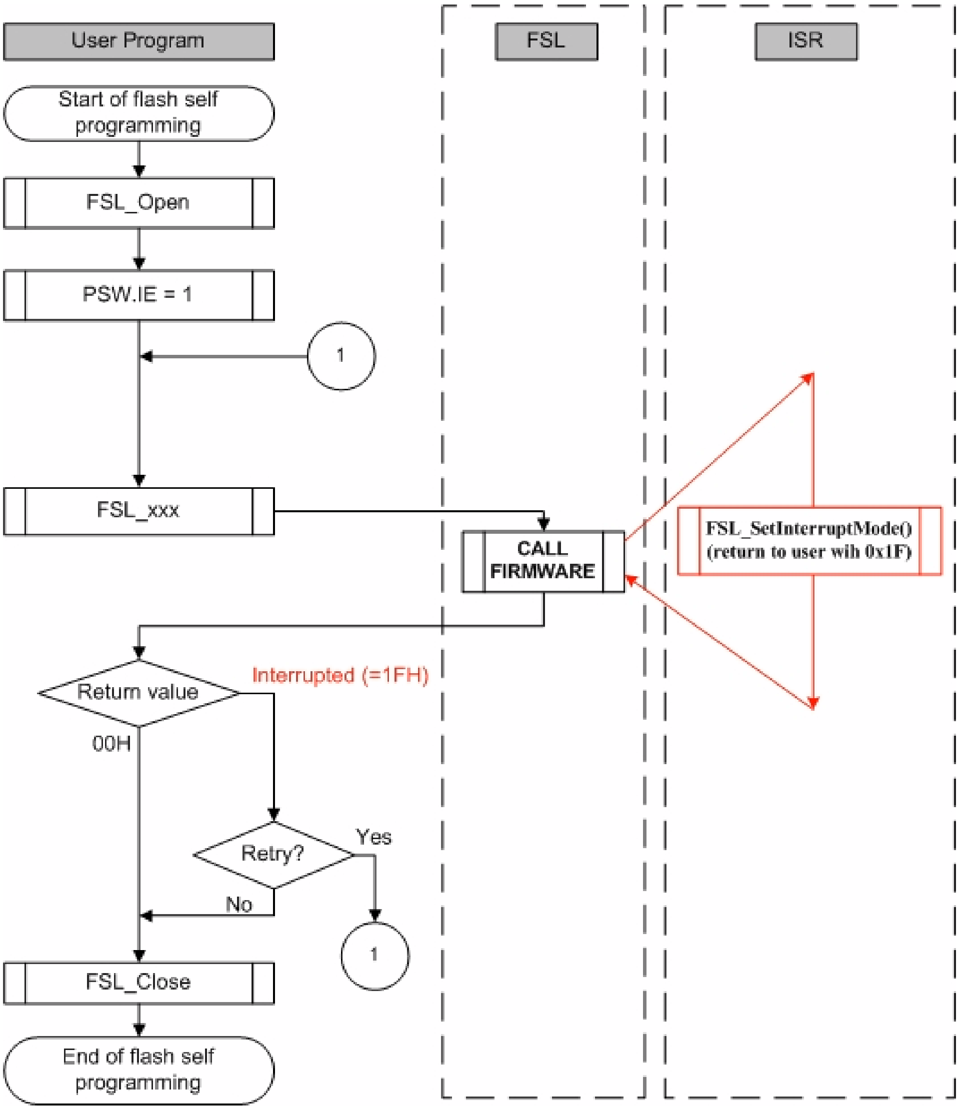


Figure 3-3 Flow of Processing in Case of Interrupt (Mode 1)

In this case, user application should recall the function to resume the processing until the FSL function is finished.

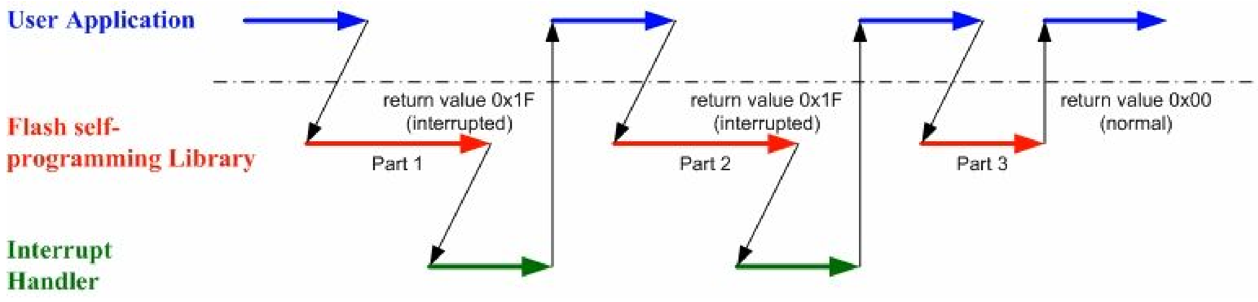


Figure 3-4 FSL Function Process with Resuming Mechanism

The following code-sample shows a suggestion on how to handle interruptions.

```
 do
{
   my_status_u08 =  FSL_BlankCheck (block_u16);

   // in case of FSL_ERR_INTERRUPTION is returned here,
   // the corresponding ISR is already executed !!

} while (my_status_u08 == FSL_ERR_INTERRUPTION);
```

The following table shows how to resume (continue) self-programming commands interrupted and suspended by an interrupt service. The most of them are continued by re-calling the same function with unchanged parameters as long the function returns the value 0x1F. Exception is the self-programming initialization that requires a different function to be continued. Please refer to the table below for details:

Table 3-1 Resume/Restart process for interrupted self-programming functions

Function name	Resume method
FSL_Init	Call FSL_Init_cont (**not FSL_Init**) when it returns the status 0x1F (FSL_ERR_INTERRUPTION)
FSL_Init_cont	Re-call FSL_Init_cont as long it returns the status 0x1F (FSL_ERR_INTERRUPTION)
FSL_BlankCheck	Re-call FSL_BlankCheck(..) as long it returns the status 0x1F (FSL_ERR_INTERRUPTION)
FSL_Erase	Re-call FSL_Erase(..) as long it returns the status 0x1F (FSL_ERR_INTERRUPTION)
FSL_Write	Re-call FSL_Write(..) as long it returns the status 0x1F (FSL_ERR_INTERRUPTION)
FSL_IVerify	Re-call FSL_IVerify(..) as long it returns the status 0x1F (FSL_ERR_INTERRUPTION)
FSL_EEPROMWrite	Re-call FSL_EEPROMWrite(..) as long it returns the status 0x1F (FSL_ERR_INTERRUPTION)

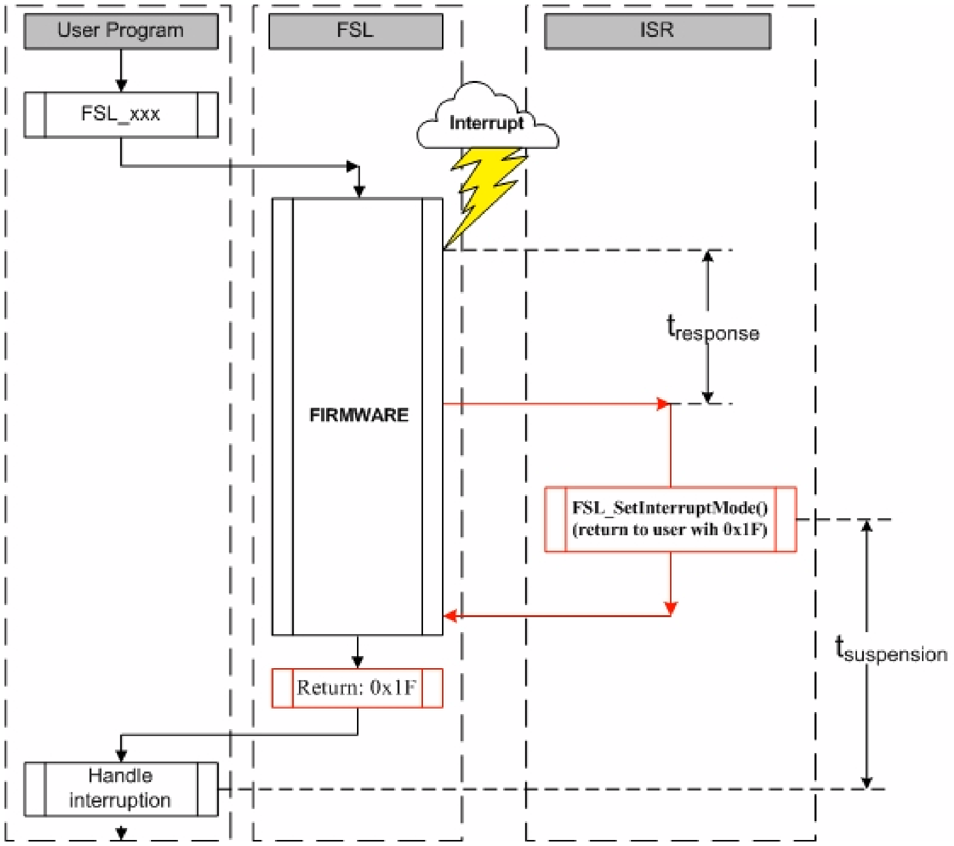
3.1 Interrupt response time and suspension delay

Unlike the case for an ordinary interrupt, an interrupt generated during self-programming is handled via post-interrupt servicing in the firmware (i.e. setting 0x1F as return value of a self-programming function). Consequently, the response time is longer than that of an ordinary interrupt.

There are to different cases regarding the interrupt response time:

1. Interrupt response time from the occured interrupt to interrupt servicing.
2. The time where the user call the function FSL_SetInterruptMode inside the ISR till return to the application with 0x1F status.

The following figure illustrates the two cases:



The following tables illustrates the interrupt response time values for above described cases.

Table 3-2 Interrupt response time and suspension delay

Function name	Interrupt response time and suspension delay			
	$t_{response}$: Time from interrupt to ISR		$t_{suspension}$: Time from FSL_SetInterruptMode call to user application	
	Max (unit in µs)	f_{CLK} = 20MHz	Max (unit in µs)	f_{CLK} = 20MHz
FSL_Init	1000/f_{CLK} + 65	115 µs	5332/f_{CLK} (f_{CLK} > 4MHz) 4532/f_{CLK} (f_{CLK} < 4MHz)	267 µs(f_{CLK} > 4MHz)
FSL_Init_cont	1000/f_{CLK} + 65	115 µs	1332/f_{CLK} + 65	132 µs
FSL_BlankCheck	1370/f_{CLK} + 65	83 µs	103134/f_{CLK} + 13.2	5.2 ms
FSL_Erase	3128/f_{CLK} + 73	230 µs	2126240/f_{CLK} + 231614	338 ms
FSL_Write	2382/f_{CLK} + 60	180 µs	(1616 + 5558 x W) /f_{CLK} + 17.6 + 413 x W	(98.4 + 690.4 x W) µs
FSL_IVerify	1493/f_{CLK} + 18	93 µs	225728/f_{CLK} + 2378.2	13.7 ms
FSL_EEPROMWrite	2808/f_{CLK} + 60	201 µs	(2264 + 6186 x W) /f_{CLK} + 43+ 418 x W	(156.1 + 727.3 x W) µs

Caution: The values shown in the tabel above are estimated values, therefore there is no warranty.

3.2 Restrictions during interrupt servicing

The following described restrictions are related to interrupt servicing during self-programming.

- If processing related to self-programming is performed or a setting related to it is changed during processing of an interrupt that has occurred during execution of self-programming, then the operation is not guaranteed. Do not perform processing related to self-programming and change settings related to it during interrupt servicing.
- Do not use register bank 3 during interrupt servicing, because self-programming uses register bank 3.
- Save and restore registers used for interrupt servicing during interrupt servicing.
- Do not execute any other self-programming library function as long the currently executed but suspended function returns the status 0x1F. The only one exception is the function FSL_Init() that can be called at any time.
- Do not change any parameter of the self-programming library function (address, block-number, ...) being executed as long its returned status is 0x1F.
- Do not erase RAM areas used by self-programming. Please refer to the chapter "software environment" for detailed information.
- The data buffer used by the FSL_Init, FSL_Write/ FSL_EEPROMWrite, FSL_GetXXX, FSL_SetXXX and FSL_SwapBootCluster functions does not require separate areas to be secured; therefore the same area can be shared by user application.

Chapter 4 Boot-swapping

Reason for Bootswapping

A permanent data loss may occur when rewritting the vector table, the basic functions of the program, or the self-programming area, due to one of the following reasons:

- a temporary power failure
- an externally generated reset

The user program is thus not able to be restarted through reset. Likewise the rewrite process can no longer be performed. This potential risk can be avoided by using a boot swap functionality.

Boot swap Function

The boot swap function FSL_InvertBootFlag replaces the current boot area, boot cluster 0[Note], with the boot swap target area, boot cluster 1[Note].

Before swapping, user program should write the new boot program into boot cluster 1. And then swap the two boot cluster and force a hardware 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

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

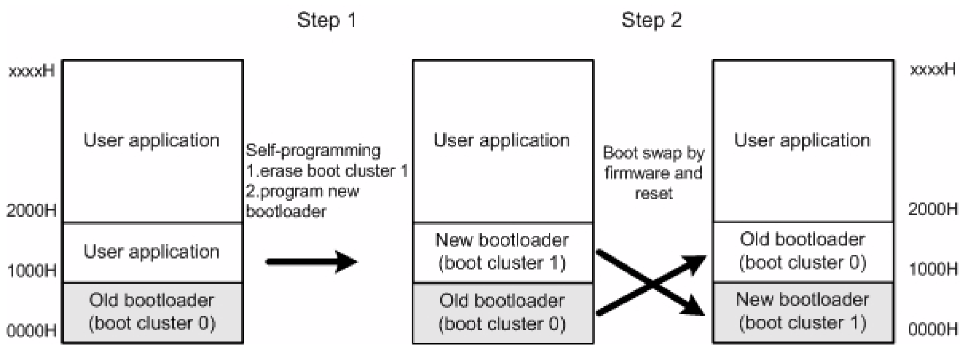


Figure 4-1 Summary of Boot Swapping Flow

Caution

To rewrite the flash memory by using a programmer (such as the PG-FP5) after boot swapping, follow the procedure below.

1. **Chip erase**
2. **PV (program, verify) or EPV (erase, program, and verify)**

(Unless step 1 is performed, data may not be correctly written.)

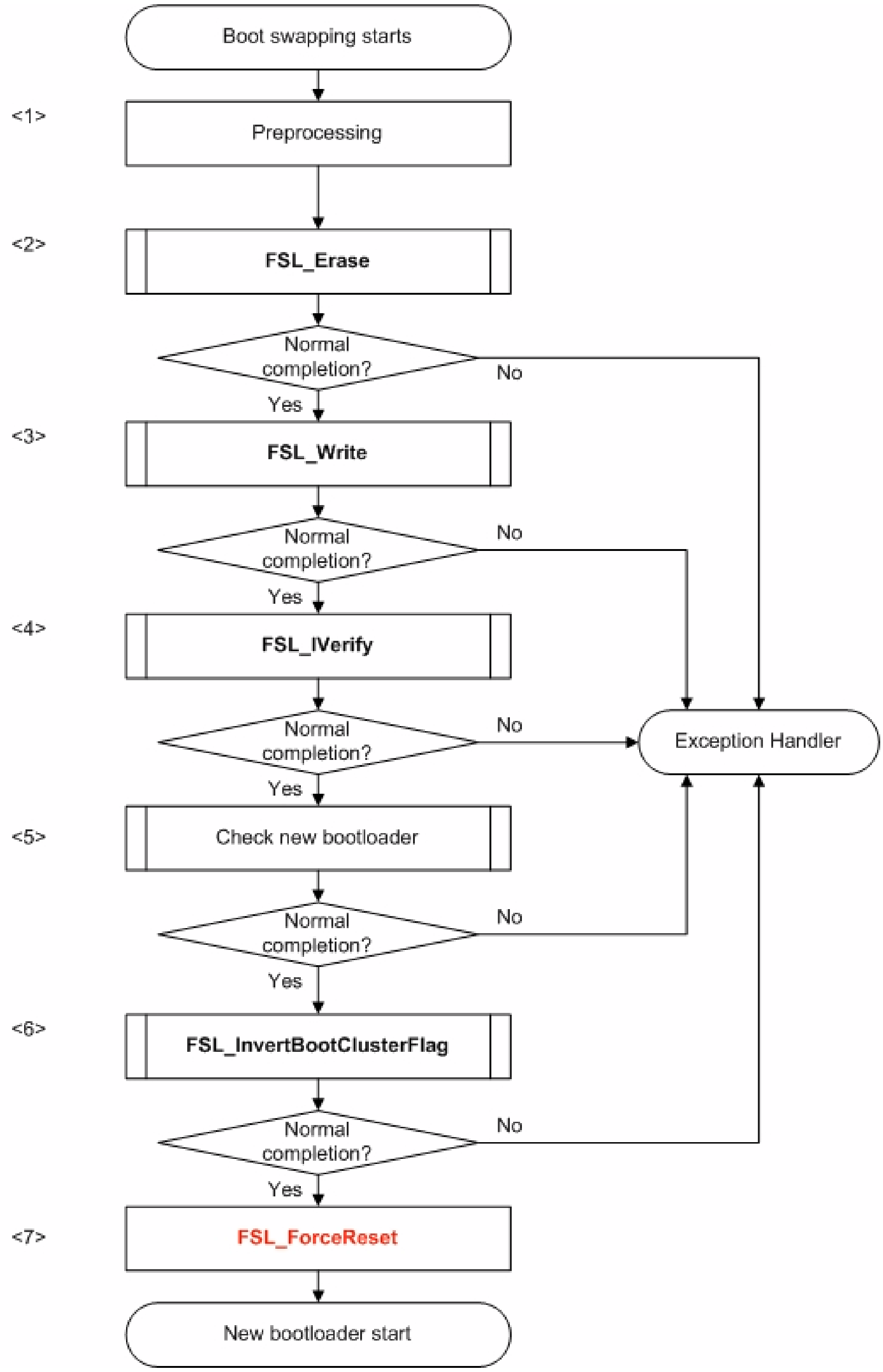


Figure 4-2 Flow of Boot Swapping

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
- Check FLMD0 voltage level

<2> Erasing blocks 2 to 3

Call the erase function FSL_Erase to erase blocks 2 to 3.

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

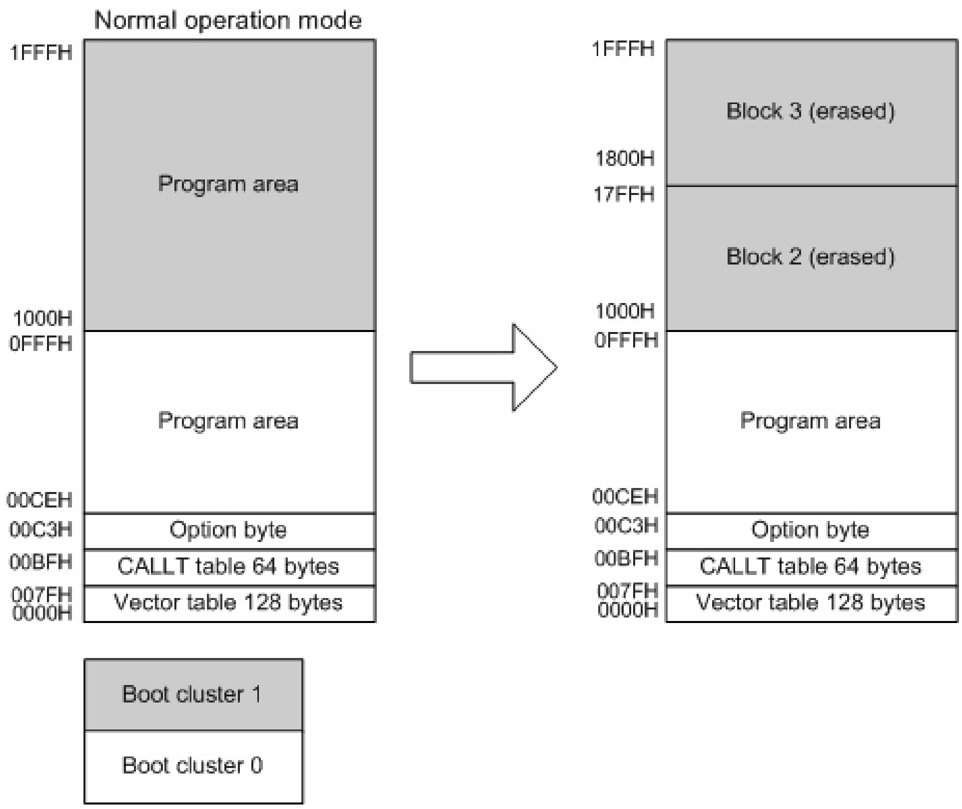


Figure 4-3 Erasing Boot Cluster 1

<3> Writing new program to boot cluster 1

Use the FSL_Write function to write the new bootloader (1000H to 1FFFH).

Note The write function writes data in word units (256 bytes max.).

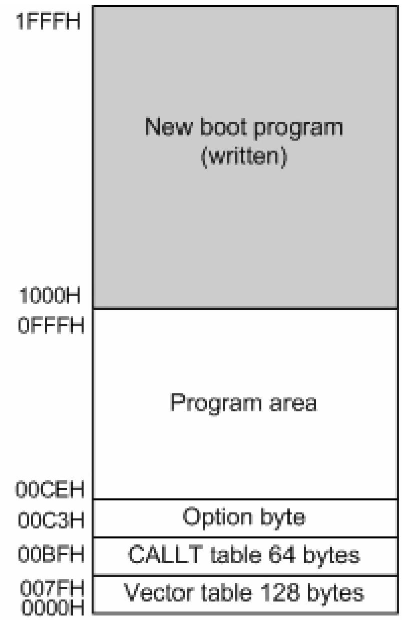


Figure 4-4 Writing New Program to Boot Cluster 1

<4> Verifying Blocks 2 to 3

Call the verify function FSL_IVerify to verify Blocks 2 to 3.

Note The verify function verifies only a 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 5 Library for NEC Compiler

This chapter contains the details on the self-programming library for the NEC Compiler.

Note: These library is currently not implemented.

Chapter 6 Library for IAR Compiler

This chapter describes the details on the self-programming library for the IAR Compiler (Version V4.XX). The library will be delivered in pre-compiled form for different data models (far model and near model).

- fsl_near.r26 : near data model
- fsl_far.r26 : far data model

Note: These libraries are independent from the code model.

6.1 Library function prototypes

The flash self-programming library consists of the following functions.

Table 6-1 Self-programming Library - function prototypes

Function prototype	Outline
void FSL_Open(void)	Opens a flash self programming session.
void FSL_Close(void)	Closes a flash self programming session.
fsl_u08 FSL_Init(fsl_u08* data_buffer_pu08)	Initialization of the self-programming environment.
fsl_u08 FSL_Init_cont(fsl_u08* data_buffer_pu08)	Continue initialization of the entry RAM after interrupted FSL_Init function.
fsl_u08 FSL_ModeCheck(void)	Checks FLMD0 voltage level.
fsl_u08 FSL_BlankCheck(fsl_u16 block_u16)	Checks if specified block is empty.
fsl_u08 FSL_Erase(fsl_u16 block_u16)	Erases a specified block.
fsl_u08 FSL_IVerify(fsl_u16 block_u16)	Verifies a specified block (internal verification).
fsl_u08 FSL_Write(fsl_u32 s_address_u32, fsl_u08 word_count_u08)	Writes up to 64 words (each word equals 4 bytes) to a specified address.
fsl_u08 FSL_EEPROMWrite(fsl_u32 s_address_u32, fsl_u08 word_count_u08)	Blankcheck,writes and verify up to 64 words to a specified address.
fsl_u08 FSL_GetSecurityFlags(fsl_u08 *destination_pu08)	Reads the security information.
fsl_u08 FSL_GetActiveBootCluster(fsl_u08 *destination_pu08)	Reads the current value of the boot flag in extra area.
fsl_u08 FSL_GetBlockEndAddr(fsl_u32 *destination_pu32, fsl_u16 block_u16)	Puts the last address of the specified block into *destination_addr_H and destination_addr_L*
fsl_u08 FSL_GetFlashShieldWindow(fsl_u16* start_block_pu16, fsl_u16* end_block_pu16)	Read the flash shield window from the extra area into *start_block_pu16 end_block_pu16*.
fsl_u08 FSL_InvertBootFlag(void)	Inverts the current value of the boot flag in the extra area.
fsl_u08 FSL_SetFlashShieldWindow(fsl_u16 start_block_u16, fsl_u16 end_block_u16)	Sets the falsh shield window.
fsl_u08 FSL_SetChipEraseProtectFlag(void)	Sets the chip-erase-protection flag in the extra area.

Function prototype	Outline
fsl_u08 FSL_SetBlockEraseProtectFlag(void)	Sets the block-erase-protection flag in the extra area.
fsl_u08 FSL_SetWriteProtectFlag(void)	Sets the write-protection flag in the extra area.
fsl_u08 FSL_SetBootClusterProtectFlag(void)	Sets the bootcluster-update-protection flag in the extra area.
void FSL_SwapBootCluster(void)	This functions swaps the boot cluster 0 and 1 physically. After reset the boot cluster is active regarding the boot flag.
void FSL_ForceReset(void)	Generate software reset.
void FSL_SetInterruptMode(void)	This function forces the FSL to return to the user as fast as possible.

6.2 Library explanation

Each self-programming function is explained in the following format.

Flash self-programming Function name

Outline Outlines the self-programming function.

Function prototype Shows the C-Compiler function prototype of the current function.

Note In this manual, the data type name is defined as followed.

Definition	Data Type
fsl_u08	unsigned char
fsl_u16	unsigned int
fsl_u32	unsigned long int

Argument Indicates the argument of the self-programming function.

Return Value Indicates the return value from the self-programming function.

Register status after calling Indicates the status of registers after the self-programming function is called.

Call example Indicates an example of calling the self-programming function from a user program written in C language.

Flow Indicates the program flow of the self-programming function.

6.2.1 FSL_Open

Outline This function offers an standardized but configurable way to open a self-programming session. If required, the interrupt controller can be backed-up and reprogrammed for flash update period only. Additional applications specific code can be added here if necessary for opening the flash update process.The FLMD0 will be switched to HIGH level according to macro definition FSL_FLMD0_HIGH.

Note

- Call this function at the beginning of the self-programming operation.
- User may customize this function in the source files **fsl_user.h** and **fsl_user.c**, do a few more preprocesses, so as to adapt personal requirements.

Function prototype void FSL_Open (void)

Pre-condition None

Argument None

Return value None

Flow The following figure shows the flow of the self-programming open function.

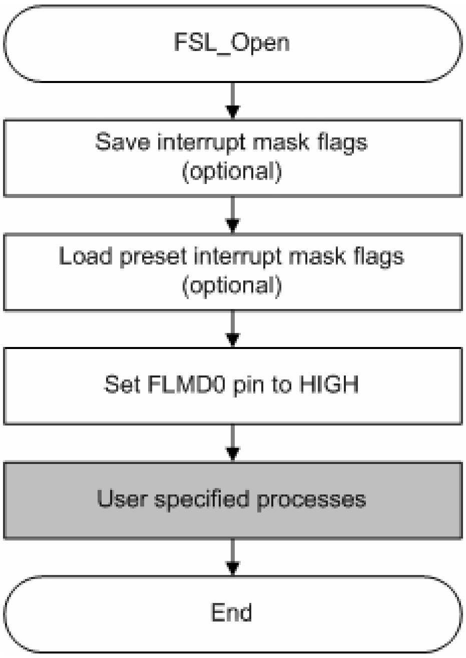


Figure 6-1 Flow of self-programming Open Function

Note The preset interrupt mask flags are defined in the FSL user-configurable source file **fsl_user.h**

```
#define   FSL_MK0L_MASK   0xFF   /* all interrupts disabled */
#define   FSL_MK0H_MASK   0xFF   /* all interrupts disabled */
#define   FSL_MK1L_MASK   0xFF   /* all interrupts disabled */
#define   FSL_MK1H_MASK   0xFF   /* all interrupts disabled */
#define   FSL_MK2L_MASK   0xFF   /* all interrupts disabled */
#define   FSL_MK2H_MASK   0xFF   /* all interrupts disabled */
/*For the correct settings please refer to the chapter "Interrupt Functions"
of the corresponding device user's manual.*/
```

Interrupt backup If backup of interrupt mask flags is not necessary, user may comment out the following line.

```
#define FSL_INT_BACKUP
```

FLMD0 port setting example Following example shows the macro definition for the FLMD0 control.

```
/* FLMD0 control bit                        */
#define  FSL_FLMD0_HIGH   {BECTL_bit.no7 = 1;}
#define  FSL_FLMD0_LOW    {BECTL_bit.no7 = 0;}

/* FSL_Open();    */
FSL_FLMD0_HIGH;
```

6.2.2 FSL_Close

Outline This function offers an standardized but configurable way to close a self-programming session. If reprogrammned in FSL_Open(), the interrupt controller will be restored automatically. Additional applications specific code can be added here if necessary for closing the flash update process. The FLMD0 will be switched to LOW level according to macro definition FSL_FLMD0_LOW.

Note

- Call this function at the end of the self-programming operation.
- User may customize this function in the source files **fsl_user.h** and **fsl_user.c**.

Function prototype void FSL_Close (void)

Pre-condition None

Argument None

Return value None

Flow The following figure shows the flow of the self-programming end function.

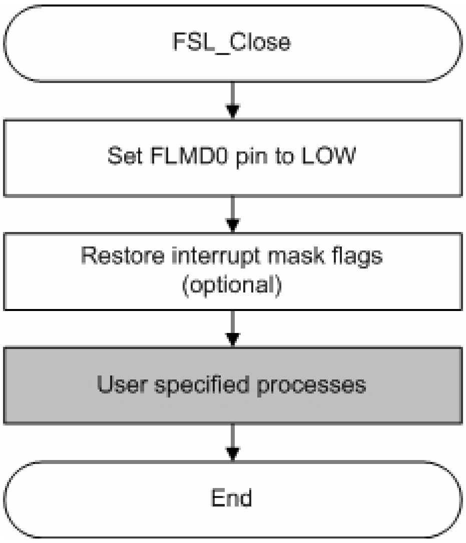


Figure 6-2 Flow of self-programming End Function

6.2.3 FSL_Init

Outline This function Initializes internal self-programming environment. After initialization the start address of the data-buffer is registered for self-programming.

Function prototype fsl_u08 FSL_Init (fsl_u08* data_buffer_pu08)

Pre-condition

- The function FSL_Open() was successfully called.
- The constant FSL_SYSTEM_FREQUENCY has to be adapted according to the used system frequency.

Note This frequency value will not be checked by the FSL, whether it is in the valid range.

Argument

Argument	C Language
First address of data buffer[Note]	fsl_u08* data_buffer_pu08

Argument	Assembler
First address of data buffer[Note]	Data model near: AX Data model far: [SP+0] = LOW(LWRD(data_buffer_addr)) [SP+1] = HIGH(LWRD(data_buffer_addr)) [SP+2] = LOW(HWRD(data_buffer_addr)) [SP+3] = HIGH(HWRD(data_buffer_addr))

Note For details on data buffer, please refer to the chapter "Software Environment".

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion - Initialisation completed
1FH	Initialization interrupted by user interrupt. To resume the intialization the FSL_Init_cont function must be called.
OTHER	Error

Register status after calling **A = return value, X = destroyed**

Call example

```
/* Operation without interrupts     */

extern fsl_u08 fsl_data_buffer[FSL_DATA_BUFFER_SIZE]; /* see fsl_user.c */

my_status_u08 = FSL_Init((fsl_u08*)&fsl_data_buffer);

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

6.2.4 FSL_Init_cont

Outline This function resumes the interrupted FSL_Init function. After initialization the start address of the data-buffer is registered for self-programming.

Function prototype fsl_u08 FSL_Init_cont (fsl_u08* data_buffer_pu08)

Pre-condition

- The function FSL_Open() was successfully called and FSL_Init was interrupted.
- The constant FSL_SYSTEM_FREQUENCY has to be adapted according to the used system frequency.

Note This frequency value will not be checked by the FSL, whether it is in the valid range.

Argument

Argument	C Language
First address of data buffer[Note]	fsl_u08* data_buffer_pu08

Argument	Assembler
First address of data buffer[Note]	Data model near: AX Data model far: [SP+0] = LOW(LWRD(data_buffer_addr)) [SP+1] = HIGH(LWRD(data_buffer_addr)) [SP+2] = LOW(HWRD(data_buffer_addr)) [SP+3] = HIGH(HWRD(data_buffer_addr))

Note For details on data buffer, please refer to the chapter "Software Environment".

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion - Initialisation completed
1FH	Initialization interrupted by user interrupt. To resume the intialization the FSL_Init_cont function must be called.
OTHER	Error

Register status after calling **A = return value, X = destroyed**

Call example

```
/* Operation without interrupts     */
extern fsl_u08 fsl_data_buffer[FSL_DATA_BUFFER_SIZE]; /* see fsl_user.c */
my_status_u08 = FSL_Init((fsl_u08*)&fsl_data_buffer);
while(my_status_u08 == 0x1F)
{
   my_status_u08 = FSL_Init_cont((fsl_u08*)&fsl_data_buffer);
}
if( my_status_u08 != 0x00 ) my_error_handler();
```

6.2.5 FSL_ModeCheck

Outline This function checks the voltage level at FLMD0 pin, ensuring the hardware requirement of self-programming.

For details on FLMD0 and hardware requirement, please refer tothe chapter "Hardware Environment".

Note Call this function after calling the self-programming open function FSL_Open to check the voltage level of the FLMD0 pin.

Caution If the FLMD0 pin is at low level, operations such as erasing and writing the flash memory cannot be performed. To manipulate the flash memory by self-programming, it is necessary to call this function and confirm, that the FLMD0 pin is at high level.

Function prototype fsl_u08 FSL_ModeCheck (void)

Pre-condition The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument None

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion -FLMD0 pin is at high level.
01H	Abnormal termination -FLMD0 pin is at low level.

Register status after calling **A = return value**

Call example

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

6.2.6 FSL_BlankCheck

Outline

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

Note

- If the block is not blank, it should be erased and blank checked again.
- Because only one block is checked at a time, call this function once for each block.

Function-prototype

fsl_u08 FSL_BlankCheck (fsl_u16 block_u16)

Pre-condition

The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C Language
block number to be checked	fsl_u16 block_u16

Argument	Assembly
block number to be checked	Data model near: AX Data model far: AX

Return Value

The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion Specified block is blank (erase operation is completed).
05H	Parameter error Specified block number is outside the allowed range.
1BH	Black check error Specified block is not blank (erase operation is not completed).
1FH	Process interrupted. A user interrupt occurs while this function is in process.

Register status after calling

A = return value

Call example

```
 my_block_u16 = 0x007F;

do
{
  my_status_u08 = FSL_BlankCheck(my_block_u16);

  // in case of FSL_ERR_INTERRUPTION is returned here,
  // the corresponding ISR is already executed !!!

} while (my_status_u08 == FSL_ERR_INTERRUPTION);

// exit if error occurs
if (my_status_u08 != FSL_OK) my_error_handler(....)
```

6.2.7 FSL_Erase

Outline This function erases a specified block.

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

Function prototype fsl_u08 FSL_Erase (fsl_u16 block_u16)

Pre-condition The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C Language
block number to be erased	fsl_u16 block_u16

Argument	Assembly
block number to be checked	Data model near: AX Data model far: AX

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error Specified block number is outside the allowed range.
10H	Protect error Specified block is included in the boot area and rewriting the boot area is disabled or block is outside the flash shield window.
1AH	Erase error An error occurred during this function in process.
1FH	Process interrupted. A user interrupt occurs while this function is in process.

Register status after calling **A = return value**

Call example

```
 my_block_u16 = 0x007F;

do
{
  my_status_u08 = FSL_Erase(my_block_u16);

  // in case of FSL_ERR_INTERRUPTION is returned here,
  // the corresponding ISR is already executed !!!

} while (my_status_u08 == FSL_ERR_INTERRUPTION);

// exit if error occurs
if (my_status_u08 != FSL_OK) my_error_handler(....)
```

6.2.8 FSL_IVerify

Outline 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 written data in the flash memory is at a sufficient voltage level.
- It is different from a logical verification that just compares data.

Caution After writing data, verify (internal verification) the block including the range in which the data has been written. If verification is not executed, the written data is not guaranteed.

Function prototype fsl_u08 FSL_IVerify (fsl_u16 block_u16)

Pre-condition The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C language
the to-verify block number	fsl_u16 block_u16

Argument	Assembly
block number to be checked	Data model near: AX Data model far: AX

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error Specified block number is outside the allowed range.
1BH	Verify (internal verify) error An error occurs during this function is in process.
1FH	Process interrupted. A user interrupt occurs while this function is in process.

Register status after calling **A = return value**

Call example

```
 my_block_u16 = 0x007F;

do
{
  my_status_u08 = FSL_IVerify(my_block_u16);
} while (my_status_u08 == FSL_ERR_INTERRUPTION);

if (my_status_u08 != FSL_OK) my_error_handler(....)
```

6.2.9 FSL_Write

Outline

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.

Function prototype

fsl_u08 FSL_Write (fsl_u32 s_address_u32, fsl_u08 word_count_u08)

Pre-condition

The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init. Data buffer was filled with data, which will be written into the flash.

Argument

Argument	C language
starting address of the data to be written[Note]	fsl_u32 s_address_u32
Number of the data to be written (1 to 64)	fsl_u08 word_count_u08

Argument	C language
starting address of the data to be written[Note]	Data model near: AX = HIGH(address) BC = LOW(address) Data model far: AX = HIGH(address) BC = LOW(address
Number of the data to be written (1 to 64)	Data model near: D Data model far: D

Note

- **s_address_u32** is a physical address(e.g. 1FC00H), not a logical address(e.g. 5BC00H)
- **(s_address_u32** + (Number of data to be written x 4 bytes)) must not straddle over the end address of a single block.
- **s_address_u32** must be a multiple of 4
- Most significant byte (MSB) of the **s_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 Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error - Start address is not a multiple of 1 word (4 bytes). - The number of data to be written is 0. - The number of data to be written exceeds 64 words. - Write end address (Start address + (Number of data to be written x 4 bytes)) exceeds the flash memory area.
10H	Protect error Specified range includes the boot area and rewriting the boot area is disabled or address is outside the flash shield window.
1CH	Write error Data is verified but does not match after this function operation is completed or FLMD0 pin is low.
1FH	Process interrupted. A user interrupt occurs while this function is in process.

Register status after calling **A = return value; X, B, C and D destroyed**

Call example

```
 // prepare data and write it into the data buffer for the writing process
..........
..........

my_address_u32 = 0x0001FC00;   // set address for write procedure
my_write_count_u08 = 0x02;     // set word count

do
{
  my_status_u08 = FSL_Write(my_address_u32, my_write_count_u08);

  // in case of FSL_ERR_INTERRUPTION is returned here,
  // the corresponding ISR is already executed !!!

} while (my_status_u08 == FSL_ERR_INTERRUPTION);

// exit if error occurs
if (my_status_u08 != FSL_OK) my_error_handler(....)
```

6.2.10 FSL_EEPROMWrite

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

Different to **FSL_Write**, blank check will be performed, before "writing" n words. After "writing" n words internal verify is performed.

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

- It is not allowed to overwrite data in flash memory.
- Only blank flash cells can be used for the write.

Function prototype fsl_u08 FSL_EEPROMWrite (fsl_u32 s_address_u32, fsl_u08 word_count_u08)

Pre-condition The self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C language
starting address of the data to be written[Note]	fsl_u32 s_address_u32
Number of the data to be written (1 to 64)	fsl_u08 word_count_u08

Argument	C language
starting address of the data to be written[Note]	Data model near: AX = HIGH(address) BC = LOW(address) Data model far: AX = HIGH(address) BC = LOW(address
Number of the data to be written (1 to 64)	Data model near: D Data model far: D

Note

- **(s_address_u32** + (Number of data to be written x 4 bytes)) must not straddle over the end address of a single block.
- **s_address_u32** must be a multiple of 4
- Most significant byte (MSB) of the **s_address_u32** has to be 0x00 In other words, only *0x00abcdef* is a valid flash address.
- **word_count***4 has to be smaller than the size of data buffer. The firmware does not check this.

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error - Start address is not a multiple of 1 word (4 bytes). - The number of data to be written is 0. - The number of data to be written exceeds 64 words. - Write end address (Start address + (Number of data to be written x 4 bytes)) exceeds the flash memory area.
10H	Protect error Specified range includes the boot area and rewriting the boot area is disabled or address is outside the flash shield window.
1CH	Write error Data is verified but does not match after this function operation is completed or FLMD0 pin is low..
1DH	Verify error Data is verified but does not match after it has been written.
1EH	Blank error Write area is not a blank area.
1FH	Process interrupted. A user interrupt occurs while this function is in process.

Register status after calling **A = return value; X, B, C and D destroyed**

```
 // prepare data and write it into the data buffer for the writing process
..........
..........

my_address_u32 = 0x0001FC00;   // set address for write procedure
my_write_count_u08 = 0x02;     // set word count

do
{
  my_status_u08 = FSL_EEPROMWrite(my_address_u32, my_write_count_u08);

  // in case of FSL_ERR_INTERRUPTION is returned here,
  // the corresponding ISR is already executed !!!

} while (my_status_u08 == FSL_ERR_INTERRUPTION);

// exit if error occurs
if (my_status_u08 != FSL_OK) my_error_handler(....)
```

6.2.11 FSL_GetSecurityFlags

Outline This function reads the security (write-/erase-protection) information from the extra area.

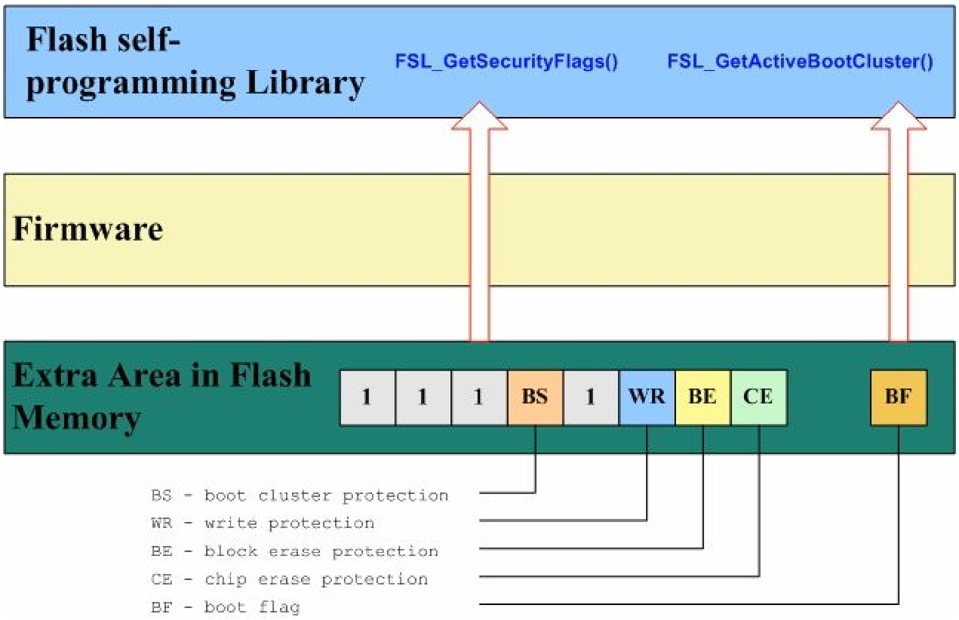


Figure 6-3 Security Information Structure

Function prototype fsl_u08 FSL_GetSecurityFlags (fsl_u08 *destination_pu08)

Pre-condition The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C language
Storage address of the security information	fsl_u08 *destination_pu08

Argument	Assembly
Storage address of the security information	Data model near: AX Data model far: [SP+0] = LOW(LWRD(dest_address)) [SP+1] = HIGH(LWRD(dest_address)) [SP+2] = LOW(HWRD(dest_address)) [SP+3] = HIGH(HWRD(dest_address))

Return Value

The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error

Change in the destination address.

Security flag will be written in the destination address.

Meaning of each bit of security flag.
Bit 0: Chip erase protection (0: Enabled, 1: Disabled)
Bit 1: Block erase protection (0: Enabled, 1: Disabled)
Bit 2: Write protection (0: Enabled, 1: Disabled)
Bit 4: Boot area overwrite protection (0: Enabled, 1: Disabled)
Bits 3, 5, 6 and 7 are always 1.

Example

If *EB*H (i.e. *11101011*) is written to destination address, boot area overwrite and write operations to the flash area are forbidden.

Register status after calling

A = return value, X = destroyed

Call example

```
/* extern variable declaration(see fsl_user.c) */
extern fsl_u08 fsl_data_buffer[FSL_DATA_BUFFER_SIZE];

/* 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 & 0x01){ myPrintFkt("Chip erase protection disabled!"); }
else{ myPrintFkt("Chip erase protection enabled!"); }
```

6.2.12 FSL_GetActiveBootCluster

Outline This function reads the current value of the boot flag in extra area.

Function prototype fsl_u08 FSL_GetActiveBootCluster (fsl_u08 *destination_pu08)

Pre-condition The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C language
Destination address of the boot swap info	fsl_u08 *destination_pu08

Argument	Assembly
Storage address of the security information	Data model near: AX Data model far: [SP+0] = LOW(LWRD(dest_address)) [SP+1] = HIGH(LWRD(dest_address)) [SP+2] = LOW(HWRD(dest_address)) [SP+3] = HIGH(HWRD(dest_address))

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error

Changes in the destination address.

Boot flag will be written in the destination address.

00H: Boot area is not swapped.
01H: Boot area is swapped.

Register status after calling **A = return value, X = destroyed**

Call example

```
/* extern variable declaration(see fsl_user.c) */
extern fsl_u08 fsl_data_buffer[FSL_DATA_BUFFER_SIZE];

/* get boot-swap flag                                   */
my_status_u08 = FSL_GetActiveBootCluster((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!"); }
```

6.2.13 FSL_GetBlockEndAddress

Outline This function puts the last address of the specified block into *destination_pu32.

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.

Function prototype fsl_u08 FSL_GetBlockEndAddr ((fsl_u32*) destination_pu32, fsl_u16 block_u16)

Pre-condition The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C language
Destination address of the block end address info	fsl_u32 *destination_pu32
Block number the end-address is asked for	fsl_u16 block_u16

Argument	Assembly
Destination address of the block end address info	Data model near: AX Data model far: [SP+0] = LOW(LWRD(dest_addr)) [SP+1] = HIGH(LWRD(dest_addr)) [SP+2] = LOW(HWRD(dest_addr)) [SP+3] = HIGH(HWRD(dest_addr))
Block number the end-address is asked for	Data model near: BC Data model far: AX

Return Value The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error

Changes in the destination address.

Block end address will be written in the destination address.

Example

If *6C*H is given as block number, *367FF*H will be written to the destination address.

Register status after calling **A = return value, X, B, C = destroyed**

Call example

```
/* extern variable declaration(see fsl_user.c) */
extern fsl_u08 fsl_data_buffer[DATA_BUFFER_SIZE];


fsl_u32 my_address_u32;
fsl_u08 my_block_u16 = 0x007F;

/* get end adress of the block                         */
my_status_u08 = FSL_GetBlockEndAddr((fsl_u32*)&my_address_u32, my_block_u16);

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

/*      ####### ANALYSE my_address_u32 #######         */
```

6.2.14 FSL_GetFlashShieldWindow

Outline

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 finction FSL_SetFlashShieldWindow.

Example:

Flash shield window start block is 0x60
Flash shield window end block is 0x63

This configuration of the flash shield window prohibits the user to write e.g. into the block …..0x5E,0x5F,0x64,0x65…..

Function prototype

fsl_u08 FSL_GetFlashShieldWindow(fsl_u16* start_block_pu16, fsl_u16* end_block_pu16)

Pre-condition

The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C language
Destination address for the start block of the flash shield window	fsl_u16* start_block_pu16
Destination address for the end block of the flash shield window	fsl_u16* end_block_pu16

Argument	Assembly
Destination address for the start block of the flash shield window	Data model near: AX Data model far: [SP+0] = LOW(LWRD(FSW_start_block)) [SP+1] = HIGH(LWRD(FSW_start_block)) [SP+2] = LOW(HWRD(FSW_start_block)) [SP+3] = HIGH(HWRD(FSW_start_block))
Destination address for the end block of the flash shield window	Data model near: BC Data model far: [SP+4] = LOW(LWRD(FSW_end_block)) [SP+5] = HIGH(LWRD(FSW_end_block)) [SP+6] = LOW(HWRD(FSW_end_block)) [SP+7] = HIGH(HWRD(FSW_end_block))

Return Value

The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error

Register status after calling

A = return value, X, B, C = destroyed

Call example

```
/* extern variable declaration(see fsl_user.c) */
extern fsl_u08 fsl_data_buffer[DATA_BUFFER_SIZE];

fsl_u32 my_address_u32;
fsl_u08 my_block_u16 = 0x007F;
/* read flash shield window                         */
my_status_u08 = FSL_GetBlockEndAddr((fsl_u16*)&myFSW_start, (fsl_u16*)&myFSW_end

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

/*      ####### ANALYSE flash shield window #######         */
```

6.2.15 FSL_SetFlashShieldWindow

Outline

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 is 0x60
Flash shield window end block is 0x63

This configuration of the flash shield window prohibits the user to write e.g. into the block …..0x5E,0x5F,0x64,0x65…..

Function prototype

fsl_u08 FSL_SetFlashShieldWindow(fsl_u16 start_block_u16, fsl_u16 end_block_u16)

Pre-condition

The flash self-programming environment was successfully opened by the functions FSL_Open and FSL_Init.

Argument

Argument	C language
Start block for the flash shield window	fsl_u16 start_block_u16
End block for the flash shield window	fsl_u16 end_block_u16

Argument	Assembly
Start block for the flash shield window	Data model near: AX Data model far: AX
End block for the flash shield window	Data model near: BC Data model far: BC

Return Value

The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error - Internal error
10H	Protection error - Attempt is made to enable a flag that has already been disabled. - Attempt is made to change the boot area swap flag while rewriting of the boot area is disabled.
1AH	Erase error An erase error occurs while this function is in process.
1BH	Internal verify error A verify error occurs while this function is in process.
1CH	Write error A write error occurs while this function is in process.

Register status after calling A = return value, X, B, C = destroyed

Call example

```
fsl_u16 myFSW_start = 0x0002;
fsl_u16 myFSW_end = 0x0004;


/* set flash shield window                          */
my_status_u08 = FSL_SetFlashShieldWindow(myFSW_start, myFSW_end);

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

6.2.16 FSL_SetXXX and FSL_InvertBootFlag

Outline The self-programming library has 5 functions for setting security bits . Each dedicated function sets a corresponding security flag in the extra area.

These functions are listed below.

Funtion name	Outline
invert boot flag function	Inverts the current value of the boot flag*.
set chip-erase-protection function	Sets the chip-erase-protection flag*.
set block-erase-protection function	Sets the block-erase-protection flag*.
set write-protection function	Sets the write-protection flag*.
set boot-cluster-protection function	Sets the bootcluster-update-protection flag*.

* This flag is stored in the flash extra area.

Caution

1. **Chip-erase protection and boot-cluster protection cannot be reset by programmer.**
2. After RESET the other boot-cluster is activated. Please ensure a valid boot-loader inside the area, before calling the function.
3. Each security flag can be written by the application only once until next reset.
4. Block-erase protection and write protection can be reset by programmer.

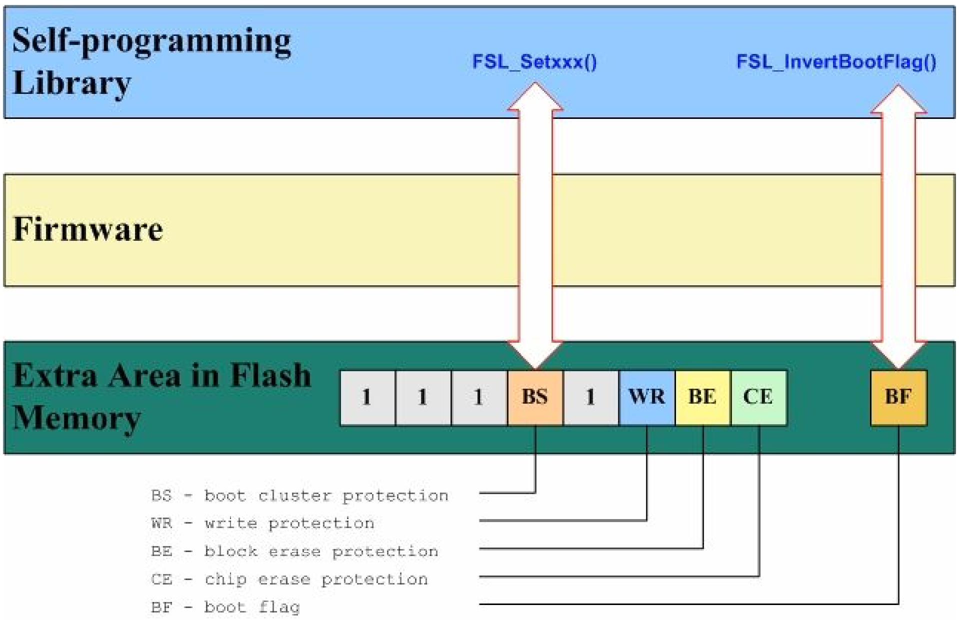


Figure 6-4 Extra Area

Function prototypes

Function name	Function prototype
invert boot flag function	fsl_u08 FSL_InvertBootFlag(void)
set chip-erase-protection function	fsl_u08 FSL_SetChipEraseProtectFlag(void)
set block-erase-protection function	fsl_u08 FSL_SetBlockEraseProtectFlag(void)
set write-protection function	fsl_u08 FSL_SetWriteProtectFlag(void)
set boot-cluster-protection function	fsl_u08 FSL_SetBootClusterProtectFlag(void)

Argument

None

Return Value

The status is stored in *A register* in assembly language, and returned in the *fsl_u08* type variable in C language.

Status	Explanation
00H	Normal completion
05H	Parameter error - Internal error
10H	Protection error - Attempt is made to enable a flag that has already been disabled. - Attempt is made to change the boot area swap flag while rewriting of the boot area is disabled.
1AH	Erase error An erase error occurs while this function is in process.
1BH	Internal verify error A verify error occurs while this function is in process.
1CH	Write error A write error occurs while this function is in process.

Register status after calling

A = return value

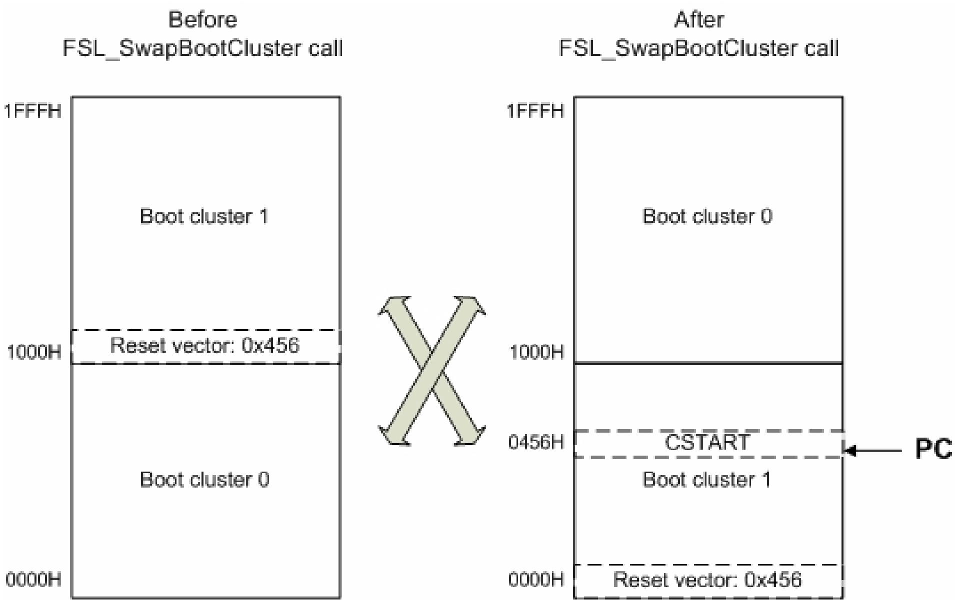
Call example

```
my_status_u08 = FSL_SetBlockEraseProtectFlag();

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

6.2.17 FSL_SwapBootCluster

Outline This function performs the physically 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.



Note **After the execution of this function boot cluster 1 is located from the address 0x0000 to 0x1000 and PSW.IE bit is cleared! After reset the boot clusters will be switch regarding the boot swap flag.**

Function prototype void FSL_SwapBootCluster(void)

Pre-condition None

Argument None

Return value None

6.2.18 FSL_ForceReset

Outline This function generates a software reset. For detailed information please refer to the device Users Manual.

Function prototype void FSL_ForceReset(void)

Pre-condition None

Argument None

Return value None

6.2.19 FSL_SetInterruptMode

Outline This function forces the interrupted FSL function to leave as fast as possible to the user application. Usage is only inside ISRs permitted.

Caution:
If FSL_SetInterruptMode function was called before execution of any other FSL_XXX function, the FSL_XXX function may with 0x1F (interrupted status), also if no interrupt is occured.

Function prototype void FSL_SetInterruptMode(void)

Pre-condition Interrupt is occured.

Argument None

Return value None

6.3 Sample linker file

The self-programming library uses three segments for data, code and constants allocation:

- **FSL_CODE(code)**
  Within this segment the flash self-programming library will be located. Be sure to locate this segment within internal flash.
- **FSL_CNST(constants)**
  Internal frequency constant will be located inside this segment.
- **FSL_DATA(data)**
  Internal data will be located inside this segment.

Listed below is a sample linker file(for uPD78F1166) for the self-programming library.

```
//-------------------------------------------------------------------------
//      Define CPU
//-------------------------------------------------------------------------
-c78000


//-------------------------------------------------------------------------
//      Size of the stack.
//      Remove comment and modify number if used from command line.
//-------------------------------------------------------------------------
//-D_CSTACK_SIZE=80


//-------------------------------------------------------------------------
//      Allocate the read only segments that are mapped to ROM.
//-------------------------------------------------------------------------
//      Interrupt vector segment.
//-------------------------------------------------------------------------

-Z(CODE)INTVEC=00000-0007F


//-------------------------------------------------------------------------
//      CALLT vector segment.
//-------------------------------------------------------------------------
 -Z(CODE)CLTVEC=00080-000BD


//-------------------------------------------------------------------------
//      OPTION BYTES segment.
//-------------------------------------------------------------------------

-Z(CODE)OPTBYTE=000C0-000C3


//-------------------------------------------------------------------------
//      SECURITY_ID segment.
//-------------------------------------------------------------------------
-Z(CODE)SECUID=000C4-000CE
```

```
//-----------------------------------------------------------------------
//      Reserved ROM area for Minicube Firmware: 000D0-00383
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
//      FAR far data segments.
//      The FAR_I and FAR_ID segments must start at the same offset
//      in a 64 Kb page.
//-----------------------------------------------------------------------
-Z(FARCONST)FAR_ID=0CF00-3FFFF
-Z(FARDATA)FAR_I=FD700-FFE1F


// FSL
// ============
-Z(CODE)FSL_CODE=0100-0FFE
-Z(CONST)FSL_CNST=0100-0FFE


//-----------------------------------------------------------------------
//      Startup, Runtime-library, Near, Interrupt
//      and CALLT functions code segment.
//-----------------------------------------------------------------------

-Z(CODE)RCODE,CODE=02000-0FFFF


//-----------------------------------------------------------------------
//      Far functions code segment.
//-----------------------------------------------------------------------
-Z(CODE)XCODE=[02000-3FFFF]/10000


//-----------------------------------------------------------------------
//      Data initializer segments.'
//-----------------------------------------------------------------------
-Z(CONST)NEAR_ID=[02000-0FFFF]/10000
-Z(CONST)SADDR_ID=[02000-0FFFF]/10000
-Z(CONST)DIFUNCT=[02000-0FFFF]/10000


//-----------------------------------------------------------------------
//      Constant segments
//-----------------------------------------------------------------------
-Z(CONST)NEAR_CONST=_NEAR_CONST_LOCATION_START-_NEAR_CONST_LOCATION_END
-P(CONST)FAR_CONST=[02000-3FFFF]/10000
-Z(CONST)SWITCH=02000-0FFFF
-Z(CONST)FSWITCH=[02000-3FFFF]/10000


//-----------------------------------------------------------------------
//      Allocate the read/write segments that are mapped to RAM.
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//      Short address data and workseg segments.
//-----------------------------------------------------------------------
-Z(DATA)WRKSEG=FFE20-FFEDF
-Z(DATA)SADDR_I,SADDR_Z,SADDR_N=FFE20-FFEDF


//-----------------------------------------------------------------------
//      Near data segments.
//-----------------------------------------------------------------------

-Z(DATA)NEAR_I,NEAR_Z,NEAR_N,DS_DBF,FSL_DATA=FD702-FFE1F


//-----------------------------------------------------------------------
//      Far data segments.
//-----------------------------------------------------------------------
-Z(FARDATA)FAR_Z=FD708-FFE1F
-P(DATA)FAR_N=[FD700-FFE1F]/10000
```

```
//-----------------------------------------------------------------------
//      Heap segments.
//-----------------------------------------------------------------------
-Z(DATA)NEAR_HEAP+_NEAR_HEAP_SIZE=FD700-FFE1F
-Z(DATA)FAR_HEAP+_FAR_HEAP_SIZE=[FD700-FFE1F]/10000


//-----------------------------------------------------------------------
//      Stack segment.
//-----------------------------------------------------------------------
-Z(DATA)CSTACK+_CSTACK_SIZE=FD700-FFD1F
```

6.4 How to integrate the library into an application

1. copy all the files into your project subdirectory

2. add all fsl*.* files into your project (workbench or make-file)
   NOTE: Only one FSL library file (*.r26) must be included.
   (for data model near -> fsl_near.r26 or data model far -> fsl_far.r26)

3. adapt project specific files as follows:
   - fsl_user.h:
     - adapt the system frequency expressed in [Hz]
     - adapt the size of data-buffer you want to use for data exchange between firmware and application
     - define the interrupt scenario (enable interrupts that should be active during self-programming)
     - define the back-up functionality during selfprogramming whether required or not
   - fsl_user.c:
     - adapt FSL_Open() and FSL_Close() due to your requirements

4. adapt the *.XCL file due to your requirements (at least segments FSL_CODE, FSL_CNST and FSL_DATA should be defined)

5. re-compile the project

Chapter 7 Sample code

The following example shows the typically call and interrupt handling sequence of the self-programming library.

```
// ======================================================================
// execute the selected command
// ======================================================================
 FSL_Open();

 if (FSL_ModeCheck() != FSL_OK) My_Error_Handler(....);

 my_status_u08 = FSL_Init( &my_data_buffer);

 while (my_status_u08 == FSL_ERR_INTERRUPTION);
 {
    my_status_u08 = FSL_Init_cont( &my_data_buffer);
 }

 // check block by block if blank
 for (my_block_u16 = 0; my_block_u16 <= 0x7F; my_block_u16++)
 {
 // blank-check current block as long as not completed or error occurs
 // -------------------------------------------------------------------
    do
    {
       my_status_u08 = FSL_BlankCheck(my_block_u16);

       // in case of FSL_ERR_INTERRUPTION is returned here,
       // the corresponding ISR is already executed !!!

    } while (my_status_u08 == FSL_ERR_INTERRUPTION);

    // exit if error occurs
    if (my_status_u08 != FSL_OK) My_Error_Handler(....);
 }
 FSL_Close();
// ======================================================================


// ======================================================================
// handling of the FSL_SetInterruptMode function inside interrupts
// ======================================================================

#pragma vector = INTSRE3_vect
__interrupt void isr_sre3(void)
{
  // store received data into receive buffer
  .........
  .........
  .........
  if( receive_buffer_full )
  {
     .........
     .........
     .........
     FSL_SetInterruptMode();
  }
}
```