NEC

Application Note

Self-Programming

32-/16-bit Single-Chip Microcontroller

Self-Programming Library for embedded Single Voltage FLASH

Document No. U16929EE3V2AN00
Date Published May 2007

Printed in Germany

NOTES FOR CMOS DEVICES

① VOLTAGE APPLICATION WAVEFORM AT INPUT PIN

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (MAX) and V_{IH} (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (MAX) and V_{IH} (MIN).

② HANDLING OF UNUSED INPUT PINS

Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

③ PRECAUTION AGAINST ESD

A strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

④ STATUS BEFORE INITIALIZATION

Power-on does not necessarily define the initial status of a MOS device. Immediately after the power source is turned ON, devices with reset functions have not yet been initialized. Hence, power-on does not guarantee output pin levels, I/O settings or contents of registers. A device is not initialized until the reset signal is received. A reset operation must be executed immediately after power-on for devices with reset functions.

⑤ POWER ON/OFF SEQUENCE

In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current.

The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

⑥ INPUT OF SIGNAL DURING POWER OFF STATE

Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements.

Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

For further information, please contact:

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668,
Japan
Tel: 044-435-5111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554, U.S.A.
Tel: 408-588-6000
800-366-9782
http://www.am.necel.com/

[Europe]

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

Hanover Office
Podbielski Strasse 166 B
30177 Hannover
Tel: 0 511 33 40 2-0

Munich Office
Werner-Eckert-Strasse 9
81829 München
Tel: 0 89 92 10 03-0

Stuttgart Office
Industriestrasse 3
70565 Stuttgart
Tel: 0 711 99 01 0-0

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908-691-133

Succursale Française
9, rue Paul Dautier, B.P. 52180
78142 Velizy-Villacoublay Cédex
France
Tel: 01-3067-5800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091-504-2787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 638 72 00

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

Branch The Netherlands
Limburglaan 5
5616 HR Eindhoven
The Netherlands
Tel: 040 265 40 10

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27 ZhiChunLu Haidian
District, Beijing 100083, P.R.China
TEL: 010-8235-1155
http://www.cn.necel.com/

NEC Electronics Shanghai Ltd.
Room 2509-2510, Bank of China Tower,
200 Yincheng Road Central,
Pudong New Area, Shanghai P.R. China P.C:200120
Tel: 021-5888-5400
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/

Seoul Branch
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku,
Seoul, 135-080, Korea
Tel: 02-558-3737

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R. O. C.
Tel: 02-2719-2377

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

G05.11-1A

- **The information in this document is current as of March, 2006. 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).

M8E 02. 11-1

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

In the history of microcontroller development the selection of appropriate non volatile memory always played an important role. In the beginning only mask ROM and OTP (one time programmable) devices were available. The later on developed external EEPROM and FLASH devices achieved a new level of flexibility in the system development. However, the high price of these external solutions often made it necessary for mass products to do a cost reduction step after development or after production ramp up. The re-programmable memory was then replaced by mask ROM.

The further ongoing technology improvement and the possibility to embed non volatile memory into the microcontroller silicon resulted in a new generation of applications concentrating all intelligence into one single device.

The continuously increasing application complexity requires more and more, that program updates, bug fixes or feature extensions are possible also during production or in the field. This requirement together with the continuously closing price gap between embedded Flash and mask ROM leads to the clear trend to use Flash in mass production.

NEC's embedded Flash microcontrollers support this trend with their excellent quality and high sophisticated features. Using our dedicated Flash programmer PG-FP4 or third party tools, Flash programming is made easy from development to production. Data security is ensured by special protection features, that can be set in different levels according to the application requirements. Partially or even complete application update end of line or in the field is supported by our secure self-programming.

The Flash programming using dedicated Flash programmers is explained in the users manuals of the programmers. Information with respect to the connection between the programmers and the devices or special device treatment during programming can also be found in the devices users manuals.

This document concentrates on the wide field of self-programming. With respect to this, two basic technologies have to be differentiated. They differ regarding the Flash features, especially regarding self-programming:

- Single voltage Flash
  No dedicated programming voltage is required. Flash features and self-programming of Single Voltage Flash is covered by this document.

- Dual voltage Flash
  This Flash needs a dedicated Flash programming voltage (usually called V_{PP}). A similar document covering self-programming of dual voltage Flash is available on the internet too.

[MEMO]

Chapter 2 Definition of Terms

Bootloader
: A piece of software located in the boot segment handling the reprogramming of the device.

BROM
: Embedded Mask ROM that holds the device internal firmware and testing routines

Code Flash
: Embedded Flash where the application code is stored.

Data Flash
: Embedded Flash implemented for storage of the data of the EEPROM emulations. Beside that also code operation is possible from Data Flash

DeviceInfo
: The following chapters describe many features of the NEC's embedded Flash devices identified as system requirements. Due to technical reasons, like different device technology and due to the cost impact not all devices contain every explained feature. Furthermore, several device dependent data (latencies, programming times/sizes,...) is important for self-programming.
The collection of all required information about the NEC devices, called *DeviceInfo* later on in this document, can be found in the file "V850_SVF_DRL_Vxxx.pdf", which can also be loaded from the internet.

Note: The DeviceInfo is an important reference for this application note and so shall be loaded from the internet too.

Dual Operation
: Dual operation is the capability to fetch code during reprogramming of the flash memory.

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. The size of the extra area is not included in the size of the device. Means a 128kbyte flash device has 128kByte of flash available for the user.

Extra Data
: Data stored in the device internal extra area(s). Part of the data are protection flags, security flags and variable reset vector. All this is explained later on in the manual.

Firmware
: Firmware is a piece of software that is located in an BROM and handling the interfacing to the flash.

Flash
: "Flash EPROM" - Electrical erasable nonvolatile memory. The difference to ROM is, that 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 Environment
: This is device internal hardware (charge pumps, sequencer,...), required to do any Flash programming. It is not continuously activated, only for programming and self-programming. Activation and deactivation is done by the *SelfLib*.
The re-programmed Flash is not accessible at all while the Flash environment is active. So parts

of the SelfLib need to be executed outside the Flash (e.g. in embedded RAM). Copying code to RAM is supported by the SelfLib and explained later in this document

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

ROM
: “Read only memory” - nonvolatile memory. The content of that memory can not be changed

RAM
: “Random access memory” - volatile memory with random access

SelfLib
: SelfLib is the short form of “Self-Programming Library”.

Self-Programming
: Capability to re-program the embedded flash without external programming tool only via control code running on the micro controller

Single Voltage
: For the reprogramming of single voltage flashes the voltage needed for erasing and programming are generated on-board of the micro controller. No external voltage needed like for dual- voltage flash types

Chapter 3 Overview

This document describes the usage of the self-programming libraries for NEC's microcontrollers with embedded single voltage Flash.

The libraries provide a user friendly interface to the device internal firmware. The firmware itself initiates and controls the different Flash programming steps (Erase, program,...), executed in background by a dedicated hardware.

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

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

For support of other development environments, additional development effort may be necessary. Especially, but maybe not only, the calling conventions with respect to the device internal firmware may be different. As the device firmware is developed using the GHS tool chain, the firmware interface might have to be adapted to call the firmware functions using the GHS calling conventions.

In addition to the library, demo programs are available, showing the implementation and usage of the library. These programs are device and target hardware dependent.

The different options of set-up and usage of the SelfLib are explained in detail in this document.

Note: Please read all chapters of the application note carefully and follow exactly the given sequences and recommendations. This is required in order to make full use of the high level of security and safety provided by the devices.

The self-programming libraries with demo programs, this application note and device dependent information can be downloaded from the following URL:

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

3.1 Process Technology and Flash Technology

NEC V850 devices are produced in different technologies and with different Flash implementations. This results in slightly different self-programming interfaces, different timings and different support of security and protection related features.

UC2 is a 0.25 µm process and as a successor of UC1 (0.35 µm process) it has been the first technology to produce single voltage Flash that needs no dedicated external programming voltage. The Flash implementation used for UC2 is NEC proprietary.

The other process technologies, such as MF2, CZ6HSF or UX4 use the SuperFlash® Flash implementation.

In several countries including the United States and Japan, SuperFlash® is a registered trademark of Silicon Storage Technology.

In the course of this document the SuperFlash® implementation is called SST Flash.
The SST Flash technology differs from UC2 in the cell structure, what is transparent for the user, but also in the block structure, what is relevant for the programming and self-programming strategies.
Furthermore, the timings and the protection features differ (See later chapters).

3.2 Technology Specific Chapters

Different device firmware implementations with corresponding self-programming interfaces have been developed, in order to support self-programming on the different technologies and with the different Flash implementation features in the best way.
With the self-programming interface also the self-programming libraries, containing the user application interface, slightly differ. We have to distinguish between the following three implementations:

- UC2 Flash technology and UC2 style self-programming interface. The chapter names corresponding to UC2 self-programming start with “UC2...”
- CZ6HSF or UX4 technology with SST Flash, featuring a self-programming interface originally introduced on the first UX4 devices. The chapter names corresponding to devices supporting this self-programming interface start with “SST (UX4/CZ6HSF)...”
- MF2 or UX4 technology with SST Flash, featuring a self-programming interface originally introduced on MF2 devices. So, the chapter names corresponding to devices supporting this self-programming interface start with “SST (MF2/UX4)...”

Please refer to the device documentation or to the *DeviceInfo* to select the correct library documentation chapters out of this manual for the selected device.

Chapter 4 Flash

4.1 Block Structures

4.1.1 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 organisation 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.

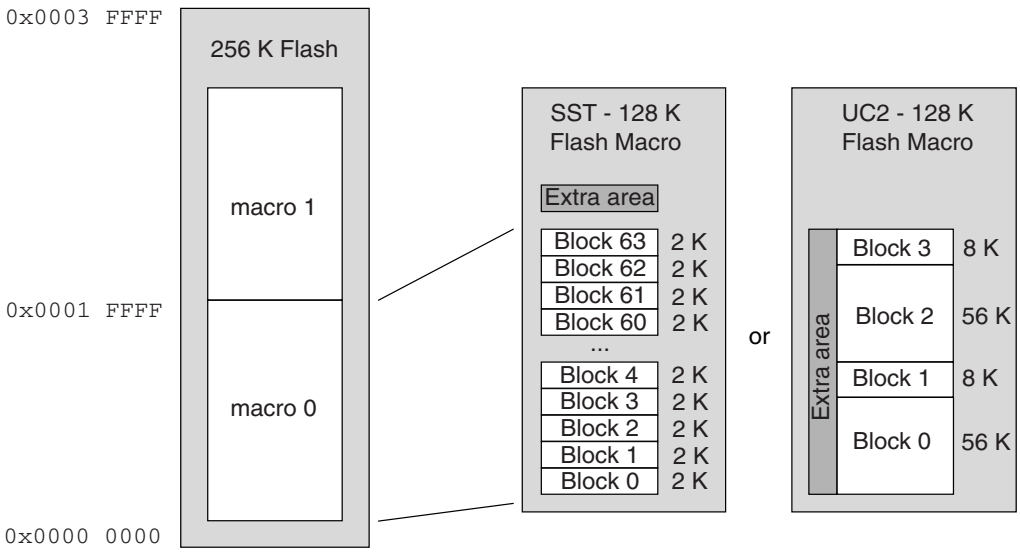
4.1.2 Flash block structures

Depending on the specification, design and technology NEC's embedded Flash microcontrollers contain a certain number of Flash macros with a certain size.

Each Flash macro consists of several blocks and may contain one or more extra areas.
The blocks contain the user data. The extra area contains security and protection related information (see 4.2 "Flash Protection" on page 18 and 4.3 "Security" on page 21), so called *ExtraData*. The extra area is not accessible by normal read operations. Writing of the extra area contents is done by special *SelfLib* commands which are part of the different reprogramming sequences. Handling of this area is transparent for the user.

Depending on the Flash technology (e.g. UC2 or Super Flash®, licensed by SST) the block size differs from 2 K up to 120 K.

Figure 4-1: Embedded Flash Configuration Samples



4.2 Flash Protection

4.2.1 Protection strategy

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

The two major items to be considered in the protection concept are:

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

In the following the strategies regarding these items is described in detail.

(1) Illegal read-out of the 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 interfaces*
  Some devices contain the NEC's N-Wire 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. Please refer to the *DeviceInfo* to check, whether the protected N-Wire interface is available. As the protection is not directly related to a Flash feature, it is mentioned, but not described here. Please refer to the device users manual or to the NEC N-Wire tools description for that.
- *Direct read-out via programming interface*
  The standard programming interface (e.g. for PG-FP4) supports a command to read out the Flash contents on all current devices (Earlier UC2 devices don't support the command. Please check the *DeviceInfo)*. This feature helps a lot in the development and debugging phase and for failure evaluations. This command can be disabled by a protection flag (See 4.2.2 "Protection configuration settings" on page 20).
- *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 the Flash in two ways:
  - By the application itself using self-programming
    Please ensure, that this possibility is not implemented or protected in your application (e.g. Password mechanism).
  - By the programmer interface
    In order to disable this, the Flash Write and the Flash Block Erase commands can be disabled (See 4.2.2 "Protection configuration settings" on page 20). 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.

(2) Illegal or accidental reprogramming of the 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. NEC V850 devices provide features even for that (See *device information* to check the individually supported features):

- *Partly reprogramming by the programmer interface*
  See “illegal read-out of the Flash contents”
- *Complete reprogramming by the programmer interface*
  If also the complete erasing and reprogramming by this interface shall be disabled, in addition to the Flash Write and the Block Erase commands also the Chip Erase command can be disabled (See 4.2.2 ”Protection configuration settings” on page 20). 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 the self-programming*
  It is also possible to disable reprogramming parts or the complete Flash via the application (See 4.2.2 ”Protection configuration settings” on page 20). This protection is block wise organized, starting from 0x0000000. So it is possible to protect e.g. a Bootloader or more code and data up to the complete application.

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

4.2.2 Protection configuration settings

This chapter explains the protection relevant settings (Part of the *ExtraData*) and mechanisms, implemented in NEC embedded Flash devices (See *DeviceInfo* to check, which features are supported by which device).
For the usage of these settings and the protection strategy, please refer to section 4.2.1 ”Protection strategy” on page 18.

The protection configuration can be set by the dedicated Flash programmers, like PG-FP4 or via self-programming (In UC2 not change by self-programming).

Note: If set once, resetting is only possible by the Chip Erase using a dedicated Flash programmer (SST Flash and UC2 Flash). Furthermore, in UC2 Flash the protection settings are made invalid by erasing certain combinations of or all lower 4 Flash blocks (See 5.3 ”Extra Information Handling” on page 29).

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.
- *Program command disable (Programmer interface)*
  Writing to Flash via programming interface is disabled. It does not affect self-programming.
  The Flag is valid for the complete Flash.
- *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.
- *Chip Erase command disable (Programmer interface)*
  Erasing the complete device via programming interface is disabled. It does not affect self-programming
  The Flag is valid for the complete Flash.
- *Boot Cluster size definition (SST Flash devices only)*
  Boot cluster size information is relevant for the boot cluster protection (see below)
- *Boot Cluster Protection (SST Flash devices only)*
  If set, erasing and writing on the Flash by the application using the self-programming is disabled for the boot cluster.

Note: The Boot Cluster size determines just the number of blocks affected by the Boot Cluster Protection. Different from that, the later explained Bootswap mechanism affects a fix number of blocks. This number of blocks is device dependent and can be found in the *DeviceInfo.*

4.3 Security

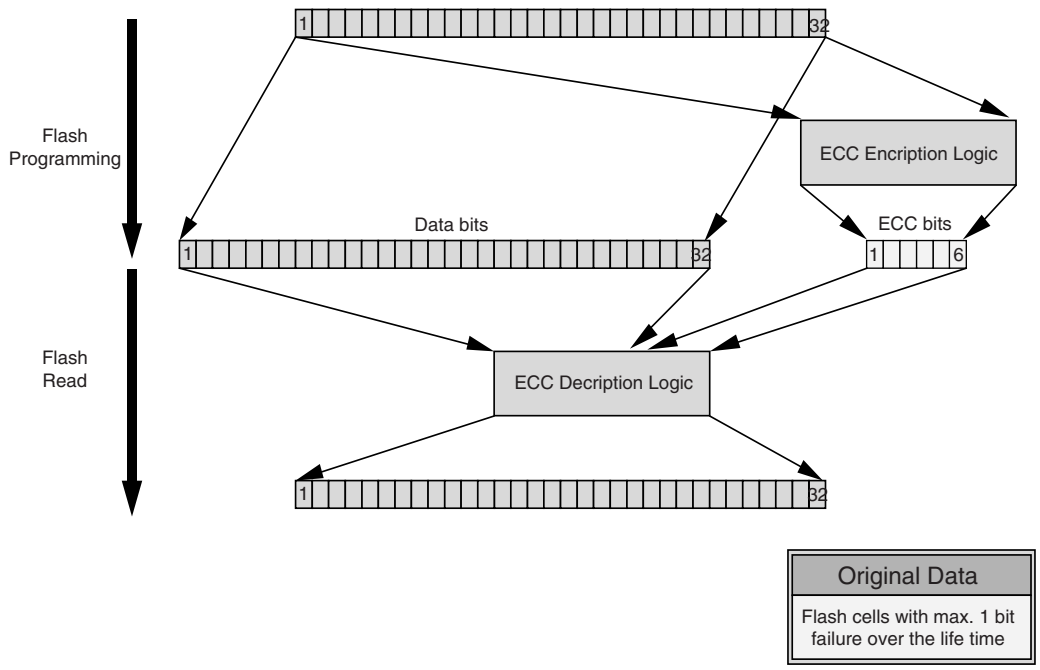
NEC's Flash devices are equipped with dedicated security features. The features have to be separated for normal operation, where data retention is important and for reprogramming, where secure reprogramming in case of power fail or other problems is important.

4.3.1 Normal operation (Error Correction Circuit - ECC)

NEC's Flash devices contain Error Correction Circuits (ECC) to provide correct Flash data even in case of a single bit failure in the Flash cells.

ECC bases on the fact, that beside the Flash data redundant ECC data is written into additional Flash cells. This data is then used to correct 1 bit failure per word.

Figure 4-2: ECC Functionality



During the write procedure the data to be written is encrypted in an ECC encryption matrix and the resulting additional bit are written to additional Flash cells too. When reading during normal operation the data word and the additional ECC bits are put into a decrypting matrix. The result of this matrix is the original data word. One of the data or ECC bits might be wrong without affecting the correctness of the resulting data word. ECC is an on-line method. That means, that from user point of view ECC has NO impact on the data read performance.

4.3.2 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 the reprogramming process goes wrong, it must be possible to keep a 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/reset.

Two major options with different advantages and disadvantages have to be considered. Depending on the application, bootloader and Flash technology (Flash block sizes) the appropriate solution has to be selected:

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

4.3.3 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 no data, that needs to be reprogrammed using self-programming, is located in 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 a part of the bootloader is temporarily not available. Especially in case of big sized Flash blocks (UC2) use of this method needs to be considered carefully, as it might result in waste of Flash memory, if the bootloader does not occupy a complete Flash block.

On the other hand handling of this method is easy, as just one or more Flash blocks (application program/data) are reprogrammed in a standard procedure.
If blocks with an attached extra area are affected (Blocks 0~3 in UC2 Flash devices), the secure self-programming with bootloader update procedure is required, if any *ExtraData* has been set!

Furthermore, increased safety by protection against reprogramming the bootloader due to program failures is possible (SST flash only). The block protection feature (See section 4.2.2 ”Protection configuration settings” on page 20) 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.

4.3.4 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. This may be necessary if only few big blocks are available.

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. NEC provides the Boot Swap functionality for that.
Boot swap means, that two Flash blocks or clusters of blocks can be swapped in the address range. This swapping is done depending on *Boot Swap* bits, that can be set in the corresponding Flash extra areas. When a valid bootloader is contained in the corresponding block, the bits are set accordingly and 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 also a block containing a bootloader can be updated securely.

4.4 Simultaneous Operation During Self-Programming

The different self-programming steps, especially Erase, but also Write, Internal Verify,... consume time. During that time it is usually not possible to stop the application, at least basic functionality is required. As the at least the Flash area, that is reprogrammed, is not available during self-programming, special provisions need to be taken.
Depending on the application requirements, different simultaneous operation scenarios are possible:

- Stop application
- Execute in RAM
- Dual operation

4.4.1 Stop application

When doing the self-programming steps the application is simply stopped. No operations, even no simple ones like Watchdog triggering are possible that time.
Disadvantage is, that the time for some self-programming operations is quite long. Especially the Erase procedure may need several seconds, depending on the device, technology and block structure.

4.4.2 Execute in RAM or external memory

User applications can be executed in RAM (Or external memory, if available) during execution of the self-programming functions. Additional software overhead to copy the functions to internal RAM or external memory has to be considered.
Supported is given in two basic ways, which can be used even in parallel. The options are:

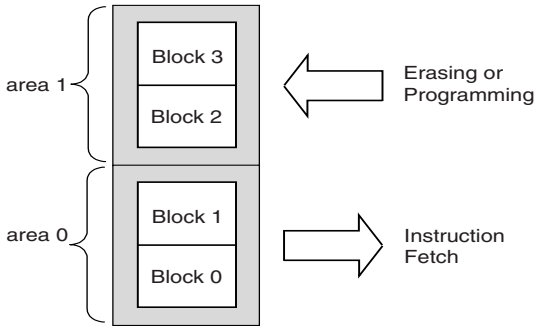
- Interrupts
  Using interrupts to execute user functions, while self-programming.
- Polling (Not SST (CZ6HSF / UX4))
  Initiate background self-programming functions and continue the user application, which then polls the end of the user application.

For both options latencies have to be considered regarding user function execution. The latency for interrupt functions is in the range of several hundred microseconds. The polling latency can be much longer, as some library functions cannot be executed in background by a dedicated hardware.
Please also refer to chapters "User Application Execution During Self-Programming" for the different technologies.
The device dependent latency times are quoted in the *DeviceInfo*.

4.4.3 Dual operation

Execute code in one Flash area while do self-programming on another area.
Normally during self-programming the complete Flash is not available. However, when at least two Flash areas (Macros) are on the device, dual operation could be implemented.
Anyhow, the hardware effort required for this feature exceeds other features effort by far. Therefore, dual operation on Code Flash only is not supported.
Anyhow, if devices contain Data Flash, code can be executed on Data Flash while re-programming Code Flash.

Figure 4-3: Dual Operation Flash



4.5 Other Features

4.5.1 Variable reset vector

Most devices support configuration of the address of the first instruction executed after reset, the so called variable reset vector.

The configuration is also part of the *ExtraData* and the handling of the configuration is the same as for the protection flags. Configuration is possible either by a dedicated Flash programmer or by a SelfLib instruction.

By the variable reset vector it is possible to place a bootloader in any Flash block of the device and so to decouple the bootloader from the interrupt vector table, which is located in the 1st Flash block and starts 0x00000000. So, the interrupt vector table can become an integral part of the application instead of the bootloader.
This requires, that no secure bootloader update is required (Secure bootloader update requires the boot swap mechanism, which uses the 1st Flash blocks) and that the bootloader does not use interrupts.

Note: The variable reset vector only determines the program start after reset. The interrupt vector table is not affected. It is always located at 0x00000000.

[MEMO]

Chapter 5 UC2 - Self-Programming

5.1 Device Reprogramming Overview

First of all it is important to select the correct reprogramming flow. Please check with respect to your application, whether secure reprogramming and secure bootloader update is required. The *DeviceInfo* contains the information, if the boot swap feature (required for secure bootloader update) is supported (See section 4.3.2 ”Secure reprogramming using self-programming” on page 22).
If secure bootloader update is required, please refer to 5.4 ”Secure Reprogramming Flow with Bootloader Update” on page 30 for the correct flow.

In the following, the basic reprogramming steps are explained. Standard block reprogramming and the secure bootloader reprogramming is essential part of the next sub-chapters. The standard block reprogramming flow is also embedded in the secure bootloader reprogramming.

The following principle steps have to be executed for any device reprogramming:

Table 5-1: Basic Steps

Step	Function	Description
1	Setup the user program	Setup communication Interface, watchdog timer, FLMD0 voltage activation,...
2	Self-Programming initialisation	See section 7.1 ”Initialisation” on page 49.
3	Reprogram the Flash	Take care to keep the correct sequence for reprogramming single Flash blocks (see section 5.2 ”Basic Block Reprogramming Flow” on page 28) and for secure bootloader reprogramming (see section 5.4 ”Secure Reprogramming Flow with Bootloader Update” on page 30).
4	End Reprogramming	Return to bootloader, start application, reset or...

Note: Especially for the first four blocks these basic steps need to be executed on one block completely before touching the next block. Only so the integrity of the extra information, like boot swap flag, protection flags,... is ensured.

5.2 Basic Block Reprogramming Flow

This is the standard flow, that shall be used to reprogram all Flash blocks.

Step four is required for the first four blocks, when either secure bootloader update (See 5.4 ”Secure Reprogramming Flow with Bootloader Update” on page 30) is intended or if protection flags are set (See 5.3 ”Extra Information Handling” on page 29).

To ensure, that the data is written correctly and the data retention is given it is important to keep the following sequence for block reprogramming. Whenever a function returns an error, you may not continue the sequence because the next steps may not work correctly and the result is undefined.

Table 5-2: Block Reprogramming Sequence

Step	Flash Lib. Function	Description
1	Block Erase	The Flash block is erased
2	Write	Data is written to the Flash
3	Set security/protection flags	Security/protection related flags are set (1st four blocks only, see above)
4	Internal Verify	This step is executed after the last Write to a dedicated block to ensure the specified data retention of the Flash The Internal Verify is a check for higher reliability of the programming, but no manipulation or operation on a 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, Vdd/Gnd bounce)

5.3 Extra Information Handling

The first four blocks are combined with an extra area each. The device internal firmware does a special majority decision from these blocks on start-up in order to decide, if a protection flag is set, or not. An extra area is erased together with the corresponding block. Special care has to be taken with respect to the protection flags.
The flags are set/re-written using the library function 7.4 ”Protection/Safety Related Operations” on page 59.

Setting protection flags of variable reset vector

UC2 Flash protection flags can only be set by a dedicated Flash programmer via the programming interface (e.g. Using the PG-FP4). It is not possible, to change the settings by self-programming (e.g. set or clear the Write protection flag). Following that, the decision for the protection need to be done on virgin programming with the flash programmer.
However, due to the organisation of the UC2 Flash (extra area is erased together with the flash block) it is necessary to re-write the security flags when reprogramming one of the first 4 blocks. Please refer to 5.2 ”Basic Block Reprogramming Flow” on page 28 for the correct sequence.

Note: Re-writing the flags need to be done before reprogramming the next block. By that the reprogramming flow does not enter an intermediate state, where the firmware cannot detect the correct flag setting in case of startup after reprogramming interruption.

Boot swap flag

The boot swapping feature can swap the blocks 0 and 1 with the blocks 2 and 3. All extra areas contain a boot flag. To signal a valid bootloader in the corresponding blocks, the flags in the blocks 0 and 1 respectively in 2 and 3 need to be set both. The blocks with both flags set are swapped to 0x00000000. If all 4 flags are set (e.g. after power fail during self-programming) the blocks with the flags set last are swapped to 0x00000000. If no valid flag pair can be found (e.g. after initial programming), the block 0 is swapped to 0x00000000.

Notes: 1. The boot flag is set by writing it to 0. After a Block Erase it is reset, the value is 1.

2. If the 2nd block (block 1) need to be reprogrammed, the secure reprogramming sequence (See section 5.4 ”Secure Reprogramming Flow with Bootloader Update” on page 30) must be used, if ever a secure reprogramming sequence with boot swap has been done before. This is, because in case of a reprogramming interruption, the majority selection of the boot swap feature does not work correctly, if block 1 does not contain a set boot flag.

5.4 Secure Reprogramming Flow with Bootloader Update

The basic feature allowing this way of secure reprogramming is, that on device start-up the internal firmware checks, which block contains a valid bootloader. To do so the boot swap flags in the extra area(s) are checked.
By the boot swap feature and the correct reprogramming sequence it can be assured, that always a valid boot program is started to continue the reprogramming sequence in case that the reprogramming algorithm was interrupted (Power fail, accidental reset).
The bootloader must be able to control the complete reprogramming sequence including data transfer of the new Flash contents.

The sample configuration considers a device with 8 blocks and with boot swap capability, where the blocks 0/1 can be swapped with 2/3.

Note: The block number in the diagrams below are the physical block numbers. They are always fix, even after block swapping. E.g.: block 2 before boot swapping remains block 2 even after the swap. This is done for better illustration.
When using the SelfLib functions, logical block numbers are used. On logical block numbers the block on address 0x00000000 is always block 0, followed by block 1 and so on. This is valid even after block swapping. E.g.: Block 2 before swapping is block 0 after swapping. By that software development is easier, as the block numbers need not be modified depending on the swapping.

Table 5-3: UC2 Flash Secure Reprogramming Sequence Including Bootloader Update (1/3)

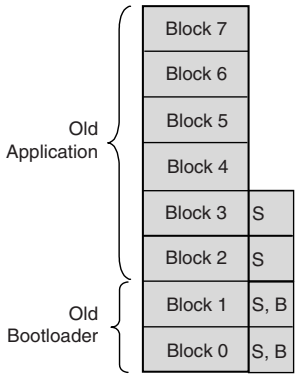
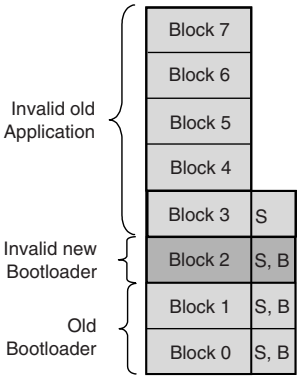
Step	SelfLib Function	Description	Sample Configuration
1	Initial Status	• Boot Program in the lower 2 areas • Safety flags in the extra areas of the blocks 0 and 1 set (e.g. Write disable, Block Erase disable) • Boot flag in the extra areas of the blocks 0 and 1 is set • Blocks 2~7 contain the application	
2	Reprogram block 2	Execute the complete block reprogramming flow (See 5.2 ”Basic Block Reprogramming Flow” on page 28): • Erase • Program • Re-write security flags & set boot flag • Internal Verify When erasing the first application program block the application is no longer valid and able to run. In case of power fail, the bootloader handles the complete reprogramming flow.	

Table 5-3: UC2 Flash Secure Reprogramming Sequence Including Bootloader Update (2/3)

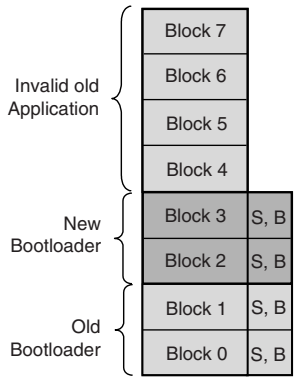
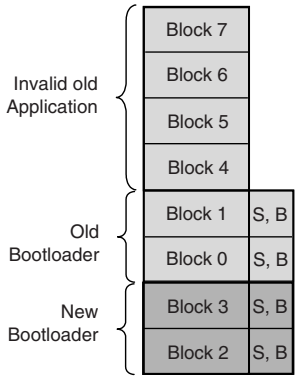
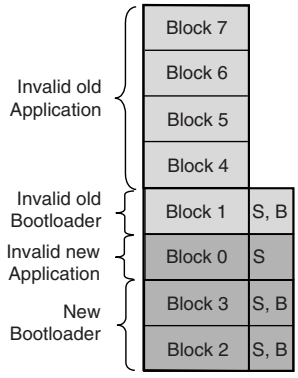
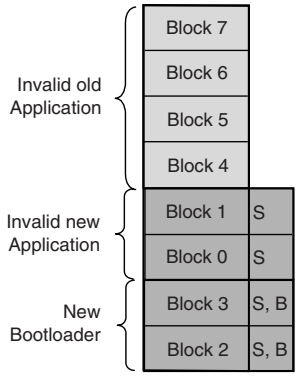
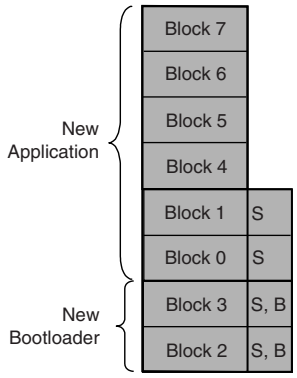
Step	SelfLib Function	Description	Sample Configuration
3	Reprogram block 3	Execute the complete block reprogramming flow (See 5.2 "Basic Block Reprogramming Flow" on page 28): • Erase • Program • Re-write security flags & set boot flag • Internal Verify Now the new bootloader is valid. Even though all boot flags are set the new bootloader is mapped to 0x00000000 in case of a Reset, as a sequential number is also stored in the extra area	
4	Swap blocks 0/ 1 and 2/3	After swapping the blocks, the begin of the new bootloader (block 2) is located at 0x00000000	
5	Reprogram block 0	Execute the complete block reprogramming flow (See 5.2 "Basic Block Reprogramming Flow" on page 28): • Erase • Program • Re-write security flags • Internal Verify Note: Pass the logical block numbers (not physical block numbers mentioned in the diagram on the right side) as parameters to the SelfLib functions requiring block numbers!	
6	Reprogram block 1	Execute the complete block reprogramming flow (See 5.2 "Basic Block Reprogramming Flow" on page 28): • Erase • Program • Re-write security flags • Internal Verify Note: Pass the logical block numbers (not physical block numbers mentioned in the diagram on the right side) as parameters to the SelfLib functions requiring block numbers!	

Table 5-3: UC2 Flash Secure Reprogramming Sequence Including Bootloader Update (3/3)

Step	SelfLib Function	Description	Sample Configuration
7	Reprogram the blocks 4~7	Execute the complete block reprogramming flow for each block separately, without setting security flags (See 5.2 ”Basic Block Reprogramming Flow” on page 28): • Erase • Program • Internal Verify	

5.5 Library Sections

In order to support the different environments and reprogramming sequences, the library is splitted up into several sections. These sections have to be linked differently according to the execution environment of the library and the reprogramming program (See section 5.6 ”Flash Environment” on page 34).

The following library sections exist:

- SelfLib_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 initialisation is part of this section.
- SelfLib_RomOrRam
  This section contains the user interface. Depending on the reprogramming sequence (See section 5.6 ”Flash Environment” on page 34) this section is executed in RAM or in Flash. If executed in RAM, the copy procedure from Flash to RAM is automatically executed by the library initialisation (See section 7.1.1 ”Library initialization” on page 49).
  Especially in the Time saving sequence (See section 5.7.2 ”Time saving reprogramming sequence” on page 37) the user self-programming control function should also be located in this section. This ensures, that the function is copied automatically to the RAM by the SelfLib initialisation.
- SelfLib_ToRam
  This section always needs to be executed outside the Flash. As it is usually copied to the internal RAM, it is called SelfLib_ToRam. The sections contains the device Firmware interface.
- SelfLib_RAM
  This section contains the variables required for SelfLib. It can be located to internal or external RAM.
- SelfLib_UsrIntToRam
  This section contains the Interrupt functions. If interrupts are not used, the section is empty but should exist, as address references to this section are used in the SelfLib initialisation.
  If used, the section is copied to the RAM as the first SelfLib section.
  If the standard interrupt handling procedure is supported by the device, the library sections to be copied to RAM must be copied to the begin of the devices internal RAM (See section (1)”Fix address jump” on page 42). If the registered interrupt handling is supported, the address may be everywhere in the RAM.

5.6 Flash Environment

Several things need to be considered when working with the *SelfLib.* An important item is, that the **complete** Flash is **not accessible** at all during certain steps of reprogramming.
This depends on the status of the so called *Flash environment*, which is device internal hardware (charge pumps, sequencer,...), required to do any Flash programming. It is not continuously activated, only for reprogramming. The Flash is not accessible when the Flash reprogramming environment is active.

Flash environment activation and deactivation is handled by the *SelfLib*. Especially deactivation is time consuming. In order to archive fast reprogramming the environment should be kept activated during the whole reprogramming. On the other hand, during activated environment the program execution may not be done from the 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 (firmware interface) are executed from internal RAM.
Following that, 3 major scenarios can be considered for Flash self-programming, that are all supported by the SelfLib:

- *External memory sequence*
  The complete application must be located in external memory or internal RAM. As environment activation/deactivation is required only once, the sequence is time saving. However, due to the internal memory limitation either only a small application in iRAM is possible or bigger external memory is necessary.
- *Time saving reprogramming sequence*
  Most parts of the self-programming code are executed in the internal RAM, including the reprogramming control functions. Only basic application and initialisation and de-initialisation of the self-programming are done in the embedded Flash.
- *RAM saving reprogramming sequence*
  Only a small part of the SelfLib is executed in the internal RAM, the rest is executed in the embedded Flash. Frequent activation and deactivation of the *Flash environment* is necessary.

In the following chapter, these sequences are explained in detail.

5.7 Reprogramming Scenarios

The SelfLib contains all functions which are important for self-programming except the control program which is application specific and therefore has to be made by the user:

- *Flash environment* Activation / DeActivation
- User interface
  This contains the functions to be called by the Flash reprogramming control program, which has to be written by the user
- Firmware interface

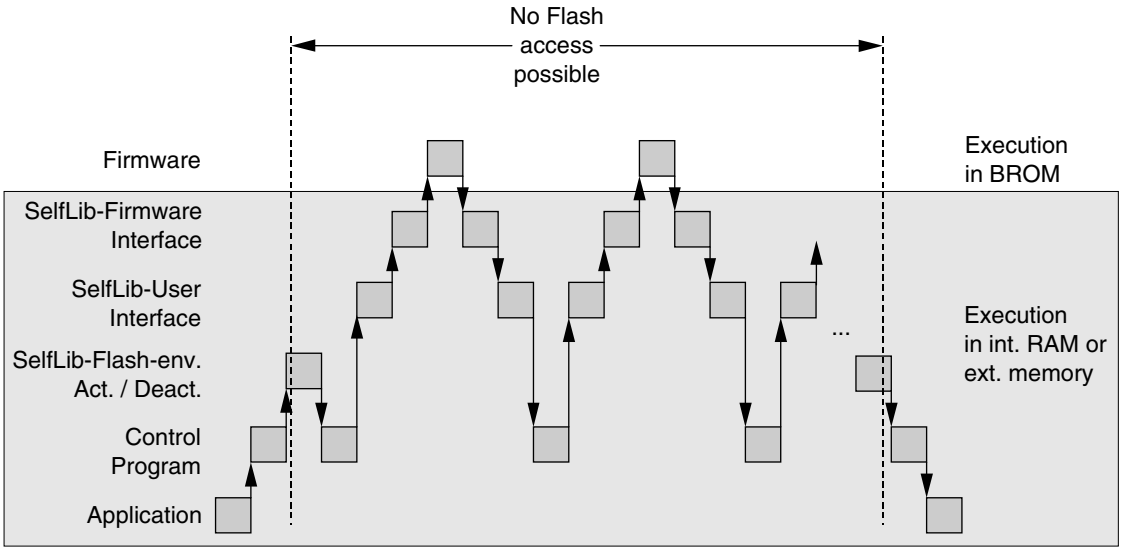
This functions are called in the different scenarios under different Flash related conditions and on different locations.

The device internal reprogramming firmware is called via the firmware interface of the SelfLib. The firmware is located in a mask ROM, called BROM. This firmware contains the actual Flash operations which are tested and evaluated by NEC.

5.7.1 External memory sequence

The application, containing the SelfLib, is located in internal RAM or external Memory. All program code is permanently available (Except interrupt vector table) at the linked location, even when the *Flash environment* is activated.
Due to that the *Flash environment* can be activated during the complete reprogramming and therefore the reprogramming sequence is time saving.

Figure 5-1: External Memory Reprogramming Sequence



It does not matter, if the program is linked to the destination address (position dependent) and stored there or if it is linked to another address (position independent code) and just copied there (e.g. via a serial interface).

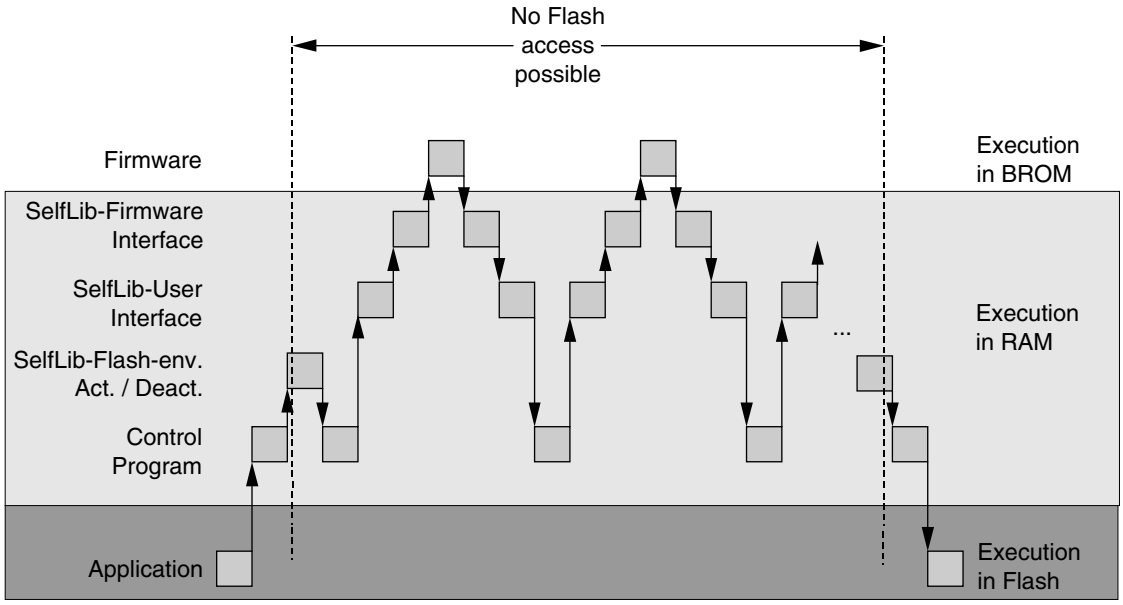
Library handling
As the memory location of the application is permanently available during self-programming, nothing need to be copied to a “save” location, but everything can be executed, where it is.
No copy action by the SelfLib initialisation is required (See section 7.1.1 ”Library initialization” on page 49).

5.7.2 Time saving reprogramming sequence

The *Flash environment* activation is time consuming. 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.

This sequence is best for devices with bigger internal RAM. Otherwise small and simple communication routines, interrupt routines, etc. have to be used.

Figure 5-2: Basic Time Saving Reprogramming Sequence

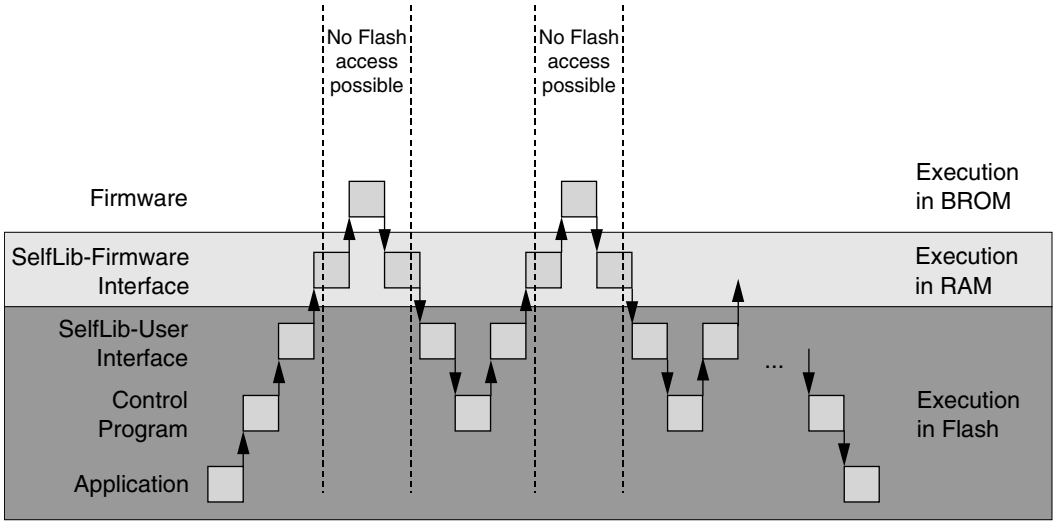


Library handling
The application, including the control program and the SelfLib are located in the internal Flash.
As the memory location of the application is not permanently available during self-programming, the user interface, the firmware interface and the self-programming control program are copied to a “save” location. This may be the internal RAM, but also external RAM, if available, is acceptable.
The *Flash environment* activation and deactivation need to be done explicitly by the control program (done by the user).
Beside SelfLib_UsrIntToRam, the SelfLib_ToRAM and the SelfLib_RomOrRam sections need to be copied. This is automatically done by the SelfLib initialisation routine (See section 7.1.1 ”Library initialization” on page 49).
When the self-programming control function including all called sub-functions is also located in the SelfLib_RomOrRam, this function is also copied there. NEC recommends to put the control function into this section as by that the SelfLib user functions can be called directly without function pointers and address recalculation.
For accessing the copied functions from outside the copied parts, please refer to the function SelfLib_FunctionNewAddress (See section 7.1.2 ”New function address” on page 52).

5.7.3 RAM saving reprogramming sequence

The RAM saving sequence allows execution of most of the software in the Flash. This is only possible, if the reprogramming environment is deactivated during program execution. The result is frequently done environment activation and deactivation and so increased reprogramming time.

Figure 5-3: Basic RAM Saving Reprogramming Sequence



This sequence is best for memory consuming control programs in applications without external memory. Less internal RAM is used as only the device firmware interface and interrupt routines need to be executed in RAM.

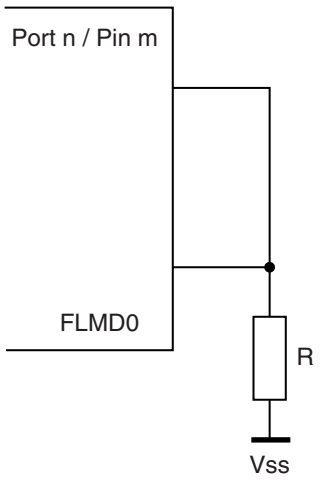
Library handling
The application, including the control program and the SelfLib are located in the internal Flash.
As the memory location of the application is not permanently available during self-programming, the firmware interface (In SelfLib_ToRAM) is copied to a “save” location. This may be the internal RAM, but also external RAM, if available, is acceptable.
It is automatically done by the SelfLib initialisation routine (See section 7.1.1 ”Library initialization” on page 49).

The *Flash environment* activation and deactivation is done implicitly by the firmware interface functions. Please do not call the environment activation/deactivation function supplied by the SelfLib. Please also refer to the SelfLib initialisation function (See section 7.1.1 ”Library initialization” on page 49).

5.8 Hardware Requirements

The requirements of self-programming regarding the device external hardware is very low as no dedicated external programming voltage is required.
However, it has to be considered, that a security concept against unintended reprogramming has been set-up. It is required, that the devices FLMD0 pin is externally attached to V_{DD} (I/O voltage) during self-programming. FLMD0 should be attached to V_{DD} immediately before or after SelfLib initialisation (See ODO) and reset to V_{SS} at the end of self-programming. Default state on reset is V_{SS}.

Figure 5-4: 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}.

5.9 User Application Execution During Self-Programming

The standard way to set-up a reprogramming flow is, that a self-programming control program controls the complete reprogramming flow and the SelfLib calls. The SelfLib functions interact with the Flash. According to the called SelfLib function the execution requires times from a few nanoseconds up to several hundreds of milliseconds. In many applications it is acceptable, that no further software execution during that time is possible. This is the easiest way of a reprogramming sequence.
However, for most applications it is not acceptable, that no more code can be executed during execution of long lasting SelfLib functions (like the Erase function).

The NEC single voltage Flash devices are designed in a way, that Flash manipulation operations only have to be initiated and then are executed in the background. Only completion of the operations need to be checked in order to continue the self-programming. By that, in order to support application software execution during self-programming the SelfLib can be set-up to offer two major options, if necessary both in parallel:

- Interrupt function execution
- Execution status polling

The next sub-chapters will explain the two options and discuss the execution latencies which have to be considered.

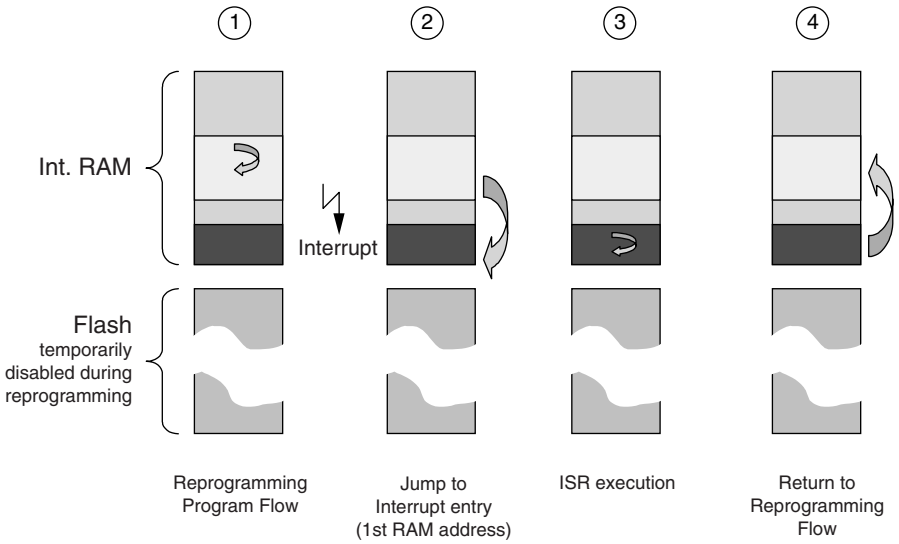
5.9.1 Interrupts

This sub-chapter explains the available Interrupt function options.
The interrupt vector table is not accessible when the *Flash environment* is activated. Following that normal jumps into the vector table are not possible during that time.

Therefore, the execution of interrupt functions in RAM or external memory is supported in a different way.

Note: Special interrupt handling is done only while the *Flash environment* is activated. While not activated the normal user defined interrupt vector table is used.

Figure 5-5: Interrupt Routine Execution in Internal RAM / External Memory



To enable interrupt handling, two different approaches have been implemented into the different devices. Depending on the devices either one of the approaches is implemented or both. If both approaches are supported, the selection of the used option is done by the function SelfLib_RegisterInt (See section 7.1.3 ”Register interrupts” on page 53).

- Fix address jump
- Registered interrupts

Please refer to the *device information* for the handling approach implemented into your device.

As the interrupts may not be executed in the embedded Flash, they shall be executed either in the internal RAM or in external memory. If executed in the internal RAM, it is recommended to place all functions into the SelfLib_UsrIntToRam section, that is automatically copied into the RAM by the *SelfLib initialisation* (See section 7.1.1 ”Library initialization” on page 49).

(1) Fix address jump

All interrupt vectors are relocated to one entry point in the internal RAM:

- New entry point of **all non maskable interrupts** is the 1st address of the internal RAM (depending on the device, e.g. 0xFFFFE000). A handler routine must check the interrupt source. The source can be read from the exception cause register ECR (See device users manual).
- New entry point of **all maskable** interrupts is the word address following the non maskable interrupt entry (depending on the device, e.g. 0xFFFFE004). A handler routine must check the interrupt source. The source can be read from the exception cause register ECR (See device users manual).

Notes: 1. If the handler routine is located in the SelfLib_UsrIntToRam section, please take care that the destination address of this section is the 1st RAM address! If automatically copied by the function SelfLib_Init, the parameter *destAddSelfLib* must be the 1st RAM address. Furthermore, the handler routine must be located at the beginning of this section.

2. The handler routine needs to be compiled as an interrupt function (i.e. terminate with a RETI instruction, save/restore all used registers,...)

(2) Registered interrupts

Interrupt handler routines to be executed on registered must be written as normal c functions (no RETI instruction at the end), however need to save and restore all used registers. As this issue is not supported by the c-compiler, a function prologue and epilogue in assembler is necessary (See the sample program provided with the SelfLib).

The functions must be registered for the firmware (See section 7.1.3 ”Register interrupts” on page 53) before activating the *Flash environment*. By that the interrupt functions are known by the firmware and then called on interrupt request, when the *Flash environment* is activated.

5.9.2 Status polling

Most SelfLib functions with long execution times are Flash modification functions (e.g. Erase, Blank Check,...). The basic flow of these functions is, that a *Flash environment* background operation is initiated and then the status of the operation is polled. This status also returns the result of the background operation.
Regarding handling of the status requests two options (Modes) can be used.

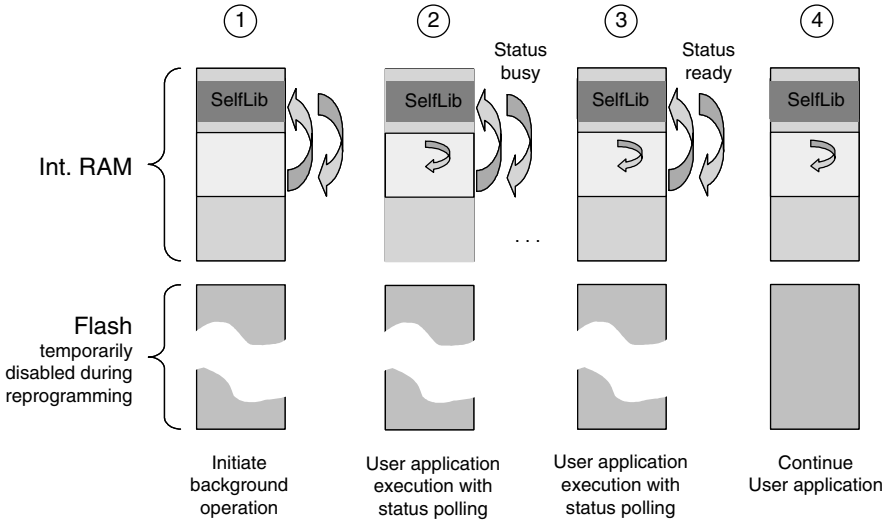
(1) Interrupt mode

In the Interrupt mode the status polling is done implicitly by the SelfLib functions. The SelfLib functions are called and return after the background operation is finished. In that case these functions can only be left by interrupts for user application execution. In this mode the reprogramming control program is smaller and the status polling by the SelfLib is well tested.

(2) Polling mode

In the polling mode the SelfLib functions (e.g. SelfLib_Erase) only initiate the background function and return then to the user application, controlling the reprogramming. The user application does the status polling and can meanwhile execute user code. As the polling is not time critical, no special care has to be taken regarding execution time while polling the status.
The status polling is shifted from the well tested SelfLib to the user application.

Figure 5-6: Application Flow in Polling Mode



In the SelfLib initialization (See section 7.1.1 ”Library initialization” on page 49) the user can select, if the polling is done implicitly in the library or if the polling is done in the user application.

Note: Depending on the selected option, the reprogramming program looks different. In the device dependent application sample programs provided by NEC on the internet, both options are implemented. Selection is done by a global define in the target.h. This is USER_APPLICATION.

5.9.3 Execution latencies

The execution latencies for interrupt routines and other user routines differ quite a lot and therefore have to be considered separately. In both cases the latency for most of the functions is quite low (below 100us). However, for a time critical application (e.g. triggering a watchdog) the worst case latency is of interest. Following that, we have measured the maximum latency for a complete reprogramming flow at the most common frequencies, using all SelfLib functions. The measured values can be found in the *device information.*

(1) Interrupt routines

The execution latency of interrupt routines is quite low. The longest latency is caused by the SelfLib functions *Flash environment* activation/deactivation (see section 7.2 ”Flash Environment” on page 54).
Other routines do not disable interrupts for a longer time.

(2) Polling mode

In polling mode the latency for execution of user functions is determined by the run time of the SelfLib functions. Depending on the device hardware, Flash technology and firmware implementation the execution times may differ from device to device. The worst execution time can be expected in the *Flash environment* activation/deactivation or in the secure flag handling routines SelfLib_GetInfo_GetSecFlags/SelfLib_SetSecFlags (See section 7.4 ”Protection/Safety Related Operations” on page 59).

Chapter 6 UC2 - SelfLib Configuration

SelfLib implementation relevant issues are discussed in the following sub-chapters.

6.1 Used Resources

The only resources used by the self-programming, is memory.

Beside the SelfLib sections the stack is used. The stack usage is depending on the library and on the used compiler environment. Furthermore, also the device firmware uses the stack.

The complete FLASH is not available at time during self-programming.

6.2 Software Considerations

This chapter lists up special issues to be considered when implementing self-programming into the application.

DMA operation during self-programming
DMA operations during self-programming (While the Flash environment is activated) are not allowed at all. Background is the fact, that accesses sequences to protected SFR registers may fail in case of DMA accesses to the peripheral bus during this sequence

Run-Time Library calls
During Self-Programming (activated Flash Environment) It is not possible to call run-time library functions, if the library is not copied to RAM together with the Self-Programming code.
In case of program failures during self-programming, the code should be explicitly checked for such calls.

Switch instruction in C
In case of the user status check, the switch instruction should be avoided in all the code executed in RAM. Depending on the used compiler this instruction may result in a call to a run-time library, which has not been copied.

6.3 Compiler Configuration

As the SelfLib is delivered in source code, the user has to take care for the correct compiler settings in his applications. Due to the huge amount of setting options, not all could be tested. The settings coming along with the application samples should be taken as reference for the user application development.

6.3.1 Project settings

The most important settings for a successful application build shall be explained in the following:

Position independent code (PIC)
The SelfLib code shall be compiled position independent.

Inline function prologue and epilogue
In order to avoid any library calls for function prologue and epilogue into a section, not available during self-programming (Flash), they need to be inlined.

V850E core instruction set
The extended instruction set of the V850E core is used for the assembler parts. So this core should be selected.

The settings are only required for the complete SelfLib and all code added to the SelfLib sections. The rest of the project may have different settings. However, it is recommended to use the same settings project wide, if possible.

Green Hills (GHS) Multi 2000

To set the settings project wide, click on the project file in the project window to select it before entering the menu.

- Select position independent code
- Select inline prologue
- Select the processor type "V850E1" resp. "V850E/MS1" for V850E core instruction set

Further required settings:

- Set the following driver options for the assembler files to ensure, that c-style preprocessing directives are processed:
  Click on an assembler file to select it. Then select the menu “Project -> File options”. In the driver options field type in “-preprocess_assembly_files”. Do this for each assembler file.
- Disable usage of the **callt** instruction, as it results in calls to the run-time library, which is not copied into the RAM in case of time saving scenario.

IAR Embedded Workbench

Using the IDE, the settings can be found in the menu “Project -> Options”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- Category ICCV850: Optimize for speed. By that, the function prologue/epilogue is inlined
- Category general: Select CPU variant V850ES or V850E for the V850E core instruction set

No explicit setting required for position independent code.

NEC CA850

Using the IDE, the following can be configured in the menu “Tool -> Compiler Options” under the tab “Others”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- In the box "Any Option" type in “-Ot” for inline prologue/Epilogue

Using the IDE, the following can be configured in the menu “Tool -> Linker Options” under the tab “Library”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- De-select "Link standard library" and de-select "link mathematics library"

No explicit setting required for position independent code.
The core selection is done during project initialization by selecting the correct target device.

[MEMO]

Chapter 7 UC2 - SelfLib API

The SelfLib contains functions of several different categories. These are called by the user program in the sequences as described in the last chapters. The function calls and parameters are explained in the following.

7.1 Initialisation

Functions explained in this chapter are called at the beginning of the self-programming. They are necessary for the SelfLib initialisation, but some functions can also be used for set-up of a reprogramming control program.

7.1.1 Library initialization

Library call:

void SelfLib_Init (	void *destAddSelfLib, u32 frequency, REPR_SCENARIO reprogScenario, USR_APP usrApp)

Required parameters:

destAddSelfLib	The *Relocated Sections* are copied to this location. Depending on the reprogramming scenario (see below) these are SelfLib_UsrIntToRam, SelfLib_ToRam and SelfLibRomOrRam The value is irrelevant and may be set to 0 only, if the external memory scenario is used (See section 5.7.1 ”External memory sequence” on page 36) and the SelfLib_UsrIntToRam section is empty.

Caution: If the device supports the fix address jump interrupt handling (See section (1)”Fix address jump” on page 42), the address must be set to the beginning of the internal RAM (e.g. 0xFFFFE000), as the interrupt handler entry is there.

frequency	The operation frequency of the device (Hz) e.g.: crystal 5 MHz, internal PLL with the factor 10: frequency=50,000,000
reprogScenario	The following values are defined in the file selflib.h (Please refer to section 5.7 ”Reprogramming Scenarios” on page 35 for the explanation of the values):
EXTERNAL_MEMORY:	External memory scenario is used. Just the section SelfLib_UsrIntToRam is copied to the address destAddSelfLib. If empty, no copy action is executed
TIME_SAVING:	The SelfLib_ToRam, SelfLib_RomOrRam and the SelfLib_UsrIntToRam section are copied to the address destAddSelfLib
RAM_SAVING:	The SelfLib_ToRam, SelfLib_RomOrRam and the SelfLib_UsrIntToRam section are copied to the address destAddSelfLib.

usrApp	The following values are defined in the file selflib.h (Please refer to 5.9 ”User Application Execution During Self-Programming” on page 40 for the explanation of the values):
POLLING:	The user control program need to poll the self-programming status
INTERRUPT:	The SelfLib polls the self-programming status Please also refer to the other functions description to check, whether status polling is required or not

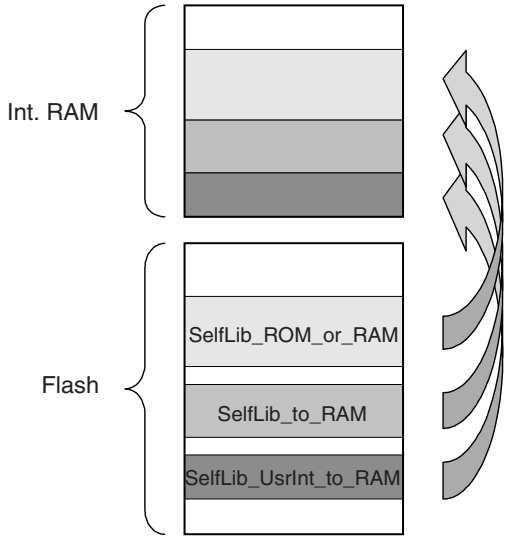
Returned value:

-

Function description:

The function copies the sections into the RAM if requested (see above).

Figure 7-1: Section Copy in SelfLib Initialisation



The SelfLib_UsrInt_to_RAM section is copied first (to destAddSelfLib), in order to enable the user to place it on a defined address for easy interrupt handling support (See section (1)”Fix address jump” on page 42).
Directly behind that SelfLib_to_RAM is copied, followed by SelfLib_ROM_or_RAM. The copy routine doesn’t leave any gaps between the copied sections.

In addition to copying the sections, the SelfLib data and internal pointers are initialised.

Function call sample:

The following function call samples shall explain the usage of the SelfLib_Init function. Depending on the application one of the samples may be used. The parameters have to be adapted to the user application.

(1) The program to control the reprogramming (e.g. bootloader or monitor program) including the library is located in the device internal flash. The RAM saving reprogramming sequence is selected (See section 5.7.3 "RAM saving reprogramming sequence" on page 38). To do Flash programming the *Selflib_ToRAM* section needs to be copied to internal RAM or external memory.

```
SelfLib_Init(   SELFLIB_DEST,     /* The copy destination address */
                FREQUENCY,        /* Operation frequency */
                RAM_SAVING);      /* Select the correct reprogramming scenario */
```

(2) The program to control the reprogramming (e.g. bootloader or monitor program) including the library is located in the device internal flash. The fast reprogramming sequence is selected (See section 5.7.2 "Time saving reprogramming sequence" on page 37). To do Flash programming the Selflib_ToRAM and SelfLib_RomOrRam sections need to be copied to internal RAM or external memory.

```
SelfLib_Init(   SELFLIB_DEST,     /* The copy destination address */
                FREQUENCY,        /* Operation frequency */
                TIME_SAVING);     /* Select the correct reprogramming scenario */
```

(3) The program to control the reprogramming (e.g. bootloader or monitor program) including the library is already located in the device internal RAM or external memory. Functions in the SelfLib_ToRAM and SelfLib_RomOrRam section can be executed directly on their original location and need not be copied. No interrupt routines copied to RAM during reprogramming are used

```
SelfLib_Init(   0,                /* All functions are executed at the linked address */
                FREQUENCY,        /* Operation frequency */
                EXTERNAL_MEMORY);/* Select the correct reprogramming scenario */
```

(4) Same as (3), except, that interrupt support is used. SELFLIB_DEST must be the 1st RAM address.

```
SelfLib_Init(   SELFLIB_DEST,     /* All functions are executed at the linked address */
                FREQUENCY,        /* Operation frequency */
                EXTERNAL_MEMORY);/* Select the correct reprogramming scenario */
```

Sample definitions (Depending on the application and the device):

```
#define SELFLIB_DEST   0xFFFFE000        /* e.g. Destination address 0xFFFFE000 */
#define FREQUENCY      16000000          /* e.g. System operation frequency 16MHz */
```

7.1.2 New function address

Library call:

```
void *SelfLib_FktNewAddress (   void *addFct,
                                void *destAddSelfLib )
```

Required parameters:

addFkt	Original Function address.
destAddSelfLib	Destination address of the section SelfLib_RomOrRam. Values see section 7.1.1 ”Library initialization” on page 49

Returned value:

Address of the function inside the SelfLib_RomOrRam or the SelfLib_UsrIntSection on its destination address.

Function description:

If a user function is located in the section SelfLib_RomOrRam (Only TIME_SAVING scenario) or the SelfLib_UsrInt it is copied to a save RAM location by SelfLib_Init. If the function has then to be called from outside of the section where it is located, the normally used relative function calls don’t work any longer.Following that, it has to be called by a function pointer. To set-up the function pointer SelfLib_FktNewAddress returns the new address of the function.

7.1.3 Register interrupts

Library call:

u32 SelfLib_RegisterInt (	u16 handlerAddress, u32 serJobAddress)

Required parameters:

handler Address	Original interrupt table address. Please be reminded, that neither the interrupt control register address, nor the interrupt number is used, but the original interrupt vector address in normal operation mode
serJobAddress	Interrupt handler function addresses. If the handler routine is copied to RAM, the RAM address need to be registered!

Returned value:

SELFLIB_OK	Registering was successful
SELFLIB_ERR_PARAMETER	Error during registering (e.g. Max number of interrupts exceeded)

Function description:

A maximum of ten (10) interrupts and their interrupt handler routines may be registered (See section (2)”Registered interrupts” on page 42).
You may place the routines (And all subroutines!!!) in the SelfLib_UsrIntToRam section that is automatically copied to the RAM (See section 7.1.1 ”Library initialization” on page 49). For calculation of the new address in the RAM, please use the function described in section 7.1.2 ”New function address” on page 52.

Sample:

```
// functions to be registered are intfct1() and intfct2()
err = SelfLib_registerInt(0x260, &intFct1);
err = SelfLib_registerInt(0x340, &intFct2);
```

Note: If both, registered interrupts and fix address jumps are supported (See section 5.9.1 ”Interrupts” on page 41) by the device, this function is used to select one of the methods.
For registered interrupts, call this function as explained before.
For fix address jumps call the function once in the following way:

```
err = SelfLib_registerInt(0xFFFF, 0x00000000);
```

7.2 Flash Environment

The Flash environment functions are only necessary for the *external memory sequence* or the time *saving sequence* (See section 5.7 ”Reprogramming Scenarios” on page 35).

7.2.1 Activate environment

Library call:

u32 SelfLib_FlashEnv_Activate(void)

Required parameters:

-

Returned value:

SELFLIB_OK	Block is blank
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 ”Hardware Requirements” on page 39)

Function description:

This function activates the Flash environment. After activation no normal Flash read access (e.g. instruction fetch) is possible any more.

Note: During execution of this function no interrupts can be served. As the execution time is relatively long (may exceed several hundreds of µs), measures against interrupt overrun have to be taken.

7.2.2 Deactivate environment

Library call:

void SelfLib_FlashEnv_Deactivate(void)

Required parameters:

-

Returned value:

-

Function description:

This function deactivates the Flash environment. After deactivation normal Flash read access (e.g. instruction fetch) is possible again.

Note: During execution of this function no interrupts can be served. As the execution time is relatively long (may exceed several hundreds of µs), measures against interrupt overrun have to be taken.

7.3 Basic Reprogramming

Basic reprogramming functions are related to the standard block reprogramming (See section 5.2 ”Basic Block Reprogramming Flow” on page 28). These functions support the operations Blank Check, Erase, Write, Internal Verify.

7.3.1 Area Blank Check

Library call:

u32 SelfLib_BlankCheck(u32 blockNo)

Required parameters:

blockNo	Block number to be checked (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Block is blank
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 ”Hardware Requirements” on page 39)
SELFLIB_ERR_PARAMETER	Block with the block number blockNo does not exist
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0xA). Device not blank

Function description:

This function checks, if the block is blank (erased) or not

Remark for polling mode (See section 5.9 ”User Application Execution During Self-Programming” on page 40):
In polling mode this function returns before the Blank Check is finished. The only return values then are SELFLIB_OK to indicate a successful background function initiation, SELFLIB_ERR_FLMDO or SELFLIB_ERR_PARAMETER.
The user control program need to do status checks (See section 7.5.5 ”Status check” on page 64) to poll for the end of the operation and to receive the background operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn).

7.3.2 Area Erase

Library call:

u32 SelfLib_Erase (	u32 blockNo)

Required parameters:

blockNo	Block number to be erased (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Erasing was successfully. Block is blank now
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 ”Hardware Requirements” on page 39)
SELFLIB_ERR_PARAMETER	Block with the block number blockNo does not exist
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0xA). Erase failed

Function description:

This function erases a complete Flash block.
The time required for erasing strongly depends on the used device, no. of already done erase cycles, temperatures and other conditions. It is not constant and might last up to several seconds in worst case.

Remark for polling mode (See section 5.9 ”User Application Execution During Self-Programming” on page 40):
In polling mode this function returns before the Blank Check is finished. The only return values then are SELFLIB_OK to indicate a successful background function initiation, SELFLIB_ERR_FLMDO or SELFLIB_ERR_PARAMETER.
The user control program need to do status checks (See section 7.5.5 ”Status check” on page 64) to poll for the end of the operation and to receive the background operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn).

7.3.3 Write

Library call:

u32 SelfLib_Write (	void *addSrc, void *addDest, u32 length)

Required parameters:

addSrc	Address of the source data that has to be written into the Flash
addDest	Destination where the data has to be written (The address alignment strongly depends on the used device. Please refer to the device information for the alignment)
Length	Number of WORDS to be written. According to the device and the Flash technology dedicated limitations for the Write granularity and maximum length are given. Please refer to the *device information* for the exact limitations. Values working generally are: - All V850 devices: length = 64 words (*32 bits)

Returned value:

SELFLIB_OK	Data is successfully written
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 ”Hardware Requirements” on page 39)
SELFLIB_ERR_PARAMETER	Address parameter error
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0xA). Write failed

Function description:

Data is written to the Flash. This is done word wise

Remark for polling mode (See section 5.9 ”User Application Execution During Self-Programming” on page 40):
In polling mode this function returns before the Write is finished. The only return values then are SELFLIB_OK to indicate a successful background function initiation, SELFLIB_ERR_FLMDO or SELFLIB_ERR_PARAMETER.
The user control program need to do status checks (See section 7.5.5 ”Status check” on page 64) to poll for the end of the operation and to receive the background operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn).

7.3.4 Internal Verify

Library call:

u32 SelfLib_IVerify (u32 blockNo)

Required parameters:

blockNo	Block number to be checked (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Internal Verify passed successfully
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 ”Hardware Requirements” on page 39)
SELFLIB_ERR_PARAMETER	Block with the block number blockNo does not exist
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0xA). Internal Verify failed

Function description:

This function compares the Flash contents on erase level against the write level. See 5.2 ”Basic Block Reprogramming Flow” on page 28

Remark for polling mode (See section 5.9 ”User Application Execution During Self-Programming” on page 40):
In polling mode this function returns before the Internal Verify is finished. The only return values then are SELFLIB_OK to indicate a successful background function initiation, SELFLIB_ERR_FLMDO or SELFLIB_ERR_PARAMETER.
The user control program need to do status checks (See section 7.5.5 ”Status check” on page 64) to poll for the end of the operation and to receive the background operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn).

7.4 Protection/Safety Related Operations

The function for setting the flags is explained in this chapter.

7.4.1 Get protection flags

Library call:

u32 SelfLib_GetInfo_SecFlags(void)

Required parameters:

-

Returned value:

Bit 0:	0: Chip Erase disabled 1: Chip Erase enabled
Bit 1:	0: Block Erase disabled 1: Block Erase enabled
Bit 2:	0: Write disabled 1: Write enabled
Bit 3:	0: Read command disabled 1: Read command enabled
Bit 4~31:	Reserved for future use

Unsupported Flags or reserved flags are set to 1

Function description:

This function returns a 32 bit value containing above mentioned security related bits. The meaning of the bits is explained in 4.2.2 ”Protection configuration settings” on page 20.

7.4.2 Get block swapping flag

Library call:

u32 SelfLib_GetInfo_BootSwap(void)

Required parameters:

-

Returned value:

0:	Boot swapping is not performed on device startup
1:	Boot swapping is performed on device startup

Function description:

This function returns the value of the boot swap bit (See section 4.3 ”Security” on page 21).

7.4.3 Set protection flags

Library call:

u32 SelfLib_SetSecFlags(u32 SecFlags,
u32 blockNo)

Required parameters:

SecFlags — 32-bit value containing the following bits:

Bit 0: 0: The affected two blocks are swapped to 0x0000000 on Reset
1: The affected two blocks are not swapped to 0x0000000 on Reset
Bit 1: 0: Chip Erase disabled
1: Chip Erase enabled
Bit 2: 0: Block Erase disabled
1: Block Erase enabled
Bit 3: 0: Write disabled
1: Write enabled
Bit 4: 0: Read command disabled
1: Read command enabled
Bits 5~31: Reserved for future use
Unsupported or reserved flags have to be set to 1

When using the result of the function SelfLib_GetInfo_GetSecFlags for this parameter, please consider the shifted bit position

blockNo — Block number where the flags are set (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Flag setting was successfully.
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 "Hardware Requirements" on page 39)
SELFLIB_ERR_PARAMETER	Block with the block number blockNo does not exist or the command is not supported by the device or wrong bits are set
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0xA). Setting security flags failed

Function description:

This function sets the security/safety related flags in the extra area (See section 4.2.2 "Protection configuration settings" on page 20).

Notes: 1. The security flags (Bits 1and higher) must be the same for 1st four blocks (See section 5.3 "Extra Information Handling" on page 29).

2. The boot flag must be set ('0') for the two blocks mapped to 0x00000000 and following addresses.

3. The protection flags and variable reset vector must be set during initial programming via the programming interface (e.g. using PG-FP4). Changing the flags using this function (e.g. setting an additional bit) is not possible. This function has to set the as they have been set by the initial programming.

Remark for polling mode (See section 5.9 "User Application Execution During Self-Programming" on page 40):
In polling mode this function returns before the command background execution is finished. The only return values then are SELFLIB_OK to indicate a successful background function initiation, SELFLIB_ERR_FLMDO or SELFLIB_ERR_PARAMETER.
The user control program need to do status checks to poll for the end of the operation and to receive the background operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn).

7.4.4 Set protection flags and variable reset vector

Library call:

u32 SelfLib_SetSecFlagsVRV(u32 SecFlags,
u32 resetVctAdr,
u32 blockNo)

Required parameters:

SecFlags — 32-bit value containing the following bits:

- Bit 0: 0: The affected two blocks are swapped to 0x0000000 on Reset
  1: The affected two blocks are not swapped to 0x0000000 on Reset
- Bit 1: 0: Chip Erase disabled
  1: Chip Erase enabled
- Bit 2: 0: Block Erase disabled
  1: Block Erase enabled
- Bit 3: 0: Write disabled
  1: Write enabled
- Bit 4: 0: Read command disabled
  1: Read command enabled
- Bits 5~31: Reserved for future use

Unsupported or reserved flags have to be set to 1

When using the result of the function SelfLib_GetInfo_GetSecFlags for this parameter, please consider the shifted bit position

resetVctAdr — Reset vector address: Address of the first executed instruction after Reset

blockNo — Block number where the flags are set (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Flag setting was successfully.
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 ”Hardware Requirements” on page 39)
SELFLIB_ERR_PARAMETER	Block with the block number blockNo does not exist or the command is not supported by the device or wrong bits are set
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0xA). Setting security flags failed

Function description:

This function sets the security/safety related flags in the extra area (See section 4.2.2 ”Protection configuration settings” on page 20).

Notes: 1. The security flags must be the same for 1st four blocks (See section 5.3 ”Extra Information Handling” on page 29).

2. The boot flag must be set (’0’) for the two blocks mapped to 0x00000000 and following addresses after RESET

3. The protection flags and variable reset vector must be set during initial programming via the programming interface (e.g. using PG-FP4). Changing the flags using this function (e.g. setting an additional bit) is not possible. This function has to set the as they have been set by the initial programming.

4. If the variable reset vector has not been initialized via the programming interface (e.g. using PG-FP4), this function cannot be used. Use the function SelfLib_SetSecFlags instead

Remark for polling mode (See section 5.9 ”User Application Execution During Self-Programming” on page 40):
In polling mode this function returns before the command background execution is finished. The only return values then are SELFLIB_OK to indicate a successful background function initiation, SELFLIB_ERR_FLMDO or SELFLIB_ERR_PARAMETER.
The user control program need to do status checks to poll for the end of the operation and to receive the background operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn).

7.5 Miscellaneous Functions

7.5.1 Get device number

Library call:

u32 SelfLib_GetInfo_Device(void)

Required parameters:

-

Returned value:

CPU number in decimal format
e.g. uPD70F3166 (V850ES/SA3) = 3166 (=0x0C5E)

Function description:

This function returns the CPU number, that is part of the device name, in decimal format

7.5.2 Get block count

Library call:

u32 SelfLib_GetInfo_BlockCnt(void)

Required parameters:

-

Returned value:

Number of Flash blocks in the device

Function description:

This function returns the number of Flash blocks in the device

7.5.3 Get block end address

Library call:

u32 SelfLib_GetInfo_BlockEndAdd(u32 blockNo)

Required parameters:

blockNo — Block number to be checked (Not block address, but number of the flash block)

Returned value:

End address of the selected block

Function description:

This function returns the end address of the block passed as a parameter.

7.5.4 Boot swap

Library call:

u32 SelfLib_BootSwap(void)

Required parameters:

-

Returned value:

0 ~ 4	Number of Flash block, that is located at 0x000000000 after swap
SELFLIB_ERR_PARAMETER	The command is not supported

Function description:

This function swaps the boot blocks and returns the new block number located at 0x00000000. Please refer to 5.4 ”Secure Reprogramming Flow with Bootloader Update” on page 30.

7.5.5 Status check

This function is only required in polling mode (See section 5.9 ”User Application Execution During Self-Programming” on page 40).

Library call:

u32 SelfLib_StatusCheck(void)

Required parameters:

-

Returned value:

SELFLIB_OK	The background Flash operation terminated successfully
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0xA). The background Flash operation failed.
SELFLIB_BUSY	The background Flash operation is still ongoing

Function description:

The function checks the status of started background Flash operations, like Erase, Write, Internal Verify,...

Please refer to the functions description to see, whether status check is necessary for that function in polling mode or not.

7.5.6 Check mode (FLMD0)

Library call:

u32 SelfLib_ModeCheck(void)

Required parameters:

-

Returned value:

SELFLIB_OK	V_{DD} is applied at the pin FLMD0
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 5.8 ”Hardware Requirements” on page 39)

Function description:

This function explicitly checks the FLMD0 pin and returns the current status

[MEMO]

Chapter 8 SST (UX4/CZ6HSF) - Self-Programming

8.1 SelfLib Functionality

Except the control program which is application specific and therefore has to be made by the user, the SelfLib itself contains all functions important for self-programming:

- *Flash environment* Activation / Deactivation (See below)
- User interface
  It contains the functions to be called by the user Flash reprogramming control program
- Firmware interface

The *SelfLib* calls the device internal firmware via the firmware interface, which then controls the Flash environment for re-programming. The firmware is located in a device internal mask ROM. So, it is fix in order to ensure, that as few as possible user changeable functions endanger the Flash and its contents.

8.1.1 Flash Environment

The *Flash environment* is additional hardware (charge pumps, sequencer,...), required to do the reprogramming. It is not continuously activated, but only for reprogramming.
Environment activation and deactivation is done by the library.

Note: The **complete** Flash is not accessible at all while the *Flash environment* is active. So parts of the SelfLib need to be executed outside the Flash (e.g. in embedded RAM). Copying code to RAM is supported by the *SelfLib* and explained later in this document

8.2 Device Reprogramming Overview

First of all it is important to select the correct reprogramming flow. Please check with the application and with the *DeviceInfo*, whether secure reprogramming and secure bootloader update is required and available or not (See section 4.3.2 ”Secure reprogramming using self-programming” on page 22).
If secure bootloader update is required, please refer to 8.5 ”Secure Reprogramming Flow with bootloader Update” on page 71 for the correct flow.

In the following, the basic steps are explained. The next sub chapters then explain the standard block reprogramming and the secure bootloader reprogramming. The standard block reprogramming flow is also embedded in the secure bootloader reprogramming.

The following principle steps have to be executed for any device reprogramming:

Table 8-1: Basic Steps

Step	Function	Description
1	Setup the user program	- Setup communication Interface, watchdog timer, FLMD0 voltage activation,...
2	Self-Programming initialisation	See section 10.1 ”Initialisation” on page 85.
3	Reprogram the Flash	Take care to keep the correct sequence for reprogramming single Flash blocks (see section 8.3 ”Basic Block Reprogramming Flow” on page 69) and for secure bootloader reprogramming (see section 8.5 ”Secure Reprogramming Flow with bootloader Update” on page 71).
4	End Reprogramming	Return to bootloader, start application, reset or...

8.3 Basic Block Reprogramming Flow

This is the standard flow, that shall be used to reprogram all Flash blocks.

To ensure, that the data is written correctly and the data retention is given it is important to keep the following sequence for block reprogramming. Whenever a function returns an error, you may not continue the sequence because the next steps may not work correctly and the result is undefined.

Table 8-2: Block Reprogramming Sequence

Step	Flash Lib. Function	Description
1	Block Erase	The Flash block is erased
2	Write	Data is written to the Flash
3	Internal Verify	This step is executed after the last Write to a dedicated block to ensure the specified data retention of the Flash The Internal Verify is a check for higher reliability of the programming, but no manipulation or operation on a 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, Vdd/Gnd bounce)

8.4 Extra Information Handling

Setting protection flags

It is possible to set the protection flags either during the initial programming with a dedicated Flash programmer (e.g. PG-FP4) or during self-programming using the SelfLib. Following that high flexibility during development, but also during mass production is given.

Boot swap flag

If the flag is set to 0, a boot swap is executed. In that case the upper boot block cluster (See section 4.2.2 "Protection configuration settings" on page 20) is swapped to 0x00000000, while the other cluster is swapped to the upper location.

Extra information handling during self-programming

The extra area containing the different flags is treated completely independent. Following that the flags need only be reprogrammed, when the boot swap flag is updated or when the protection settings are to be changed. Differing from UC2, the extra area is not affected by any Block Erase operation.
Following that, when normal reprogramming is done without secure bootloader update (See section 8.3 "Basic Block Reprogramming Flow" on page 69), only the normal block reprogramming flow need to be considered, even with respect to the protection flags.
See 10.3 "Protection/Safety Related Operations" on page 92 for an example on how to set the flags

Notes: 1. In order only to increase the protection level, it is possible to set additional restrictions for the Flash programming, not remove restrictions

- Any protection flag can only be set, not be cleared
- The boot cluster size can not be changed, if the block protection flag is set
- The boot swap flag can not be changed, if the block protection flag is set

2. A flag is set by writing it to 0 while a cleared flag has the value 1.

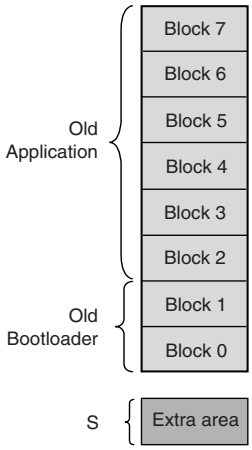
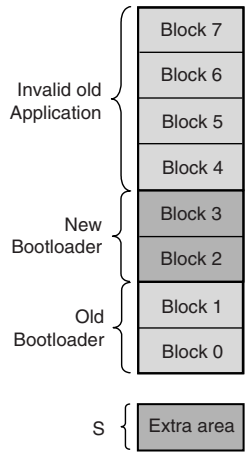
8.5 Secure Reprogramming Flow with bootloader Update

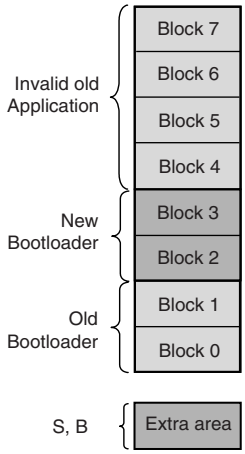
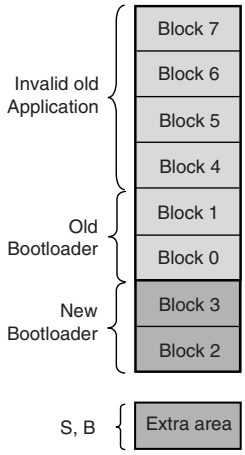
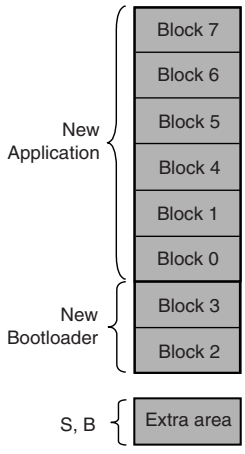
The basic feature allowing this way of secure reprogramming is, that on device start-up is checked which block contains a valid bootloader. To do so the boot swap flag in the extra area(s) is analysed.
By the boot swap feature and the correct reprogramming sequence it can be assured, that always a valid boot program is started to continue the reprogramming sequence in case that the reprogramming algorithm was interrupted (Power fail, accidental reset).
The bootloader must be able to control the complete reprogramming sequence including data transfer of the new Flash contents.

The sample configuration considers a device with 8 blocks and with boot swap capability, where the blocks 0/1 can be swapped with 2/3.

Note: The block number in the diagrams below are the physical block numbers. They are always fix, even after block swapping. E.g.: block 2 before boot swapping remains block 2 even after the swap. This is done for better illustration.
When using the SelfLib functions, logical block numbers are used. On logical block numbers the block on address 0x00000000 is always block 0, followed by block 1 and so on. This is valid even after block swapping. E.g.: Block 2 before swapping is block 0 after swapping. By that software development is easier, as the block numbers need not be modified depending on the swapping.

Table 8-3: SST(UX4) Flash Secure Reprogramming Sequence Incl. bootloader Update

Step	SelfLib Function	Description	Sample Configuration
1	Initial Status	• Boot Program in the lower 2 areas • Safety flags in the extra area set (e.g. Write disable, Block Erase disable) • Boot flag in the extra area not set. So Blocks 0/1 are located at 0x0000000 • Blocks 2~7 contain the application	
2	Reprogram blocks 2/3 with the new bootloader	The new bootloader is written to the blocks, that shall be swapped to 0x0000000 later on Execute the complete block reprogramming flow for both blocks (See 8.3 ”Basic Block Reprogramming Flow” on page 69): • Erase • Program • Internal Verify (See also 8.3 ”Basic Block Reprogramming Flow” on page 69) When erasing the first application program block the application is no longer valid and able to run. In case of power fail, the boot-loader handles the complete reprogramming flow.	

Step	SelfLib Function	Description	Sample Configuration
3	Rewrite security flags	General rule regarding the boot flag is, that it has to be inverted in order to swap the blocks. In this example the boot flag needs to be set (Example in 4.3.1 "Set protection flags and boot cluster size" on page 32) After writing the security flags and boot flag, the new bootloader in blocks 2 and 3 would be active in case of a device restart (e.g. due to power failure)	
4	Swap blocks 0/1 and 2/3	After swapping the blocks, the begin of the new bootloader (block 2) is located at 0x00000000	
5	Reprogram the blocks 0, 1, 4~7	Execute the complete block reprogramming flow for each block separately (See 8.3 "Basic Block Reprogramming Flow" on page 69): • Erase • Program • Internal Verify (See also 8.3 "Basic Block Reprogramming Flow" on page 69) Note: Pass the logical block numbers (not physical block numbers mentioned in the diagram on the right side) as parameters to the SelfLib functions requiring block numbers!	

8.6 Reprogramming Scenario

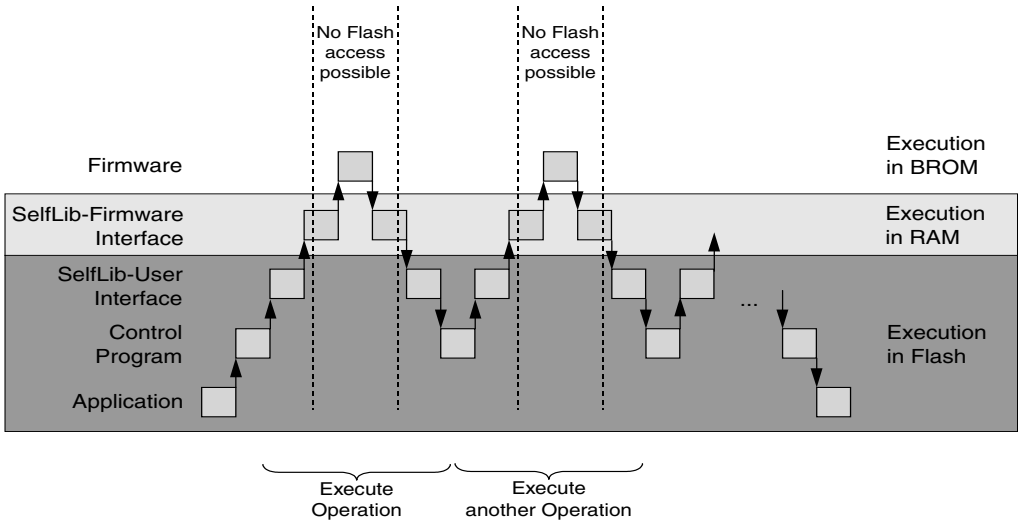
The reprogramming scenario describes the way in which reprogramming background operations are initiated and how the execution status is checked.

The scenario is explained in detail in the following.

Note: As different scenarios are under planning, the library is already prepared for different options. The necessary set-up in the library is done by the global define “SELFLIB_EXESCEN” in the file “SelfLibSpecific.h”. For the time being, the define may not be changed.

The user application, containing the re-programming control program initiates a programming operation (e.g. Erase) by calling the appropriate SelfLib function. The SelfLib passes control to the device internal firmware, which completely executes the operation and returns to the SelfLib at the end. The SelfLib returns then control immediately the control program.
The *Flash environment* is being activated implicitly by the SelfLib functions. Therefore, (de-)activation by the user program is not necessary and may lead to unpredictable program behaviour.

Figure 8-1: Firmware Execution Scenario



8.7 SelfLib Sections

In order to support the different environments and reprogramming sequences, the library is splitted up into several sections.

The following library sections exist:

- SelfLib_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 initialisation is part of this section.

- SelfLib_RomOrRam
  This section contains the user interface. Depending on the reprogramming scenario (See section 8.6 ”Reprogramming Scenario” on page 74) this section is copied into the RAM or not. The copy procedure is automatically executed by the library initialisation (See section 10.1 ”Initialisation” on page 85).

Note: This section has been introduced to gain flexibility for further device and *SelfLib* feature extensions. Currently the section is not copied to RAM.

- SelfLib_ToRam
  This section need to be executed outside the reprogrammed Flash. As it is usually copied to the internal RAM, it is called SelfLib_ToRam. The sections contains the device Firmware interface.

- SelfLib_ToRamUsrInt
  This section contains the Interrupt function entry during activated *Flash environment*.
  The device internal firmware passes all acknowledged interrupts to the 1st RAM addresses, where the interrupt function entry routine need to be located.
  In order to achieve this, the *SelfLib* initialisation copies this section to the SelfLib destination address first. **The SelfLib destination address must be the first RAM address!**

  Even if not used and empty, the section must exist, as address references to this section are used in the SelfLib initialisation.

- SelfLib_ToRamUsr
  This section contains user functions to be executed in RAM, where it is copied by the *SelfLib* initialisation
  Part of these user functions are the Interrupt functions, called by the interrupt entry in the SelfLib_ToRamUsrInt section. Furthermore, this section may also contain other user functions, like the self-programming control program, etc.
  Even if not used and empty, the section must exist, as address references to this section are used in the SelfLib initialisation.

Note: Function calls on V850 devices are realized as relative jumps. Following that, the linked relative address distance between SelfLib_ToRamUsrInt (Interrupt entry) and SelfLib_ToRamUsr (Interrupt function) may not change before and after copy to RAM. As the section SelfLib_ToRamUsr is copied immediately (4 Byte alignment) behind SelfLib_ToRamUsrInt by SelfLib_Init, the user must also consider this section sequence in the linker directive file.

- SelfLib_RAM
  This section contains the variables required for SelfLib. It can be located to internal or external RAM.

8.7.1 Automatic section copy

The application, including the control program and the SelfLib 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. The control program, as well as the user interface of the SelfLib may be located in Flash. Only the firmware interface of the SelfLib and user interrupt functions must be located outside the Flash on execution.

If the copy functionality of the SelfLib_Init function is configured, the following sections, containing these functions, are copied automatically to the destination address in RAM (See 4.1.1 ”Library initialization” on page 25):

- SelfLib_ToRamUsrInt
- SelfLib_ToRamUsr
- SelfLib_ToRam

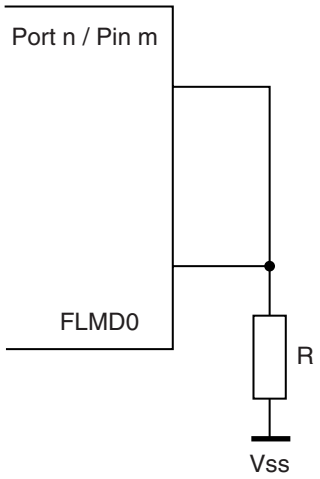
Notes: 1. The RAM destination address, passed to SelfLib_Init function must be the 1st RAM address (interrupt entry on activated Flash environment) as the section SelfLib_ToRamUsrInt need to copied there!

2. The copy functionality of the SelfLib_Init function has a drawback. As the code location differs from the linked position, source level debugging is not possible inside the copied code. For the *SelfLib* code this is not a big problem, as it is well tested, but for user code this need to be considered.
Alternative (for GHS) solution could be:
Similar to the.data section. Link the code to the destination address and use the possibility of the GHS startup routines to copy the code to the RAM from a ROM image. Please refer to the GHS documentation, "ROM linker section attribute".

8.8 Hardware Requirements

The requirements of self-programming regarding the device external hardware is very low as no dedicated external programming voltage is required.
However, it has to be considered, that a security concept against unintended reprogramming has been set-up. It is required, that the devices FLMD0 pin is externally attached to V_{DD} (I/O voltage) during self-programming. FLMD0 should be attached to V_{DD} immediately before or after SelfLib initialisation (See the device specific application sample software) and reset to V_{SS} at the end of self-programming. Default state on reset is V_{SS}.

Figure 8-2: 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}.

8.9 User Application Execution During Self-Programming

The reprogramming flow is set-up in the way, that a user written self-programming control program controls the basic reprogramming flow and the SelfLib calls. The SelfLib functions interact with the Flash. According to the called SelfLib function the execution requires times from a few nanoseconds up to several seconds. In many applications it is acceptable, that no further software execution during that time is possible. This is the easiest way of a reprogramming sequence.
However, for most applications it is not acceptable, that no more code can be executed during execution of long lasting SelfLib functions (like the Erase function).

The NEC single voltage Flash devices are designed in a way, that the time consuming Flash manipulation operations only have to be initiated and then are executed in the background. Completion of the operations is then checked by the firmware in order to continue the self-programming or return to the user application

This separates the time critical FLASH operations (executed in hardware in background) and non time critical status check operations (executed by firmware). Due to the separation it is possible to execute interrupt driven user functions during the complete self-programming.

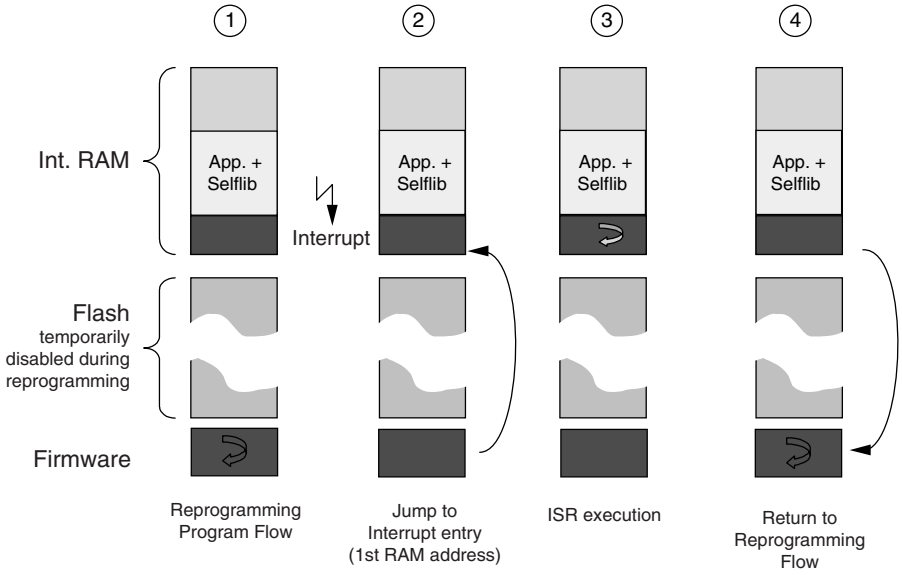
8.9.1 Interrupt functions

This sub-chapter explains the Interrupt function execution during self-programming.

On deactivated *Flash environment* the interrupt vector table is available and the normal user defined interrupt functions can be executed. When the environment is activated, this is not the case. As the Flash including the interrupt vector table is not accessible at that time, normal jumps into the vector table are not possible. Instead, the interrupt jump is handled by the device internal firmware in the following way:

- All interrupt vectors are relocated to an entry point in the internal RAM. No register handling or mode change is done by the firmware. The user handler must be designed as interrupt function including register save/restore and reti at the end
- New entry point of **all non maskable interrupts** is the 1st address of the internal RAM (depending on the device, e.g. 0xFFFFE000). A user handler routine must check the interrupt source. The source can be read from the exception cause register ECR (See device users manual).
- New entry point of **all maskable interrupts** is the word address following the masked interrupt entry (depending on the device, e.g. 0xFFFFE004). A user handler routine must check the interrupt source. The source can also be read from the exception cause register (See device users manual).

Figure 8-3: Interrupt Routine Execution in Internal RAM / External Memory



The SelfLib is designed in order to ease the interrupt user handler implementation. This is done by special treatment of the two SelfLib sections SelfLib_ToRamUsrInt and SelfLib_ToRamUsr. These sections are copied to the destination address by the SelfLib_Init function, if the copy functionality is configured.

- The section SelfLib_ToRamUsrInt is the first copied section. In the application sample it contains the user interrupt entry as the only functionality.

- The section SelfLib_UsrInt is copied by SelfLib_Init immediately behind the SelfLib_UsrInt1st (4Byte alignment). It is designed to contain the interrupt user function.

Notes: 1. The RAM destination address, passed to SelfLib_Init function must be the 1st RAM address (interrupt entry on activated Flash environment) as the section SelfLib_ToRamUsrInt need to copied there!

2. As Normal function calls from section SelfLib_UsrInt1st to SelfLib_UsrInt are intended, the sections must also be linked immediately after another with a 4Byte alignment. Please set up the linker control file accordingly!

3. The execution latency of interrupt routines on activated *Flash environment* is increased by the fact, that the interrupts are passed to the first RAM addresses by the device internal firmware. However, the increase is quite low and should not exceed 100us on maximum device frequency.

Chapter 9 SST (UX4/CZ6HSF) - SelfLib Configuration

SelfLib implementation relevant issues are discussed in the following sub-chapters.

9.1 Used Resources

The only resources used by the self-programming, is memory.

Beside the SelfLib sections the stack is used. The stack usage is depending on the library and on the used compiler environment. Furthermore, also the device firmware uses the stack.

The complete FLASH is not available at time during self-programming.

9.2 Software Considerations

This chapter lists up special issues to be considered when implementing self-programming into the application.

DMA operation during self-programming
DMA operations during self-programming (While the Flash environment is activated) are not allowed at all. Background is the fact, that accesses sequences to protected SFR registers may fail in case of DMA accesses to the peripheral bus during this sequence

Run-Time Library calls
During Self-Programming (activated Flash Environment) It is not possible to call run-time library functions, if the library is not copied to RAM together with the Self-Programming code.
In case of program failures during self-programming, the code should be explicitly checked for such calls.

Switch instruction in C
In case of the user status check, the switch instruction should be avoided in all the code executed in RAM. Depending on the used compiler this instruction may result in a call to a run-time library, which has not been copied.

9.3 Compiler Configuration

As the SelfLib is delivered in source code, the user has to take care for the correct compiler settings in his applications. Due to the huge amount of setting options, not all could be tested. The settings coming along with the application samples should be taken as reference for the user application development.

9.3.1 Project settings

The most important settings for a successful application build shall be explained in the following:

Position independent code (PIC)
The SelfLib code shall be compiled position independent.

Inline function prologue and epilogue
In order to avoid any library calls for function prologue and epilogue into a section, not available during self-programming (Flash), they need to be unlined.

V850E core instruction set
The extended instruction set of the V850E core is used for the assembler parts. So this core should be selected.

The settings are only required for the complete SelfLib and all code added to the SelfLib sections. The rest of the project may have different settings. However, it is recommended to use the same settings project wide, if possible.

System calls
The linkers can add small code fragments and appropriate labels for so called system calls. These calls are required to do e.g. terminal-io or file-io. Whenever such an operation is required, the program jumps to the special code. The debugger detects this by a debug break and executed special operations. As partially device firmware is executed on the same addresses as the normal program code, the firmware might run into the debug break too, resulting in unexpected program behaviour. This must be avoided by disabling certain debug functionality.

Green Hills (GHS) Multi 2000

To set the settings project wide, click on the project file in the project window to select it before entering the menu.

- Select position independent code
- Select inline prologue
- Select the processor type "V850E1" respectively. "V850E/MS1" for V850E core instruction set

Further required settings:

- Set the following driver options for the assembler files to ensure, that c-style preprocessing directives are processed:
  Click on an assembler file to select it. Then select the menu “Project -> File options”. In the driver options field type in “-preprocess_assembly_files”. Do this for each assembler file.
- Disable usage of the callt instruction, as it results in calls to the run-time library, which is not copied into the RAM in case of time saving scenario.

System calls can be avoided inside the debugger.

- Type in the debugger cmd window: target syscalls off
  This can also be added to the .rc or the .mbs debugger control files.

IAR Embedded Workbench

Using the IDE, the settings can be found in the menu “Project -> Options”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- Category ICCV850: Optimize for speed. By that, the function prologue/epilogue is inlined
- Category general: Select CPU variant V850ES or V850E for the V850E core instruction set
- Category Linker->Output->”Debug Information for C-Spy”: Deselect “with runtime control modules” to avoid system calls

No explicit setting required for position independent code.

NEC CA850

Using the IDE, the following can be configured in the menu “Tool -> Compiler Options” under the tab “Others”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- In the box "Any Option" type in “-Ot” for inline prologue/Epilogue

Using the IDE, the following can be configured in the menu “Tool -> Linker Options” under the tab “Library”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- De-select "Link standard library" and de-select "link mathematics library"

No explicit setting required for position independent code.
The core selection is done during project initialization by selecting the correct target device.

No explicit settings for system calls required, as these are not supported by the NEC compiler

9.4 Global SelfLib Defines - SelfLibSpecific.h

The *SelfLib* is optimized for code size. This is realized by special device and environment dependent defines. Setting these defines controls the compilation of the library. Differing from setting SelfLib options by function parameters (like in the UC2 *SelfLib*), unused code will so be avoided and the code size will be reduced.

The defines are located in the file **SelfLibSpecific.h**. This file is included in the different *SelfLib* c-modules and should there for be located in the same directory.

The defines as of today are explained in the following. Future extensions according to device changes or customer requirements may be possible:

SELFLIB_SPECIFIC_OPTIONS
Different firmware implementations are available, especially but not only with respect to the different basic technologies (UX4, CZ6HSF,...). These implementation require slightly different data structures and set-up inside the SelfLib. Please set the define according to the used device. The following settings are defined as of today. Please refer to the *DeviceInfo* for the correct setting for your device:

- SELFLIB_KX1_V1 Setting for 1st versions of the K_Line devices
- SELFLIB_PHOENIX_F Special setting for Phoenix-F
- SELFLIB_PROENIX_FS Special setting for Phoenix-FS
- SELFLIB_IX3 Special setting for V850E/Ix3 Devices

SELFLIB_STATUS_CHECK
This define is available for future extensions of the *SelfLib.* It may be used (modified) in case of future enhanced device features with respect to execution of user applications during self-programming. As of today only the first option (status check by the firmware) is supported.

- STATUS_CHECK_FW Hardware background operation is initiated by the device firmware. Status check is also done by the device firmware. User application execution possible only using interrupt functions (Similar to the “interrupt mode” in UC2)
- STATUS_CHECK_SELFLIB Hardware background operation is initiated by the device firmware. Status check is done by the SelfLib. User application execution possible only using interrupt functions (In UC2 this was called “interrupt mode”)
- STATUS_CHECK_USER Hardware background operation is initiated by the device firmware. Status check is done by the User program. User application execution is possible in the status check loop (In UC2 this was called “polling mode”).

Chapter 10 SST (UX4/CZ6HSF) - SelfLib API

The SelfLib contains functions of several different categories. These are called by the user program in the sequences as described in the last chapters. The library function calls and parameters are explained in the following.

10.1 Initialisation

Functions explained in this chapter are called at the beginning of the self-programming. They are necessary for the SelfLib initialisation, but some functions can also be used for set-up of a reprogramming control program.

10.1.1 Library initialization

Library call:

u32 SelfLib_Init(	void *destAddSelfLib, u32 frequency, SECTION_COPY copy)

Required parameters:

destAddSelfLib	The *Relocated Sections* are copied to this location. Depending on the copy parameter (see below) these are SelfLib_ToRamUsrInt, SelfLib_ToRamUsr, SelfLib_ToRam

Caution: If interrupt functions shall be executed during self-programing (See section 8.9 ”User Application Execution During Self-Programming” on page 78), the address must be set to the beginning of the internal RAM (e.g. 0xFFFFE000), as the interrupt handler entry is there.

frequency	The operation frequency of the device (Hz) e.g.: crystal 5 MHz, internal PLL with the factor 10: frequency=50,000,000 This parameter is used for CZ6HSF devices only. For other devices it must be set to 0
copy	SELFLIB_COPY: The *Relocated Sections* are copied to the address, passed to SelfLib_Init by the parameter destAddSelfLib (See above). SELFLIB_DONT_COPY: The sections are not copied

Returned value:

SELFLIB_OK	The initialisation was successful
SELFLIB_ERR_PARAMETER	Wrong frequency parameter (e.g. frequency!= 0, when the define for the Flash technology was set to default or to UX4. See 3.4 ”Global SelfLib Defines - SelfLibSpecific.h” on page 24)

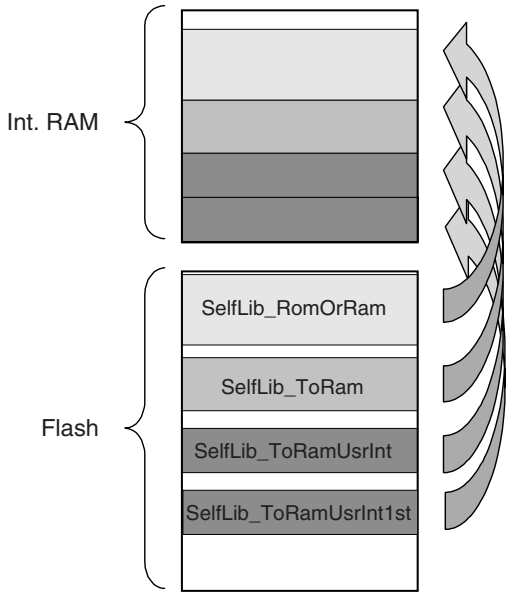
Function description:

The function copies the sections into the RAM if requested (see above).

The SelfLib_ToRamUsrInt section is copied first (to destAddSelfLib), in order to enable the user to place it on a defined address for easy interrupt handling support (See section 8.9.1 ”Interrupt functions” on page 79).
Directly behind that SelfLib_ToRamUsr is copied, followed by SelfLib_ToRam and SelfLib_RomOrRam. The copy routine doesn’t leave any holes between the copied sections.

In addition to the section copy the SelfLib data and internal pointers are initialised.

Figure 10-1: Section Copy in SelfLib Initialisation



Function call sample:

The following function call samples shall explain the usage of the SelfLib_Init function. Depending on the application one of the samples may be used. The parameters have to be adapted to the user application.

- The program to control the reprogramming (e.g. bootloader or monitor program) including the library is located in the device internal flash. To do Flash programming the sections SelfLib_ToRamUsrInt, SelfLib_ToRamUsr, SelfLib_ToRam and SelfLib_RomOrRam need to be copied to internal RAM or external memory.

```
ret = SelfLib_Init (SELFLIB_DEST, /* The copy destination address */
                    FREQUENCY,    /* Operation frequency */
                    COPY);        /* Copy the sections to SELFLIB_DEST */
```

- The program to control the reprogramming (e.g. bootloader or monitor program) including the library is already located in the device internal RAM or external memory. Functions in the SelfLib_ToRamUsrInt, SelfLib_ToRamUsr, SelfLib_ToRam and SelfLib_RomOrRam sections can be executed directly on their original location and need not be copied. No interrupt routines copied to RAM during reprogramming are used

```
ret = SelfLib_Init (SELFLIB_DEST, /*The copy destination address */
                    FREQUENCY,    /* Operation frequency */
                    DONT_COPY);   /* Don't copy the sections to SELFLIB_DEST */
```

Sample definitions (Depending on the application and the device):

```
#define SELFLIB_DEST   0xFFFFE000          /* e.g. Destination address 0xFFFFE000 */

For CZ6HSF devices:
#define FREQUENCY      16000000            /* e.g. System operation frequency 16MHz */
For UX4 devices:
#define FREQUENCY      0                   /* No frequency setting necessary --> set to 0 */
```

10.1.2 New function address

Library call:

u32 *SelfLib_FktNewAddress (void *addFct,
void *destAddSelfLib)

Required parameters:

addFkt	Original Function address.
destAddSelfLib	Destination address in RAM. See section 10.1.1 ”Library initialization” on page 85

Returned value:

Address of the function inside the SelfLib_RomOrRam or the SelfLib_ToRamUsr Section on its destination address.
If the function is not in any *SelfLib* section, 0x00000000 is returned as error value

Function description:

If a user function is located in the section SelfLib_RomOrRam or SelfLib_ToRamUsr it may be copied to a save RAM location by SelfLib_Init. If the function has then to be called from outside of the section where it is located, the normally used relative function calls don’t work any longer.Following that, it has to be called by a function pointer. To set-up the function pointer SelfLib_FktNewAddress returns the new address of the function.

10.2 Basic Reprogramming

Basic reprogramming functions are related to the standard block reprogramming (See section 8.3 "Basic Block Reprogramming Flow" on page 69). These functions support the operations Blank Check, Erase, Write, Internal Verify.

10.2.1 Area Blank Check

Library call:

u32 SelfLib_BlankCheck(	u32 blockNoStart, u32 blockNoEnd)

Required parameters:

blockNoStart	First Block number to be checked (Not block address, but number of the flash block)
blockNoEnd	Last Block number to be checked (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Blocks are blank
SELFLIB_ERR_PARAMETER	blockNoStart or blockNoEnd is invalid
SELFLIB_ERR_FLASHPROCn	(n = 0x00~0x2). At least one block is not blank

Function description:

This function checks, if a range of blocks is blank (erased) or not.

Note: The Blank Check confirms that all cells within the checked flash area still have the correct margin level (erase level). A failure of the Blank Check does not imply a flash problem as such, it only indicates the necessity to perform an Erase operation to ensure proper start conditions before the programming of the flash memory.

10.2.2 Area Erase

Library call:

u32 SelfLib_Erase (	u32 blockNoStart, u32 blockNoEnd)

Required parameters:

blockNoStart	First Block number to be erased (Not block address, but number of the flash block)
blockNoEnd	Last Block number to be erased (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Erasing was successfully. Blocks are blank now
SELFLIB_ERR_PARAMETER	blockNoStart or blockNoEnd is invalid
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0x2). Erase failed
SELFLIB_ERR_PROTECTION	Flash protection error. The error is generated, if the boot cluster protection bit is set and the Erase shall be applied to a boot block. See 10.3 ”Protection/Safety Related Operations” on page 92

Function description:

This function erases a range of Flash blocks.

The time required for erasing strongly depends on the used device, no. of already done erase cycles, temperatures and other conditions. It is not constant and might last up to several seconds in worst case.

Note: Device internally, the Block Erase function can erase certain sets of blocks in parallel, reducing the erase time significantly. If the blocks are within one Flash macro and the range is 2^n and the alignment is according to the size, these blocks are erased in parallel.
If the range, passed to the SelfLib_Erase function, does not fit the criteria for parallel erase, the range is split up automatically by the SelfLib and device firmware into fitting sub-ranges, which are then erased sequentially. As this is done automatically, the user control program does not have to take care for that.

Example:
blockNoStart = 3
blockNoEnd = 87
Erase is internally split up into: Erase 3 --> Erase 4~7 --> Erase 8~15 --> Erase 16~31 --> Erase 32~63 --> Erase 64~79 --> Erase 80~87.

10.2.3 Write

Library call:

u32 SelfLib_Write (	void *addSrc, void *addDest, u32 length)

Required parameters:

addSrc	Address of the source data that has to be written into the Flash
addDest	Destination where the data has to be written (The address alignment strongly depends on the used device. Please refer to the device information for the alignment)
Length	Number of WORDS to be written. According to the device and the Flash technology dedicated limitations for the Write granularity and maximum length are given. Please refer to the *device information* for the exact limitations. Values working generally are: - All V850 devices: length = 64 words (*32 bits)

Returned value:

SELFLIB_OK	Data is successfully written
SELFLIB_ERR_PARAMETER	Address parameter error
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0x2). Write failed
SELFLIB_ERR_PROTECTION	Flash protection error. The error is generated, if the boot cluster protection bit is set and the Erase shall be applied to a boot block. See 10.3 ”Protection/Safety Related Operations” on page 92

Function description:

Data is written to the Flash. This is done word wise.

10.2.4 Internal Verify

Library call:

u32 SelfLib_IVerify (	u32 blockNoStart, u32 blockNoEnd)

Required parameters:

blockNoStart	First Block number to be checked (Not block address, but number of the flash block)
blockNoEnd	Last Block number to be checked (Not block address, but number of the flash block)

Returned value:

SELFLIB_OK	Internal Verify passed successfully
SELFLIB_ERR_PARAMETER	blockNoStart or blockNoEnd is invalid
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0x2). Internal Verify failed

Function description:

This function compares the Flash contents on erase level against the write level. See also 8.3 ”Basic Block Reprogramming Flow” on page 69

10.3 Protection/Safety Related Operations

10.3.1 Set protection flags and boot cluster size

Library call:

u32 SelfLib_SetSecFlags(u32 SecFlags)

Required parameters:

SecFlags — 32-bit value containing the following bits:

Bit 0:	0: The boot clusters are swapped on Reset 1: The boot clusters are not swapped on Reset
Bit 1:	0: Chip Erase disabled 1: Chip Erase enabled
Bit 2:	0: Block Erase disabled 1: Block Erase enabled
Bit 3:	0: Write disabled 1: Write enabled
Bit 4:	0: Read command disabled 1: Read command enabled
Bit 5:	0: Boot Cluster is protected 1: Boot Cluster is not protected
Bits 6~23:	Reserved for future use
Bits 24~31:	Last boot block number (=Number of blocks in the boot cluster - 1)

Unsupported Flags or reserved flags have to be set to 1

Returned value:

SELFLIB_OK	Flag setting was successfully.
SELFLIB_ERR_PARAMETER	Block with the block number blockNo does not exist or the command is not supported by the device or wrong bits are set
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0x2). Setting security flags failed
SELFLIB_ERR_PROTECTION	Flash protection error. The error is generated, if the boot cluster protection bit is set and the Erase shall be applied to a boot block. See 10.3 ”Protection/Safety Related Operations” on page 92

Function description:

This function sets the security/safety related flags in the extra area (See 4.2 ”Flash Protection” on page 18 and 8.4 ”Extra Information Handling” on page 70).

Notes: 1. Boot Swap handling:
The self-programming library offers 2 functions to swap the boot blocks:

Temporary boot swap:
The function SelfLib_BootSwap() (See 4.3.4 ”Secure self-programming with bootloader update” on page 22) swaps the boot blocks immediately. As this function does not modify the boot swap flag in the device extra area, the swap effect lasts only until the next RESET. This function may be part of a secure reprogramming sequence with bootloader update (See 8.5 ”Secure Reprogramming Flow with bootloader Update” on page 71), but additionally the function SelfLib_SetInfo() is required for a permanent boot swap!

Permanent boot swap:
The function SelfLib_SetInfo() writes the boot swap flag in the Flash Extra Area. According to the flag, the boot blocks are or are not swapped from the next RESET onward (not immediately on the flag write!). The following sample code shows, how to invert the boot swap flag and program it into the extra area again in order to swap the boot blocks on the next RESET:

```
/* Load the security flags except boot swap flag */
secFlags = ( SelfLib_GetInfo_SecFlags() & ( ~0x00000001 ) );
/* Additionally load the boot swap flag. Based on the API definition it is
   automatically inverted */
secFlags |= ( SelfLib_GetInfo_BootSwap() & 0x00000001 );
/* The boot cluster size may not be set to 0xFF, which is the default value if not touched
   before. If so, we explicitly set it to a reasonable value */
if( ( secFlags >> 24 ) == 0xFF )
   secFlags &= 0x03FFFFFF;
/* Write the modified boot security flags */
ret = SelfLib_SetInfo( secFlags );
```

2. Once a protection flag has successfully been set either by this function or by the Flash programmer beforehand, it cannot be cleared again using this function. Only setting additional flags is possible
The only possibility to clear a flag is to do a Chip Erase by the Flash programmer!

3. Setting the boot cluster protection flag reduces the flexibility in order to increase the Flash protection
- Changing the boot swap flag is not longer allowed
- Boot cluster size cannot be changed any longer.

10.4 Get Device Dependent Information

One central firmware function is used to read device dependent information. The corresponding SelfLib function is SelfLib_GetInfo. In order to simplify the information gathering for the user application, different function like macros have been defined in SelfLib.h, calling SelfLib_GetInfo with appropriate parameters and corresponding result casting. 10.4.1 ”Get information” on page 94 describes the basic function, while the other sub-chapters describe the function like macros.

10.4.1 Get information

Library call:

u32 SelfLib_GetInfo(u32 option)

Required parameters:

option
- 0: Get version information
- 1: Get implemented commands
- 2: Get CPU no. and block no.
- 3: Get protection flag information
- 4: Get boot area swap and Flash macro connection method
- 5~5+block no: Get last block address

Returned value:

Option 0: CCCCCCCCDDDDDDDDAAAAAAAABBBBBBBB
C: device version (high)
D: device version (low)
A: firmware version (high)
B: firmware version (low)

Option 1: Bit0: reserved (Always 0)
Bit1: reserved (Always 0)
Bit2: reserved (Always 0)
Bit3: Block Erase
Bit4: Write
Bit5: reserved (Always 0)
Bit6: Block Internal Verify
Bit7: reserved (Always 0)
Bit8: Block Blank Check
Bit9: Get flash information
Bit10: Set flash information
Bit11: reserved (Always 0)
Bit12: Boot swap
Bit13: reserved (Always 0)
Bit14: FLMD0 check
Bit15: Read
Bit16: reserved (Always 0)
Bit17: Chip Erase
Bit18: Extra area Read
Bit19: Extra area Write
Bit20: One word Write
Bit21: Internal Verify
Bit22: Blank Check
Bits23~31: reserved (0)

Option 2: AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
A: CPU number
B: Block count

Option 3: AAAAAAAAxxxxxxxxxxxxxxxxxxSRWBCx
A: Last boot cluster block number
x: reserved
S: Boot Cluster Write (1: enable, 0: disable)
R: Read command (1: enable, 0: disable)
W: Write command (1: enable, 0: disable)
B: Block Erase command (1: enable, 0: disable)
C: Chip Erase command (1: enable, 0: disable)

Option 4: 0: Boot area is not swapped
1: Boot area is swapped

Option 5~5+Block no.:
End address of the block, specified by block no.

Function description:

This function returns device dependent information according to the option, passed to the function

10.4.2 Get protection flags

Library call:

u32 SelfLib_GetInfo_SecFlags(void)

Required parameters:

-

Returned value:

Bit 0:	Reserved - Always 1
Bit 1:	0: Chip Erase disabled 1: Chip Erase enabled
Bit 2:	0: Block Erase disabled 1: Block Erase enabled
Bit 3:	0: Write disabled 1: Write enabled
Bit 4:	0: Read command disabled 1: Read command enabled
Bit 5:	0: Boot Cluster is protected 1: Boot Cluster is not protected
Bits 6~23:	Reserved for future use
Bits 24~31:	Last boot block number (=Number of blocks in the boot cluster - 1)

Function description:

This function returns a 32 bit value containing above mentioned security related bits. The meaning of the bits is explained in 4.2.2 ”Protection configuration settings” on page 20.

10.4.3 Get block swapping flag

Library call:

u32 SelfLib_GetInfo_BootSwap(void)

Required parameters:

-

Returned value:

0:	Boot swapping is not performed on device startup
1:	Boot swapping is performed on device startup

Function description:

This function returns the value of the boot swap bit (See section 4.3 ”Security” on page 21).

10.4.4 Get device number

Library call:

u32 SelfLib_GetInfo_Device(void)

Required parameters:

-

Returned value:

CPU number in decimal format
e.g. uPD70F3166 (V850ES/SA3) = 3166 (=0x0C5E)

Function description:

This function returns the CPU number, that is part of the device name, in decimal format

10.4.5 Get block count

Library call:

u32 SelfLib_GetInfo_BlockCnt(void)

Required parameters:

-

Returned value:

Number of Flash blocks in the device

Function description:

This function returns the number of Flash blocks in the device

10.4.6 Get block end address

Library call:

u32 SelfLib_GetInfo_BlockEndAdd(u32 blockNo)

Required parameters:

blockNo	Block number to be checked (Not block address, but number of the flash block)

Returned value:

End address of the selected block

Function description:

This function returns the end address of the block passed as a parameter.

10.5 Miscellaneous Functions

10.5.1 Boot swap

Library call:

u32 SelfLib_BootSwap(void)

Required parameters:

-

Returned value:

SELFLIB_OK	Boot swap executed successfully
SELFLIB_ERR_PROTECTION	Flash protection error

Function description:

This function swaps the boot blocks. See also 10.3 ”Protection/Safety Related Operations” on page 92

10.5.2 Check mode (FLMD0)

Library call:

u32 SelfLib_ModeCheck(void)

Required parameters:

-

Returned value:

SELFLIB_OK	V_{DD} is applied at the pin FLMD0
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 8.6 ”Reprogramming Scenario” on page 74)

Function description:

This function explicitly checks the FLMD0 pin and returns the current status.

10.5.3 Read

Library call:

u32 SelfLib_Read(	void *addSrc, void *addDest, u32 len)

Required parameters:

addSrc	Source address in the Flash
addDest	Destination address in RAM
len	number of words (4 Bytes) to read

Note: On devices that can operate on frequencies >64MHz due to the Flash architecture, the granularity of len is 2 words. So, valid values of len are 2, 4, 6, ... but not 1, 3, ...
Furthermore, on such devices addSrc must be aligned to an 8 Byte address

Returned value:

SELFLIB_OK	The function has been executed successfully
SELFLIB_ERR_PARAMETER	Wrong parameters passed to the function, e.g. wrong source address or len

Function description:

The function reads the Flash contents from the device to the destination buffer.

Note: For normal self-programming this function is not required as it is faster to read the data directly by normal memory accesses instead of using this function. This function has been implemented for special serial programmer related applications.

10.6 Alternative Function Names

According to the different international requirements, definitions for alternative function names have been implemented. These names are reflected by defines in the *SelfLib* header file *SelfLib.h*

Table 10-1: Alternative function names

SelfLib function name	Alternative function name
SelfLib_BlankCheck	FlashBlockBlankCheck
SelfLib_Erase	FlashBlockErase
SelfLib_IVerify	FlashBlockIVerify
SelfLib_BootSwap	FlashBootSwap
SelfLib_Init	FlashInit
SelfLib_ModeCheck	FlashFLMDCheck
SelfLib_GetInfo	FlashGetInfo
SelfLib_StatusCheck	FlashStatusCheck
SelfLib_SetInfo	FlashSetInfo
SelfLib_Write	FlashWordWrite
SelfLib_Read	FlashWordRead
SelfLib_LibVersion	FlashLibVersion
SelfLib_FctNewAddress	FlashFctNewAddress

[MEMO]

Chapter 11 SST (MF2/UX4) - Self-Programming

In addition to the Code Flash some devices are also equipped with Data Flash. The data Flash has different features to the Code Flash and is treated differently. The self-programming library is designed to re-program the Code Flash only. For any operations on Data Flash, please refer to a special library called DFALib and the appropriate documentation.

11.1 SelfLib Functionality

The SelfLib itself contains all functions important for self-programming except the control program which is application specific and therefore has to be made by the user:

- *Flash environment* Activation / Deactivation (See below)
- User interface
  This contains the functions to be called by the user Flash reprogramming control program
- Firmware interface

The *SelfLib* calls the device internal reprogramming firmware via the firmware interface. The firmware is located in a device internal ROM or Flash, not visible in the normal address range. This firmware does the really hardware related operations and is fix in order to ensure, that as few as possible user changeable functions endanger the Flash and its contents.
If the firmware is located in Flash (e.g. on 70F3441/3444), the code is automatically copied to the device RAM in order to be executed. This procedure is transparent for the user, but a certain RAM address space is reserved during self-programming. The reserved address space is mentioned in the device users manual or the *DeviceInfo*

11.1.1 Flash Environment

The *Flash environment* is additional hardware (charge pumps, sequencer,...), required to do the Flash re-programming. When activated for Flash re-programming, the Flash is not accessible by normal read accesses, so code execution is not possible from Flash.
Depending on the library setting, the Flash environment (de)activation is done either implicitly by the library or need to be done explicitly by the user program. See also 11.6 ”User Application Execution During Self-Programming” on page 110

Note: The Flash is not accessible for normal code and data fetches while the *Flash environment* is active. So parts of the SelfLib (and firmware, see above) need to be executed outside the Flash (e.g. in embedded RAM).

11.2 Device Reprogramming Overview

First of all it is important to select the correct reprogramming flow. Please check with the application and with the *DeviceInfo*, whether secure reprogramming and secure bootloader update is required and available or not (See section 4.3.2 ”Secure reprogramming using self-programming” on page 22).
If secure bootloader update is required, please refer to 11.5 ”Secure Reprogramming Flow with bootloader Update” on page 107 for the correct flow.

In the following, the basic steps are explained. The next sub chapters then explain the standard block reprogramming and the secure bootloader reprogramming. The standard block reprogramming flow is also embedded in the secure bootloader reprogramming.

The following principle steps have to be executed for any device reprogramming:

Table 11-1: Basic Steps

Step	Function	Description
1	Setup the user program	- Setup communication Interface, watchdog timer, FLMD0 voltage activation,...
2	Self-Programming initialisation	See section 13.1 ”Initialisation” on page 123.
3	Reprogram the Flash	Take care to keep the correct sequence for reprogramming single Flash blocks (see section 11.3 ”Basic Block Reprogramming Flow” on page 105) and for secure bootloader reprogramming (see section 11.5 ”Secure Reprogramming Flow with bootloader Update” on page 107).
4	End Reprogramming	Return to bootloader, start application, reset or...

11.3 Basic Block Reprogramming Flow

This is the standard flow, that shall be used to reprogram all Flash blocks.

To ensure, that the data is written correctly and the data retention is given it is important to keep the following sequence for block reprogramming. Whenever a function returns an error, you may not continue the sequence because the next steps may not work correctly and the result is undefined.

Table 11-2: Block Reprogramming Sequence

Step	Flash Lib. Function	Description
1	Block Erase	The Flash block is erased
2	Write	Data is written to the Flash
3	Internal Verify	This step is executed after the last Write to a dedicated block to ensure the specified data retention of the Flash The Internal Verify is a check for higher reliability of the programming, but no manipulation or operation on a 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, Vdd/Gnd bounce)

11.4 Extra Information Handling

Setting protection flags and variable Reset vector

It is possible to set the protection flags and the vector either during the initial programming with a dedicated Flash programmer (e.g. PG-FP4) or during self-programming using the SelfLib. Following that high flexibility during development, but also during mass production is given.

Boot swap flag

If the flag is set, a boot swap is executed. In that case the upper boot block cluster (See section 4.2.2 ”Protection configuration settings” on page 20) is swapped to 0x00000000, while the other cluster is swapped to the upper location.

Extra information handling during self-programming

The extra area containing the different flags is treated completely independent. Following that the flags need only be reprogrammed, when the boot swap flag is updated or when the protection settings are to be changed. Differing from UC2, the extra area is not affected by any Block Erase operation.
Following that, when normal reprogramming is done without secure bootloader update (See section 11.3 ”Basic Block Reprogramming Flow” on page 105), only the normal block reprogramming flow need to be considered, even with respect to the protection flags.
See 13.3.1 ”Set protection flags, boot cluster size and variable reset vector” on page 130 for an example on how to set the flags.

Notes: 1. In order to only increase the protection level set before, it is possible to set additional restrictions for the Flash programming but not to remove restrictions

- Any protection flag can only be set, not be cleared
- The boot cluster size can not be changed, if the block protection flag is set
- The boot swap flag can not be changed, if the block protection flag is set

2. A flag is set by writing it to 0 while a cleared flag has the value 1.

11.5 Secure Reprogramming Flow with bootloader Update

The basic feature allowing this way of secure reprogramming is, that on device start-up is checked which block contains a valid bootloader. To do so the boot swap flag in the extra area(s) is analysed.
By the boot swap feature and the correct reprogramming sequence it can be assured, that always a valid boot program is started to continue the reprogramming sequence in case that the reprogramming algorithm was interrupted (Power fail, accidental reset).
The bootloader must be able to control the complete reprogramming sequence including data transfer of the new Flash contents.

The sample configuration considers a device with 8 blocks and with boot swap capability, where the blocks 0/1 can be swapped with 2/3.

Note: The block number in the diagrams below are the physical block numbers. They are always fix, even after block swapping. E.g.: block 2 before boot swapping remains block 2 even after the swap. This is done for better illustration.
When using the SelfLib functions, logical block numbers are used. On logical block numbers the block on address 0x00000000 is always block 0, followed by block 1 and so on. This is valid even after block swapping. E.g.: Block 2 before swapping is block 0 after swapping. By that software development is easier, as the block numbers need not be modified depending on the swapping.

Table 11-3: SST(MF2) Flash Secure Reprogramming Sequence Incl. bootloader Update (1/2)

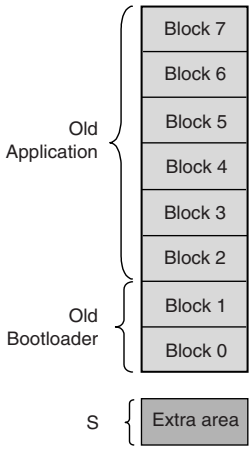
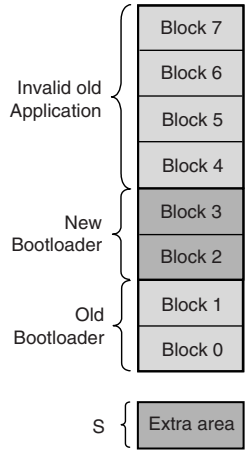
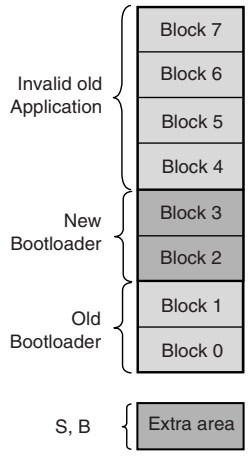
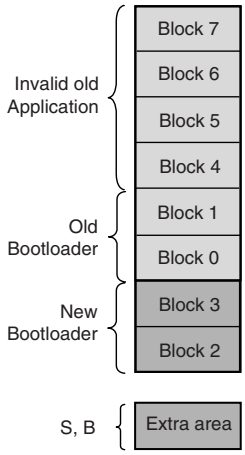
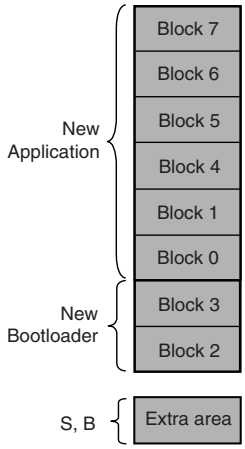
Step	SelfLib Function	Description	Sample Configuration
1	Initial Status	• Boot Program in the lower 2 areas • Safety flags in the extra area set (e.g. Write disable, Block Erase disable) • Boot flag in the extra area not set. So Blocks 0/1 are located at 0x0000000 • Blocks 2~7 contain the application	
2	Reprogram blocks 2/3 with the new boot-loader	The new bootloader is written to the blocks, that shall be swapped to 0x0000000 later on Execute the complete block reprogramming flow for both blocks (See 11.3 ”Basic Block Reprogramming Flow” on page 105): • Erase • Program • Internal Verify (See also 11.3 ”Basic Block Reprogramming Flow” on page 105) When erasing the first application program block the application is no longer valid and able to run. In case of power fail, the boot-loader handles the complete reprogramming flow.	
3	Rewrite security flags	General rule regarding the boot flag is, that it has to be inverted in order to swap the blocks. In this example the boot flag needs to be set After writing the security flags and boot flag, the new bootloader in blocks 2 and 3 would be active in case of a device restart (e.g. due to power failure)	

Table 11-3: SST(MF2) Flash Secure Reprogramming Sequence Incl. bootloader Update (2/2)

Step	SelfLib Function	Description	Sample Configuration
4	Swap blocks 0/ 1 and 2/3	After swapping the blocks, the begin of the new bootloader (block 2) is located at 0x00000000	
5	Reprogram the blocks 0, 1, 4~7	Execute the complete block reprogramming flow for each block separately (See 11.3 ”Basic Block Reprogramming Flow” on page 105): • Erase • Program • Internal Verify (See also 11.3 ”Basic Block Reprogramming Flow” on page 105) Note: Pass the logical block numbers (not physical block numbers mentioned in the diagram on the right side) as parameters to the SelfLib functions requiring block numbers!	

11.6 User Application Execution During Self-Programming

The reprogramming flow is set-up in the way, that a user written self-programming control program controls the basic reprogramming flow and the SelfLib calls. The SelfLib functions interact with the Flash. According to the called SelfLib function the execution requires times from a few nanoseconds up to several seconds. In many applications it is acceptable, that no further software execution during that time is possible. This is the easiest way of a reprogramming sequence.
However, for most applications it is not acceptable, that no more code can be executed during execution of long lasting SelfLib functions (like the Erase function).

The MF2-SST Flash devices are designed in a way, that long lasting Flash manipulation operations only have to be initiated and then are executed in the background. Completion of the operations need to be checked in order to continue the self-programming. By that, in order to support application software execution during self-programming the SelfLib can be set-up to offer two major options, if necessary both in parallel:

- Interrupt function execution
- Function execution while self-programming status polling

The define SELFLIB_STATUS_CHECK (See 12.4 ”Global SelfLib Defines - SelfLibSetup.h” on page 122) defines how the self-programming background operations are checked.
Two options are possible:

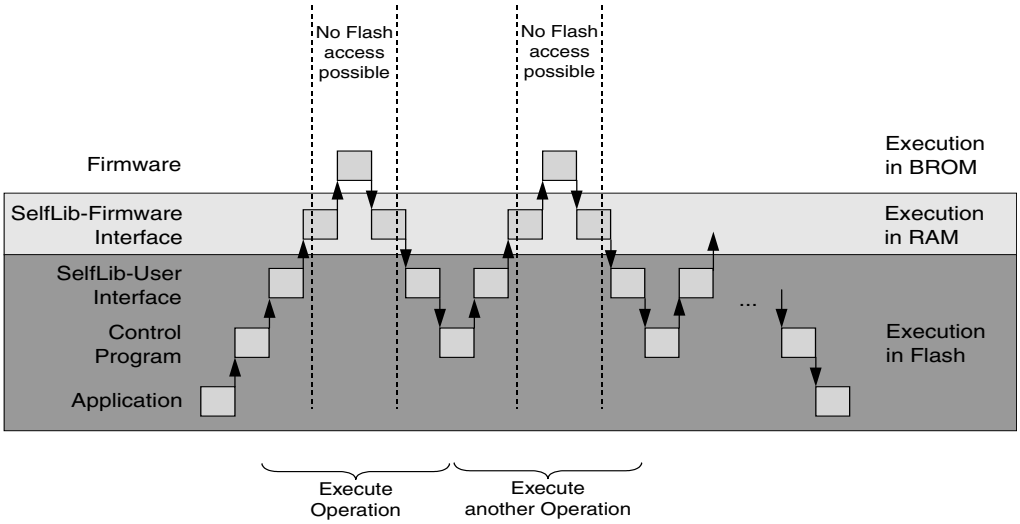
- *Internal status check*
  The status is checked internally by the *SelfLib*. The operations (e.g. Erase) don’t return until the Flash operation is finished. User code execution is possible only by interrupt functions.
  The internal RAM usage for self-programming is low, as only the firmware interface and interrupt functions are copied to the device RAM
  This option is described in the chapter 11.6.1 ”Internal status check” on page 111.
- *Status check by user*
  The status check is done in the user self-programming control program after initiating the Flash background operations. User code can be executed in the loop that checks the self-programming status.
  Additionally execution of interrupt functions is possible too.
  The RAM usage is higher, as the user self-programming control program, the SelfLib user and firmware interface are executed in the RAM.
  This option is described in the chapter 11.6.2 ”User status check” on page 112.

11.6.1 Internal status check

The user application, containing the re-programming control program initiates a programming operation (e.g. Erase) by calling the appropriate SelfLib function. The *SelfLib* passes control to the device internal firmware, which completely executes the operation and returns to the SelfLib at the end. The SelfLib returns then control immediately the control program.
The *Flash environment* is being activated implicitly by the SelfLib functions. Therefore, (de-)activation by the user program is not necessary and even more not allowed as it may lead to unpredictable program behaviour.

As the *Flash environment* is always deactivated when returning to the normal user code, the *SelfLib* user interface and the user self-programming control program need not be copied into the RAM. Only the sections containing the interrupt functions and the firmware interface are automatically copied to the RAM by the library initialisation if requested (See 13.1.1 ”Library initialization” on page 123). Following that, the RAm usage is very low, depending on the user interrupt functions.

Figure 11-1: Execution Scenario on Internal Status Check



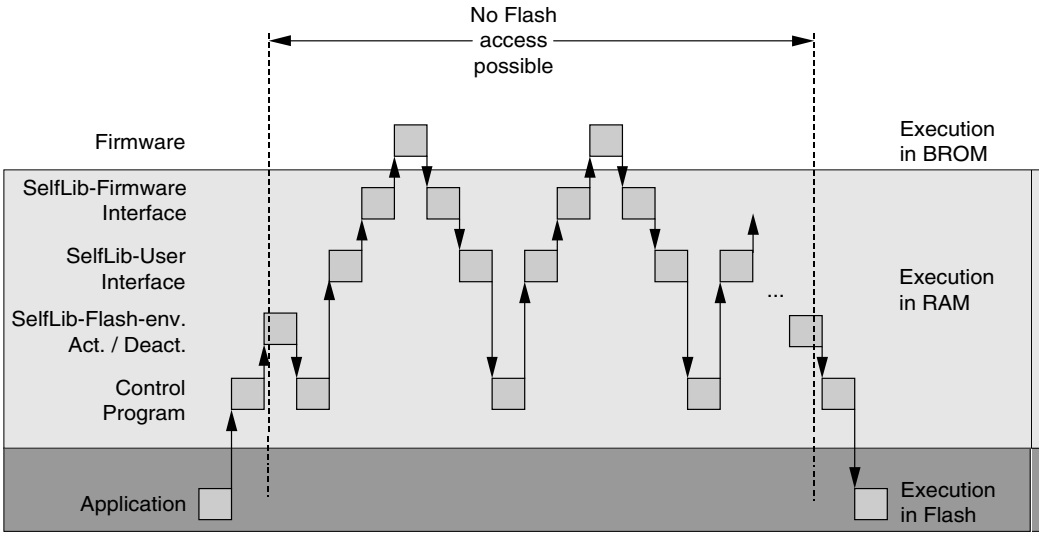
11.6.2 User status check

The user application, containing the re-programming control program initiates a programming operation (e.g. Erase) by calling the appropriate SelfLib function.
While functions with shorter execution time are completely executed within the library and firmware, Flash operations with longer execution time are only initiated as background operations. The control is then given back to the user program, that then calls the status check function to poll the status of the background operation. While polling (e.g. in a loop) other user code can be executed.

The user self-programming control program including the status polling must be executed on activated *Flash environment*. Following that, the user code, the SelfLib user and firmware interface are executed in the RAM. Due to that, the RAM usage in this mode is higher.
Additionally the *Flash environment* activation and deactivation functions must be explicitly called by the user program at the beginning respectively at the end of the self-programming.

Additionally to the code execution while status polling, also interrupt functions can be executed.

Figure 11-2: Execution Scenario on User Status Check



11.6.3 Execution latencies

The execution latencies of user functions during self-programming differ depending on, if the functions are executed as interrupt functions or during status polling.

Interrupt functions
The interrupt function latency is determined by the time frame required to activate or deactivate the *Flash environment*. During that time neither Flash nor the device internal firmware is available; no interrupt service is possible, the interrupts are disabled by the *SelfLib.* A interrupt request is delayed until the activation/deactivation is finished. The latency is measured on the different devices and mentioned in the device information. However, it is always quite low (e.g. <50 µs on the V850ES/SA3).

Status polling
The latency during user status polling is determined by the execution time of the *SelfLib* functions.
Due to the fact, that many *SelfLib* functions initiate only background operations, most function execution times are below 200 µs. However, a few functions need to be considered separately:
SelfLib_SetInfo
Around 5 ms have to be considered for the execution of this function.
As setting the security flags is not necessarily a part of the normal reprogramming flow, it need to be checked case by case, whether the latency is relevant. This execution time is not mentioned as latency in the *device information.*

SelfLib_Write
The Write function is no background operation. The control is given back to the calling user code after the words are written. So, it is recommended to write only single words and return to the user code afterwards. The maximum execution time of this function is determined by the maximum write time of one Flash cell plus some software overhead. This time is mentioned as latency in polling mode in the *device information.*

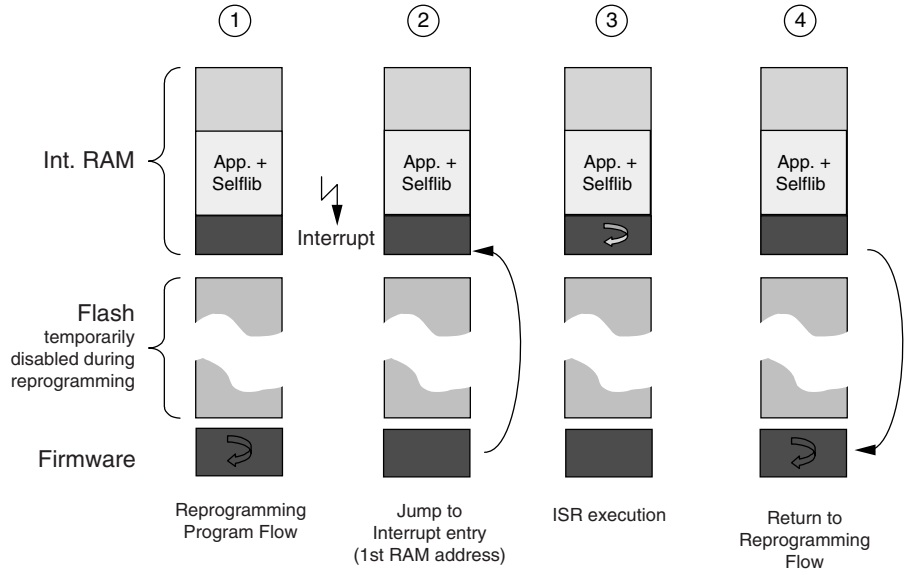
11.6.4 Interrupt functions

This sub-chapter explains the Interrupt function execution during self-programming.

On deactivated *Flash environment* the interrupt vector table is available and the normal user defined interrupt functions can be executed. When the environment is activated, this is not the case. As the Flash including the interrupt vector table is not accessible at that time, normal jumps into the vector table are not possible. Instead, the interrupt jump is handled by the device internal firmware in the following way:

- All interrupt vectors are relocated to an entry point in the internal RAM. No register handling or mode change is done by the firmware. The user handler must be designed as interrupt function including register save/restore and reti at the end
- New entry point of **all non maskable interrupts** is the 1st address of the internal RAM (depending on the device, e.g. 0xFFFFE000). A user handler routine must check the interrupt source. The source can be read from the exception cause register ECR (See device users manual).
- New entry point of **all maskable interrupts** is the word address following the masked interrupt entry (depending on the device, e.g. 0xFFFFE004). An user handler routine must check the interrupt source. The source can also be read from the exception cause register (See device users manual).

Figure 11-3: Interrupt Routine Execution in Internal RAM / External Memory



The SelfLib is designed in order to ease the interrupt user handler implementation. This is done by special treatment of the two SelfLib sections SelfLib_ToRamUsrInt and SelfLib_ToRamUsr (See also 11.7 ”SelfLib Sections” on page 116). These sections are copied to the destination address by the SelfLib_Init function, if the copy functionality is configured.

- The section SelfLib_ToRamUsrInt is the first copied section. In the application sample it contains the user interrupt entry as the only functionality.

- The section SelfLib_UsrInt is copied by SelfLib_Init immediately behind the SelfLib_UsrInt1st (4Byte alignment). It is designed to contain the interrupt user function.

Notes: 1. The RAM destination address, passed to SelfLib_Init function must be the 1st RAM address (interrupt entry on activated Flash environment) as the section SelfLib_ToRamUsrInt need to copied there!

2. As Normal function calls from section SelfLib_UsrInt1st to SelfLib_UsrInt are intended, the sections must also be linked immediately after another with a 4Byte alignment. Please set up the linker control file accordingly!

3. The execution latency of interrupt routines on activated *Flash environment* is increased by the fact, that the interrupts are passed to the first RAM addresses by the device internal firmware. However, the increase is quite low and should not exceed 100us on maximum device frequency.

11.7 SelfLib Sections

In order to support the different environments and reprogramming sequences, the library is splitted up into several sections.

The following library sections exist:

- SelfLib_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 initialisation is part of this section.

- SelfLib_RomOrRam
  This section contains the user interface. Depending on the reprogramming scenario (See section 11.6 ”User Application Execution During Self-Programming” on page 110) this section is copied into the RAM or not. The copy procedure is automatically executed by the library initialisation (See section 10.1 ”Initialisation” on page 85).
- SelfLib_ToRam
  This section contains the firmware interface and so need to be executed outside the reprogrammed Flash. As it is usually copied to the internal RAM, it is called SelfLib_ToRam.

- SelfLib_ToRamUsrInt
  This section contains the Interrupt function entry during activated *Flash environment*.
  The device internal firmware passes all acknowledged interrupts to the 1st RAM addresses, where the interrupt function entry routine need to be located.
  In order to achieve this, the *SelfLib* initialisation copies this section to the SelfLib destination address first. Following that, the SelfLib destination address must be the first RAM address!

  Even if not used and empty, the section must exist, as address references to this section are used in the SelfLib initialisation.

- SelfLib_ToRamUsr
  This section contains user functions to be executed in RAM, where it is copied by the *SelfLib* initialisation
  Part of these user functions are the Interrupt functions, called by the interrupt entry in the SelfLib_ToRamUsrInt section. Furthermore, this section may also contain other user functions, like the self-programming control program, etc.
  Even if not used and empty, the section must exist, as address references to this section are used in the SelfLib initialisation.

Note: Function calls on V850 devices are realized as relative jumps. Following that, the linked relative address distance between SelfLib_ToRamUsrInt (Interrupt entry) and SelfLib_ToRamUsr (Interrupt function) may not change before and after copy to RAM. As the section SelfLib_ToRamUsr is copied immediately (4 Byte alignment) behind SelfLib_ToRamUsrInt by SelfLib_Init, the user must also consider this section sequence in the linker directive file.

- SelfLib_RAM
  This section contains the variables required for SelfLib. It can be located to internal or external RAM.

11.7.1 Automatic section copy

The application, including the control program and the SelfLib 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. The control program, as well as the user interface of the SelfLib may be located in Flash. Only the firmware interface of the SelfLib and user interrupt functions must be located outside the Flash on execution.

If the copy functionality of the SelfLib_Init function is configured, the following sections, containing these functions, are copied automatically to the destination address in RAM (See 13.1.1 on page 123):

- SelfLib_ToRamUsrInt
- SelfLib_ToRamUsr
- SelfLib_ToRam
- SelfLib_RomOrRam (If user status polling is activated. See 11.6.2 ”User status check” on page 112)

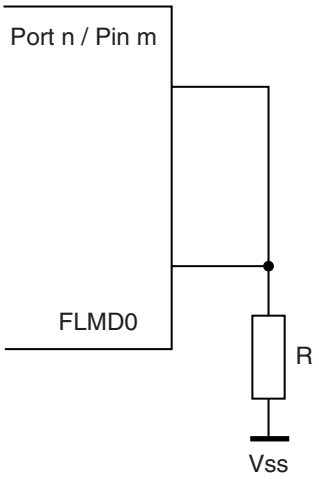
Notes: 1. The RAM destination address, passed to SelfLib_Init function must be the 1st RAM address (interrupt entry on activated Flash environment) as the section SelfLib_ToRamUsrInt need to copied there!

2. The copy functionality of the SelfLib_Init function has a drawback. As the code location differs from the linked position, source level debugging is not possible inside the copied code. For the *SelfLib* code this is not a big problem, as it is well tested, but for user code this need to be considered.
Alternative (for GHS) solution could be:
Similar to the .data section. Link the code to the destination address and use the possibility of the GHS startup routines to copy the code to the RAM from a ROM image. Please refer to the GHS documentation, “ROM linker section attribute”.

11.8 Hardware Requirements

The requirements of self-programming regarding the device external hardware is very low as no dedicated external programming voltage is required.
However, it has to be considered, that a security concept against unintended reprogramming has been set-up. It is required, that the devices FLMD0 pin is externally attached to V_{DD} (I/O voltage) during self-programming. FLMD0 should be attached to V_{DD} immediately before or after SelfLib initialisation (See the device specific self-programming application sample) and reset to V_{SS} at the end of self-programming. Default state on reset is V_{SS}.

Figure 11-4: 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}.

Chapter 12 SST (MF2/UX4) - SelfLib Configuration

SelfLib implementation relevant issues are discussed in the following sub-chapters.

12.1 Used Resources

The only resources used by the self-programming, is memory.

Beside the SelfLib sections the stack is used. The stack usage is depending on the library and on the used compiler environment. Furthermore, also the device firmware uses the stack.

The complete FLASH is not available at time during self-programming.

12.2 Software Considerations

This chapter lists up special issues to be considered when implementing self-programming into the application.

DMA operation during self-programming
DMA operations during self-programming (While the Flash environment is activated) are not allowed at all. Background is the fact, that accesses sequences to protected SFR registers may fail in case of DMA accesses to the peripheral bus during this sequence

Run-Time Library calls
During Self-Programming (activated Flash Environment) It is not possible to call run-time library functions, if the library is not copied to RAM together with the Self-Programming code.
In case of program failures during self-programming, the code should be explicitly checked for such calls.

Switch instruction in C
In case of the user status check, the switch instruction should be avoided in all the code executed in RAM. Depending on the used compiler this instruction may result in a call to a run-time library, which has not been copied.

12.3 Compiler Configuration

As the SelfLib is delivered in source code, the user has to take care for the correct compiler settings in his applications. Due to the huge amount of setting options, not all could be tested. The settings coming along with the application samples should be taken as reference for the user application development.

12.3.1 Project settings

The most important settings for a successful application build shall be explained in the following:

Position independent code (PIC)
The SelfLib code shall be compiled position independent.

Inline function prologue and epilogue
In order to avoid any library calls for function prologue and epilogue into a section, not available during self-programming (Flash), they need to be unlined.

V850E core instruction set
The extended instruction set of the V850E core is used for the assembler parts. So this core should be selected.

The settings are only required for the complete SelfLib and all code added to the SelfLib sections. The rest of the project may have different settings. However, it is recommended to use the same settings project wide, if possible.

System calls
The linkers can add small code fragments and appropriate labels for so called system calls. These calls are required to do e.g. terminal-io or file-io. Whenever such an operation is required, the program jumps to the special code. The debugger detects this by a debug break and executed special operations. As partially device firmware is executed on the same addresses as the normal program code, the firmware might run into the debug break too, resulting in unexpected program behaviour. This must be avoided by disabling certain debug functionality.

Green Hills (GHS) Multi 2000

To set the settings project wide, click on the project file in the project window to select it before entering the menu.

- Select position independent code
- Select inline prologue
- Select the processor type "V850E1" resp. "V850E/MS1" for V850E core instruction set

Further required settings:

- Set the following driver options for the assembler files to ensure, that c-style preprocessing directives are processed:
  Click on an assembler file to select it. Then select the menu “Project -> File options”. In the driver options field type in “-preprocess_assembly_files”. Do this for each assembler file.
- Disable usage of the callt instruction, as it results in calls to the run-time library, which is not copied into the RAM in case of time saving scenario.

System calls can be avoided inside the debugger.

- Type in the debugger cmd window: target syscalls off
  This can also be added to the .rc or the .mbs debugger control files.

IAR Embedded Workbench

Using the IDE, the settings can be found in the menu “Project -> Options”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- Category ICCV850: Optimize for speed. By that, the function prologue/epilogue is inlined
- Category general: Select CPU variant V850ES or V850E for the V850E core instruction set
- Category Linker->Output->”Debug Information for C-Spy”: Deselect “with runtime control modules” to avoid system calls

No explicit setting required for position independent code.

NEC CA850

Using the IDE, the following can be configured in the menu “Tool -> Compiler Options” under the tab “Others”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- In the box "Any Option" type in “-Ot” for inline prologue/Epilogue

Using the IDE, the following can be configured in the menu “Tool -> Linker Options” under the tab “Library”. To set the settings project wide, click on the top of the project tree in the project window before entering the menu.

- De-select "Link standard library" and de-select "link mathematics library"

No explicit setting required for position independent code.
The core selection is done during project initialization by selecting the correct target device.

No explicit settings for system calls required, as these are not supported by the NEC compiler

12.4 Global SelfLib Defines - SelfLibSetup.h

The *SelfLib* is optimized for code size. This is realized by scenario dependent defines. Setting these defines controls the compilation of the library. Differing from setting SelfLib options by function parameters (like in the UC2 *SelfLib*), unused code will so be avoided and the code size will be reduced.

The defines are located in the file **SelfLibSetup.h**. This file is included in the different *SelfLib* c-modules and should therefor be located in the same directory.

SELFLIB_STATUS_CHECK
As explained in chapter 11.6 ”User Application Execution During Self-Programming” on page 110, this define configures the status check for Flash background operations.

- STATUS_CHECK_INTERNAL Defines the internal status check. User code execution during self-programming is possible by interrupt functions only
- STATUS_CHECK_USER Defines the user status check. User code execution during self-programming is possible by interrupt functions or in the status polling loop

SELFLIB_SECTIONS

This define configures whether the sections to be executed outside the Flash (e.g. SelfLib_ToRam,....) are automatically copied to another location by the function SelfLib_Init (See 11.7.1 ”Automatic section copy” on page 117 and 13.1.1 ”Library initialization” on page 123).

- SELFLIB_COPY The sections are automatically copied by the SelfLib_Init function
- SELFLIB_DONT_COPY The sections are not copied.

Chapter 13 SST (MF2/UX4) - SelfLib API

The SelfLib contains functions of several different categories. These are called by the user program in the sequences as described in the last chapters. The library function calls and parameters are explained in the following.

13.1 Initialisation

Functions explained in this chapter are called at the beginning of the self-programming. They are necessary for the SelfLib initialisation, but some functions can also be used for set-up of a reprogramming control program.

13.1.1 Library initialization

Library call:
The function call is different, depending on the *SelfLib* setup (See 12.4 ”Global SelfLib Defines - SelfLibSetup.h” on page 122).

SELFLIB_SECTIONS defined as SELFLIB_COPY
void SelfLib_Init(void *destAddSelfLib)

SELFLIB_SECTIONS defined as SELFLIB_DONT_COPY
void SelfLib_Init(void)

Required parameters:

destAddSelfLib	The *Relocated Sections* are copied to this location. See 11.7 ”SelfLib Sections” on page 116

Caution: If interrupt functions shall be executed during self-programing (See section 11.6 ”User Application Execution During Self-Programming” on page 110), the address must be set to the beginning of the internal RAM (e.g. 0xFFFFE000), as the interrupt handler entry is there.

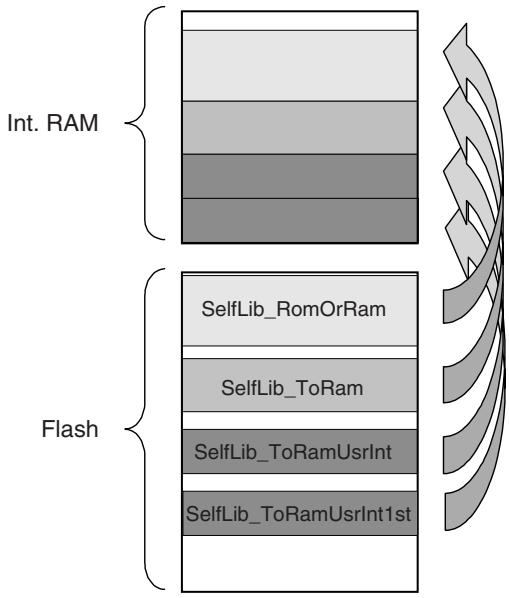
Returned value:
-

Function description:
The function copies the sections into the RAM if requested (see above).

The SelfLib_ToRamUsrInt section is copied first (to destAddSelfLib), in order to enable the user to place it on a defined address for easy interrupt handling support (See section 11.6.4 ”Interrupt functions” on page 114).
Directly behind that SelfLib_ToRamUsr is copied, followed by SelfLib_ToRam and SelfLib_RomOrRam. The copy routine doesn’t leave any holes between the copied sections.

In addition to the section copy the SelfLib data and internal pointers are initialised.

Figure 13-1: Section Copy in SelfLib Initialisation



13.1.2 New function address

Library call:

```
u32 *SelfLib_FktNewAddress (    void *addFct,
                                void *destAddSelfLib )
```

Required parameters:

addFkt	Original Function address.
destAddSelfLib	Destination address in RAM. See section 13.1.1 ”Library initialization” on page 123

Returned value:

Address of the function inside the SelfLib_RomOrRam or the SelfLib_ToRamUsr Section on its destination address.
If the function is not in any *SelfLib* section, 0x00000000 is returned as error value.

Function description:

If a user function is located in the section SelfLib_RomOrRam or SelfLib_ToRamUsr it may be copied to a save RAM location by SelfLib_Init. If the function has then to be called from outside of the section where it is located, the normally used relative function calls don’t work any longer.Following that, it has to be called by a function pointer. To set-up the function pointer SelfLib_FktNewAddress returns the new address of the function.

13.2 Basic Reprogramming

Basic reprogramming functions are related to the standard block reprogramming (See section 11.3 "Basic Block Reprogramming Flow" on page 105). These functions support the operations Blank Check, Erase, Write, Internal Verify.

13.2.1 Area Blank Check

Library call:

u32 SelfLib_BlankCheck(	u32 blockNoStart, u32 blockNoEnd)

Required parameters:

blockNoStart	First Block number to be checked (Not block address, but number of the flash block)
blockNoEnd	Last Block number to be checked (Not block address, but number of the flash block)

Returned value:

The possible return values differ depending on the configured status check scenario (See 11.6 "User Application Execution During Self-Programming" on page 110 and 12.4 "Global SelfLib Defines - SelfLibSetup.h" on page 122).
In case of user status check, this function only initiates a Flash background operation. Afterwards, the user control program need to do status checks to poll for the end of the background operation and to receive the operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn).
If the Flash operation is not called or called not long enough, a following Flash operation will return SELFLIB_ERR_FLOW.

SELFLIB_OK	In case of internal status check: Blocks are blank In case of user status check: Background operation started
SELFLIB_ERR_PARAMETER	blockNoStart or blockNoEnd is invalid
SELFLIB_ERR_FLASHPROCn	Only possible in case of internal status check: (n = 0x00~0x2). At least one block is not blank
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function checks, if a range of blocks is blank (erased) or not.

Note: The Blank Check confirms that all cells within the checked flash area still have the correct margin level (erase level). A failure of the Blank Check does not imply a flash problem as such, it only indicates the necessity to perform an Erase operation to ensure proper start conditions before the programming of the flash memory.

13.2.2 Area Erase

Library call:

u32 SelfLib_Erase (	u32 blockNoStart, u32 blockNoEnd)

Required parameters:

blockNoStart	First Block number to be erased (Not block address, but number of the flash block)
blockNoEnd	Last Block number to be erased (Not block address, but number of the flash block)

Returned value:

The possible return values differ depending on the configured status check scenario (See 11.6 "User Application Execution During Self-Programming" on page 110 and 12.4 "Global SelfLib Defines - SelfLibSetup.h" on page 122).
In case of user status check, this function only initiates a Flash background operation. Afterwards, the user control program need to do status checks to poll for the end of the background operation and to receive the operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn). If the Flash operation is not called or called not long enough, a following Flash operation will return SELFLIB_ERR_FLOW.

SELFLIB_OK	In case of internal status check: Blocks are erased successfully In case of user status check: Background operation started
SELFLIB_ERR_PARAMETER	blockNoStart or blockNoEnd is invalid
SELFLIB_ERR_PROTECTION	Flash protection error. The error is generated, if the boot cluster protection bit is set and the Erase shall be applied to a boot block. See 4.2.2 "Protection configuration settings" on page 20
SELFLIB_ERR_FLASHPROCn	Only possible in case of internal status check: (n = 0x00~0x2). At least one block could not be erased
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function erases a range of Flash blocks.

The time required for erasing strongly depends on the used device, no. of already done erase cycles, temperatures and other conditions. It is not constant and might last up to several seconds in worst case.

Note: Device internally, the Block Erase function can erase certain sets of blocks in parallel, reducing the erase time significantly. If the blocks are within one Flash macro and the range is 2^n and the alignment is according to the size, these blocks are erased in parallel.
If the range, passed to the SelfLib_Erase function, does not fit the criteria for parallel erase, the range is split up automatically by the SelfLib and device firmware into fitting sub-ranges, which are then erased sequentially. As this is done automatically, the user control program does not have to take care for that.
Example:
blockNoStart = 3
blockNoEnd = 87
Erase is internally split up into: Erase 3 --> Erase 4~7 --> Erase 8~15 --> Erase 16~31 --> Erase 32~63 --> Erase 64~79 --> Erase 80~87

13.2.3 Write

Library call:

u32 SelfLib_Write (	void *addSrc, void *addDest, u32 length)

Required parameters:

addSrc	Address of the source data that has to be written into the Flash
addDest	Destination where the data has to be written (The address alignment strongly depends on the used device. Please refer to the device information for the alignment)
Length	Number of WORDS to be written. According to the device and the Flash technology dedicated limitations for the Write granularity and maximum length are given. Please refer to the *device information* for the exact limitations. Values working generally are: - All V850 devices: length = 64 words (*32 bits)

Returned value:

SELFLIB_OK	Data is successfully written
SELFLIB_ERR_PARAMETER	Address parameter error
SELFLIB_ERR_FLASHPROCn	(n = 0x0~0x2). Write failed
SELFLIB_ERR_PROTECTION	Flash protection error. The error is generated, if the boot cluster protection bit is set and the Erase shall be applied to a boot block. See 13.3.1 ”Set protection flags, boot cluster size and variable reset vector” on page 130
SELFLIB_ERR_FLOW	Only possible in case of user status check (See 11.6 ”User Application Execution During Self-Programming” on page 110 and 13.7.3 ”Status check” on page 142): Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

Data is written to the Flash. This is done word wise.

13.2.4 Internal Verify

Library call:

u32 SelfLib_IVerify (	u32 blockNoStart, u32 blockNoEnd)

Required parameters:

blockNoStart	First Block number to be checked (Not block address, but number of the flash block)
blockNoEnd	Last Block number to be checked (Not block address, but number of the flash block)

Returned value:

The possible return values differ depending on the configured status check scenario (See 11.6 ”User Application Execution During Self-Programming” on page 110 and 12.4 ”Global SelfLib Defines - SelfLibSetup.h” on page 122).
In case of user status check, this function only initiates a Flash background operation. Afterwards, the user control program need to do status checks to poll for the end of the background operation and to receive the operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn). If the Flash operation is not called or called not long enough, a following Flash operation will return SELFLIB_ERR_FLOW.

SELFLIB_OK	In case of internal status check: Int. Verify passed successfully In case of user status check: Background operation started
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 11.8 ”Hardware Requirements” on page 118)
SELFLIB_ERR_PARAMETER	blockNoStart or blockNoEnd is invalid
SELFLIB_ERR_FLASHPROCn	Only possible in case of internal status check: (n = 0x00~0x2). Internal Verify failed
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function compares the Flash contents on erase level against the write level. See also 11.3 ”Basic Block Reprogramming Flow” on page 105

13.3 Protection/Safety Related Operations

13.3.1 Set protection flags, boot cluster size and variable reset vector

Library call:

u32 SelfLib_SetInfo(u32 SecFlags,
u32 resetVctAdr)

Required parameters:

SecFlags — 32-bit value containing the following bits:

Bit 0:	0: The boot clusters are swapped on Reset 1: The boot clusters are not swapped on Reset
Bit 1:	0: Chip Erase disabled 1: Chip Erase enabled
Bit 2:	0: Block Erase disabled 1: Block Erase enabled
Bit 3:	0: Write disabled 1: Write enabled
Bit 4:	0: Read command disabled 1: Read command enabled
Bit 5:	0: Boot Cluster is protected 1: Boot Cluster is not protected
Bits 6~23:	Reserved for future use
Bits 24~31:	Last boot block number (=Number of blocks in the boot cluster - 1)

Unsupported Flags or reserved flags have to be set to 1

resetVctAdr — Reset vector address: Address of the first executed instruction after Reset.

Returned value:

The possible return values differ depending on the configured status check scenario (See 11.6 ”User Application Execution During Self-Programming” on page 110 and 12.4 ”Global SelfLib Defines - SelfLibSetup.h” on page 122).
In case of user status check, this function only initiates a Flash background operation. Afterwards, the user control program need to do status checks to poll for the end of the background operation and to receive the operation result (SELFLIB_OK or SELFLIB_ERR_FLASHPROCn). If the Flash operation is not called or called not long enough, a following Flash operation will return SELFLIB_ERR_FLOW.

SELFLIB_OK	In case of internal status check: Flag setting was successful In case of user status check: Background operation started
SELFLIB_ERR_PARAMETER	e.g.: Already set flags shall be cleared or resetVctAdr is set to 0xffffffff
SELFLIB_ERR_PROTECTION	Flash protection error. The error is generated, if the boot cluster protection bit is set and the Erase shall be applied to a boot block. See 4.2.2 ”Protection configuration settings” on page 20
SELFLIB_ERR_FLASHPROCn	Only possible in case of internal status check: (n = 0x00~0x2). Setting security flags failed
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function sets the security/safety related flags in the extra area (See 4.2 ”Flash Protection” on page 18 and 11.4 ”Extra Information Handling” on page 106).

Notes: 1. Boot Swap handling:

The self-programming library offers 2 functions to swap the boot blocks:

Temporary boot swap:
The function SelfLib_BootSwap() (See 4.3.4 ”Secure self-programming with bootloader update” on page 22) swaps the boot blocks immediately. As this function does not modify the boot swap flag in the device extra area, the swap effect lasts only until the next RESET. This function may be part of a secure reprogramming sequence with bootloader update (See 11.5 ”Secure Reprogramming Flow with bootloader Update” on page 107), but additionally the function SelfLib_SetInfo() is required for a permanent boot swap!

Permanent boot swap:
The function SelfLib_SetInfo() writes the boot swap flag in the Flash Extra Area. According to the flag, the boot blocks are or are not swapped from the next RESET onward (not immediately on the flag write!). The following sample code shows, how to invert the boot swap flag and program it into the extra area again in order to swap the boot blocks on the next RESET:

```
/* Load the security flags except boot swap flag */
secFlags = ( SelfLib_GetInfo_SecFlags() & ( ~0x00000001 ) );
/* Additionally load the boot swap flag. Based on the API definition it is
   automatically inverted */
secFlags |= ( SelfLib_GetInfo_BootSwap() & 0x00000001 );
/* The boot cluster size may not be set to 0xFF, which is the default value if not touched
   before. If so, we explicitly set it to a reasonable value */
if( ( secFlags >> 24 ) == 0xFF )
   secFlags &= 0x03FFFFFF;
/* Write the modified boot security flags */
ret = SelfLib_SetInfo( secFlags );
```

2. Once a protection flag has successfully been set either by this function or by the Flash programmer beforehand, it cannot be cleared again using this function. Only setting additional flags is possible
   The only possibility to clear a flag is to do a Chip Erase by the Flash programmer!
3. Setting the boot cluster protection flag reduces the flexibility in order to increase the Flash protection
   - Changing the boot swap flag is not longer allowed
   - Boot cluster size cannot be changed any longer.

13.4 Get Device Dependent Information

One central firmware function is used to read device dependent information. The corresponding SelfLib function is SelfLib_GetInfo. In order to simplify the information gathering for the user application, different function like macros have been defined in SelfLib.h, calling SelfLib_GetInfo with appropriate parameters and corresponding result casting. 13.4.1 ”Get information” on page 132 describes the basic function, while the other sub-chapters describe the function like macros.

13.4.1 Get information

Library call:

u32 SelfLib_GetInfo(u32 option)

Required parameters:

option
- 0: Get version information
- 1: Get implemented commands
- 2: Get CPU no. and block no.
- 3: Get protection flag information
- 4: Get boot area swap and Flash macro connection method
- 5: Get reset vector address
- 6~6+block no: Get last block address

Returned value:

	Option 0: xxxxxxxxxxxxxxxxAAAAAAAABBBBBBBB x: reserved A: firmware version (high) B: firmware version (low)
	Option 1: Bit0: Block Erase Bit1: Write Bit2: Block Internal Verify Bit3: Block Blank Check Bit4: Get flash information Bit5: Set flash information Bit6: Boot swap Bit7: FLMD0 check Bit8: Chip Erase Bit9: Extra area Read Bit10: Extra area Write Bit11: Get block unit Bit12: Dual Operation init Bit13: Pre-set flash information Bits14~31: reserved (0)
	Option 2: AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB A: CPU number B: Block count
	Option 3: AAAAAAAAxxxxxxxxxxxxxxxxxxSRWBCx A: Last boot cluster block number x: reserved S: Boot Cluster Write (1: enable, 0: disable) R: Read command (1: enable, 0: disable) W: Write command (1: enable, 0: disable) B: Block Erase command (1: enable, 0: disable) C: Chip Erase command (1: enable, 0: disable)
	Option 4: xxxxxxxxxxxxRODMBBBBBBBBBBBBBBBB x: reserved D: Data Flash (0: Supported, 1: Not supported) R: Variable Reset vector (0: Supported, 1: Not supported) O: Dual Operation (0: Supported, 1: Not supported) M: Block Size (0: 2kByte, 1: 4kByte) B: Boot Area Swap (0: not swapped, 1: swapped)
	Option 5: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA A: Variable reset vector address
	Option 6~6+Block no.: End address of the block, specified by block no.
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function returns device dependent information according to the option, passed to the function

13.4.2 Get protection flags

Library call:

u32 SelfLib_GetInfo_SecFlags(void)

Required parameters:

-

Returned value:

	Bit 0:	Reserved - Always 1
	Bit 1:	0: Chip Erase disabled 1: Chip Erase enabled
	Bit 2:	0: Block Erase disabled 1: Block Erase enabled
	Bit 3:	0: Write disabled 1: Write enabled
	Bit 4:	0: Read command disabled 1: Read command enabled
	Bit 5:	0: Boot Cluster is protected 1: Boot Cluster is not protected
	Bits 6~23:	Reserved for future use
	Bits 24~31:	Last boot block number (=Number of blocks in the boot cluster - 1)
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.	

Function description:

This function returns a 32 bit value containing above mentioned security related bits. The meaning of the bits is explained in 4.2.2 ”Protection configuration settings” on page 20.

13.4.3 Get block swapping flag

Library call:

u32 SelfLib_GetInfo_BootSwap(void)

Required parameters:

-

Returned value:

	0:	Boot swapping is not performed
	1:	Boot swapping is performed
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.	

Function description:

This function returns the value of the boot swap bit (See section 4.3 ”Security” on page 21).

13.4.4 Get device number

Library call:

u32 SelfLib_GetInfo_Device(void)

Required parameters:

-

Returned value:

	CPU number in decimal format e.g. uPD70F3166 (V850ES/SA3) = 3166 (=0x0C5E)
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function returns the CPU number, that is part of the device name, in decimal format

13.4.5 Get block count

Library call:

u32 SelfLib_GetInfo_BlockCnt(void)

Required parameters:

-

Returned value:

	Number of Flash blocks in the device
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function returns the number of Flash blocks in the device

13.4.6 Get block end address

Library call:

u32 SelfLib_GetInfo_BlockEndAdd(u32 blockNo)

Required parameters:

blockNo	Block number to be checked (Not block address, but number of the flash block)

Returned value:

	End address of the selected block
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function returns the end address of the block passed as a parameter.

13.4.7 Get variable reset vector address

Library call:

u32 SelfLib_GetInfo_ResetVectorAdd(void)

Required parameters:

-

Returned value:

	Address of the variable reset vector
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function returns variable reset vector address (See 4.5.1 ”Variable reset vector” on page 25).

13.5 Miscellaneous Functions

13.5.1 Boot swap

Library call:

u32 SelfLib_BootSwap(void)

Required parameters:

-

Returned value:

SELFLIB_OK	Boot swap executed successfully
SELFLIB_ERR_PROTECTION	Flash protection error
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function swaps the boot blocks and returns the new block number located at 0x00000000. Please refer to 5.4 ”Secure Reprogramming Flow with Bootloader Update” on page 30.

13.5.2 Check mode (FLMD0)

Library call:

u32 SelfLib_ModeCheck(void)

Required parameters:

-

Returned value:

SELFLIB_OK	V_{DD} is applied at the pin FLMD0
SELFLIB_ERR_FLMD0	Error, no V_{DD} on FLMD0 pin (See section 8.6 ”Reprogramming Scenario” on page 74
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

This function explicitly checks the FLMD0 pin and returns the current status.

13.6 Special functions

The functions mentioned in this chapter are not available on all devices. Please refer to the device documentation or to the *DeviceInfo* to check whether the selected device supports these.

13.6.1 Read

Library call:

u32 SelfLib_Read(	void *addSrc, void *addDest, u32 len)

Required parameters:

addSrc	Source address in the Flash
addDest	Destination address in RAM
len	number of words (4 Bytes) to read

Returned value:

SELFLIB_OK	The function has been executed successfully
SELFLIB_ERR_PARAMETER	Wrong parameters passed to the function, e.g. wrong source address or len
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

The function reads the Flash contents from the device to the destination buffer.

Note: For normal self-programming this function is not required as it is faster to read the data directly by normal memory accesses instead of using this function. This function has been implemented for special serial programmer related applications.

13.6.2 Get traceability data

Library call:

u32 SelfLib_GetTRCD(void *addDest)

Required parameters:

addDest	Destination address of the data read from the device

Returned value:

SELFLIB_OK	The function has been executed successfully
SELFLIB_ERR_UNSUPPORTED	The operation is not supported by the device
SELFLIB_ERR_FLOW	Only possible in case of user status check: Error in the flow of calling self-programming functions. A preceding Flash operation has not yet finished.

Function description:

The function reads traceability data words from the device to the destination buffer. The number of words is defined by the contents of the traceability data and so, device dependent. E.g. The 70F3441 has 3 data words.
Please refer to the device documentation regarding the data meaning and count.

13.7 User Status Check Related

The functions in this sub-chapter are only necessary and available, if the user status check is configured (See section 11.6 ”User Application Execution During Self-Programming” on page 110 and 12.4 ”Global SelfLib Defines - SelfLibSetup.h” on page 122).

13.7.1 Activate environment

Library call:

void SelfLib_FlashEnv_Activate(void)

Required parameters:

-

Returned value:

-

Function description:

This function activates the Flash environment. After activation no normal Flash read access (e.g. instruction fetch) is possible any more.

Note: During execution of this function no interrupts can be served. Although the execution time is relatively short (<50 µs), measures against interrupt overrun have to be taken.

13.7.2 Deactivate environment

Library call:

void SelfLib_FlashEnv_Deactivate(void)

Required parameters:

-

Returned value:

-

Function description:

This function deactivates the Flash environment. After deactivation normal Flash read access (e.g. instruction fetch) is possible again.

Note: During execution of this function no interrupts can be served. Although the execution time is relatively short (<50 µs), measures against interrupt overrun have to be taken.

13.7.3 Status check

Library call:

u32 SelfLib_StatusCheck (void)

Required parameters:

-

Returned value:

SELFLIB_BUSY	The initiated background operation is still ongoing
SELFLIB_OK	The initiated background operation (e.g. Erase) finished successfully
SELFLIB_ERR_FLASHPROCn	(n = 0x00~0x2). The initiated background operation (e.g. Erase) finished with an error

Function description:

This function checks the status of a previously initiated Flash background operation.

Note: This function need to be called frequently, as on some Flash background operations also device internal state machines need to be handled by the status check command in order to finish the operations.

13.8 Alternative Function Names

According to the different international requirements, definitions for alternative function names have been implemented. These names are reflected by defines in the *SelfLib* header file *SelfLib.h*

Table 13-1: Alternative function names

SelfLib function name	Alternative function name
SelfLib_BlankCheck	FlashBlockBlankCheck
SelfLib_Erase	FlashBlockErase
SelfLib_IVerify	FlashBlockIVerify
SelfLib_BootSwap	FlashBootSwap
SelfLib_Init	FlashInit
SelfLib_ModeCheck	FlashFLMDCheck
SelfLib_GetInfo	FlashGetInfo
SelfLib_GetTRCD	FlashGetTRCD
SelfLib_StatusCheck	FlashStatusCheck
SelfLib_SetInfo	FlashSetInfo
SelfLib_Write	FlashWordWrite
SelfLib_Read	FlashWordRead
SelfLib_LibVersion	FlashLibVersion
SelfLib_FctNewAddress	FlashFctNewAddress
SelfLib_FlashEnv_Activate	FlashEnv(1)
SelfLib_FlashEnv_Deactivate	FlashEnv(0)

[MEMO]

Appendix A Revision History

Item	Date published	Document No.	Comment
1	November 2003	U16929EE1V1AN00	Initial Release
2	June 2004	U16929EE2V0AN00	- Self-Programming chapters separated for UC2 Flash and SST Flash - Updated/Modified UC2 secure reprogramming flow - Extended/Reworked explanations and pictures in all chapters
3	August 2005	U16929EE3V0AN00	- Minor fixes in all chapters - Added SST(MF2) Flash support. Therefore, separated SST Flash programming chapters into SST(CZ6HSF/UX4) and SST(MF2)
4	February 2006	U16929EE3V1AN00	- Minor fixes in all chapters
5	May 2007	U16929EE3V2AN00	- Bug fixes in the API chapters - Some chapter reordering for better readability - Improved explanations - Added API functions for MF2 API - Extended the chapters on "Software considerations" - Added alternative function names explanations - Changed chapter naming for the different technologies

[MEMO]

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics America Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Marketing Services & Publishing
Fax: +49(0)-211-6503-1344

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: 02-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-6250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: +81- 44-435-9608

I would like to report the following error/make the following suggestion:

Document title: _____

Document number: _____ Page number: _____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐
Organization	☐	☐	☐	☐

[MEMO]