User Manual

32

# Data Flash Access Library

Type T01, European Release

RENESAS 32-Bit MCU
RH Family / RH850 Series

Installer:
RENESAS_FDL_RH850_T01E_V1.xx



Renesas Electronics
www.renesas.com

R01US0079ED0103
19.02.2014

# Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas 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 Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics.

8. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   **“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.

   **“High Quality”:** 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; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.

9. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
10. Although Renesas Electronics endeavours to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
13. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

    **Note 1** “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority- owned subsidiaries.

    **Note 2** “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

# Regional information

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

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

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

Visit

http://www.renesas.com

to get in contact with your regional representatives and distributors.

# Preface

**Readers** This manual is intended for users who want to understand the functions of the concerned libraries.

**Purpose** This manual presents the software manual for the concerned libraries.

**Note** Additional remark or tip

**Caution** Item deserving extra attention

**Numeric notation**

| | |
|---|---|
| Binary: | xxxx or xxxB |
| Decimal: | xxxx |
| Hexadecimal | xxxxH or 0x xxxx |

**Numeric prefix** Representing powers of 2 (address space, memory capacity):

| | |
|---|---|
| K (kilo) | 210 = 1024 |
| M (mega): | 220 = $1024^2$ = 1,048,576 |
| G (giga): | 230 = $1024^3$ = 1,073,741,824 |

**Register** X, x = don’t care

**Diagrams** Block diagrams do not necessarily show the exact software flow but the functional structure. Timing diagrams are for functional explanation purposes only, without any relevance to the real hardware implementation.

# How to Use This Document

### (1) Purpose and Target Readers

This manual is designed to provide the user with an understanding of the functions and characteristics of the Self-Programming Library. It is intended for users designing application systems incorporating the library. A basic knowledge of embedded systems is necessary in order to use this manual. The manual comprises an overview of the library, API description, usage notes and cautions.

Particular attention should be paid to the precautionary notes when using the manual. These notes occur within the body of the text, at the end of each section, and in the Cautions section.

The revision history summarizes the locations of revisions and additions. It does not list all revisions. Refer to the text of the manual for details.

### (2) List of Abbreviations and Acronyms

| Abbreviation | Full form |
|---|---|
| Bootloader | A piece of software located in the Boot Cluster handling the reprogramming of the device |
| Code Flash | Embedded Flash where the application code or constant data is stored. |
| Data Flash | Embedded Flash where mainly the data of the EEPROM emulation are stored. |
| Dual Operation | Dual operation is the capability to access flash memory during reprogramming of another flash memory range.<br>Dual operation is available between Code Flash and Data Flash.<br>Between different Code Flash macros dual operation depends on the device implementation. |
| ECC | Error Correction Code |
| EEL | EEPROM Emulation Library |
| EEPROM | Electrically erasable programmable read-only memory |
| EEPROM emulation | In distinction to a real EEPROM the EEPROM emulation uses some portion of the flash memory to emulate the EEPROM behavior. To gain a similar behavior some side parameters have to be taken in account. |
| FCL | Code Flash Library (Code Flash access layer) |
| FDL | Data Flash Library (Data Flash access layer) |
| FHVE | Software protection of flash memory against programming and erasure. Not present in all devices. |
| Firmware | Firmware is a piece of software that is located in a hidden area of the device, handling the interfacing to the flash. |
| Flash | Electrically erasable and programmable non-volatile 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. |

| Abbreviation | Full form |
|---|---|
| Flash Macro | A certain number of Flash blocks are grouped together in a Flash macro. |
| ICU | Intelligent Cryptographic Unit |
| Power Save Mode | Device modes to consume less power than during normal operation. In the device documentation also called “stand-by modes” |
| RAM | “Random access memory” - volatile memory with random access |
| REE | Renesas Electronics Europe GmbH |
| REL | Renesas Electronics Japan |
| ROM | “Read only memory” - non-volatile memory. The content of that memory cannot be changed. |

# Table of Contents

# Chapter 1 Introduction

This user manual describes the internal structure, the functionality and the application programming interface (API) of the Renesas RH850 Data Flash Access Library (FDL) Type 01, designed for RH850 flash devices based on a common flash technology.

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 Renesas RH850 Data Flash Access Library Type 01 (from here on referred to as FDL) is provided for the Green Hills, IAR and Renesas compiler environments. The library and application programs are distributed using an installer tool allowing selecting the appropriate environment.

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

The Data Flash Access library, the latest version of this user manual and other device dependent information can be downloaded from the following URL:

http://www.renesas.eu/update

Please ensure to always use the latest release of the library in order to take advantage of improvements and bug fixes.

This manual is based on the assumption that the device will operate in supervisor mode. For information on other modes, refer to the user's manual for the hardware.

**Note:**

Please read all chapters of this user manual carefully. Much attention has been put to proper description of usage conditions and limitations. Anyhow, it can never be completely ensured that all incorrect ways of integrating the library into the user application are explicitly forbidden. So, please follow the given sequences and recommendations in this document exactly in order to make full use of the library functionality and features and in order to avoid malfunctions caused by library misuse.

### Flash Infrastructure

Besides the Code Flash, many devices of the RH850 microcontroller family are equipped with a separate flash area — the Data Flash. This flash area is meant to be used exclusively for data. It cannot be used for instruction execution (code fetching).

### Flash Granularity

The Data Flash of RH850 device is separated into blocks of 64 byte. While erase operations can only be performed on complete blocks, data writing can be done on a granularity of one word (4 bytes). Reading from an erased flash word will return random data. The number of available Data Flash blocks varies between the different RH850 devices. Please refer to the corresponding user's manual for the hardware for detailed information.

Data Flash is divided into Flash macro

# Chapter 2 Architecture

This chapter introduces the basic software architecture of the FDL and provides the necessary background for application designers to understand and effectively use the library. Please read this chapter carefully before moving on to the details of the API description.

## 2.1 Layered Architecture

This chapter describes the function of all blocks belonging to the EEPROM Emulation System (EES). Even though this manual describes the functional block FDL, a short description of all concerned functional blocks and their relationship can be beneficial for the general understanding.

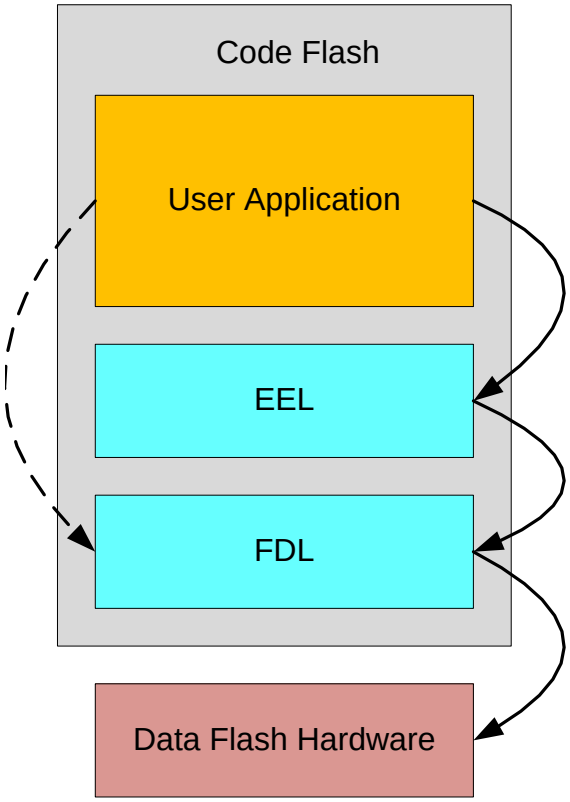



**Figure 1: Symbolic relationship between the EES functional blocks**

As depicted in the figure above, the software architecture of the EEPROM Emulation System is built up of several blocks:

- **User Application:** This functional block will use functions offered by the FDL and EEL. The user shall take care that FDL functions are not used at the same time while EEL is operating. Please refer to EEL documentation for how to stop EEL execution in order to safely access FDL functions.
- **EEPROM Emulation Library (EEL):** This functional block offers all functions and commands that the “User Application” block can use in order to handle its EEPROM data.
- **Data Flash Access Library (FDL):** The Data Flash Access Library is the subject of this manual. It should offer an access interface to any user-defined flash area, the so called “FDL-pool” (described in next chapter). Beside the initialization function, the FDL allows the execution of access-commands like write/blank check as well as suspend-able erase command.

- **Data Flash Hardware:** This functional block represents the Flash Programming Hardware controlled by the FDL.

## 2.2 Pool Definitions

The *FDL pool* defines the Flash blocks, the user application and a potential EEPROM Emulation (EEL) may use for FDL Flash access. The limits of the *FDL pool* are taken into consideration by any of the FDL Flash access commands. The user can define the size of the *FDL pool* freely at project run-time during library initialization.

The *FDL pool* provides the space for the *EEL Pool* which is allocated by the EEL inside the *FDL pool*. The *EEL Pool* provides the Flash space for the EEL to store the emulation data and management information.

All *FDL pool* space not allocated by the *EEL Pool* is freely usable by the user application, so is called the *User Pool*.

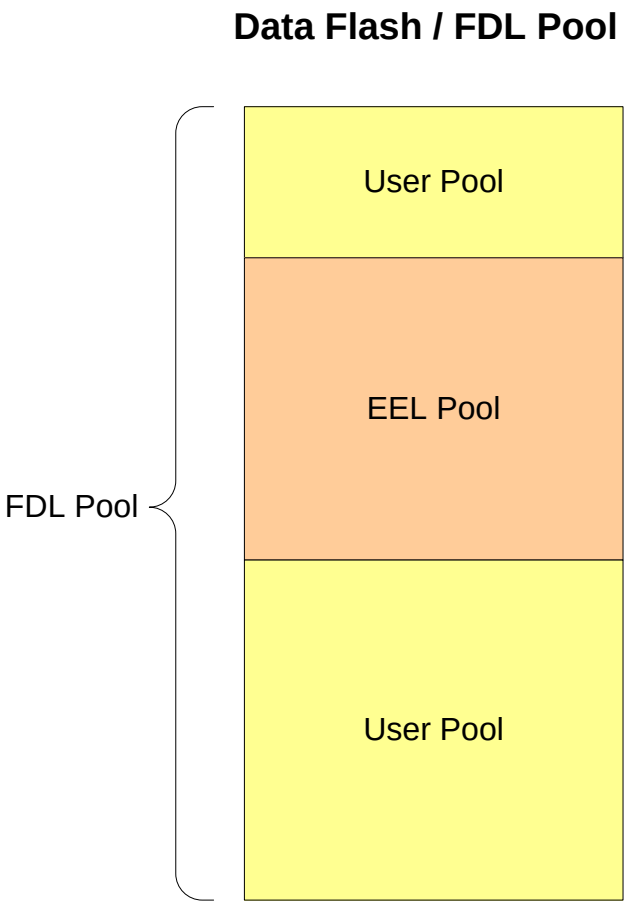



**Figure 2: Pools overview**

## 2.3 Architecture related notes

- All Data Flash related operations are executed by the FDL. Thus, the application cannot access (Erase, Write ...) the Data Flash directly. An exception is Reading the Flash contents. As the Flash is mapped to the CPU address space, it can be directly read by the CPU. The FDL provides an additional read operation that will take care of possible ECCs (error correction code) errors to allow error polling.
- The FDL allows accessing the Data Flash only.

- Parallel Flash operations (Except Read by the CPU) on Data Flash and Code Flash are not possible due to shared resources between the Flash macros.

# Chapter 3 Functional Specifications

## 3.1 Supported functions, commands and Flash operations

For a better understanding of the flows and mechanisms required for an FDL usage, the basic functions of the FDL are introduced in the following. The API of the FDL is thereby, on the one hand based on functions used to manage the operation of the library itself, on the other hand it offers so-called commands to access and control the content of the FDL pool.

The following table lists up all functions, the library will support. Please refer to the chapter 4.4 “Functions” for detailed descriptions.

**Table 1: FDL Function**

| Function | Description |
|---|---|
| R_FDL_Init | Initialize the library and Flash hardware |
| R_FDL_Execute | Initiate a Flash operation |
| R_FDL_Handler | Control an initiated Flash operation and forward the status. |
| R_FDL_SuspendRequest | Request suspending an on-going Flash operation |
| R_FDL_ResumeRequest | Resume a suspended Flash operation |
| R_FDL_StandBy | Suspend an on-going Flash operation from an asynchronous context |
| R_FDL_WakeUp | Wake-up the FDL from Stand-by state |
| R_FDL_GetVersionString | Return a pointer to the library version string |

Commands are used to manage the FDL pool. Commands are initiated via `R_FDL_Execute` and the further progress is controlled by regular execution of `R_FDL_Handler`.
The following commands can be used to execute the following Flash operations:

**Table 2: FDL commands and operations**

| Command | Initiated Flash operation | Description |
|---|---|---|
| R_FDL_CMD_ERASE | Flash erase | Erase one or more Flash blocks |
| R_FDL_CMD_WRITE | Flash write | Write one or more Flash words |
| R_FDL_CMD_BLANKCHECK | Flash blank check | Blank Check one or more Flash words. Return the fail address in case some Flash word is not blank |
| R_FDL_CMD_READ | Flash data read | Read one or more Flash words to a buffer. Return a possible ECC (Error correction code) error to the application together with the address of the error |

The following picture shows the basic flow of Flash operations at the example of erasing 2 Flash blocks. While the Flash hardware can only Erase or Write one unit (Erase 1 block, Write 1 word), the FDL will manage handling multiple units. Blank Check is executed on word basis but internally it is split in multiple units at each multiple of 0x4000 bytes boundary.

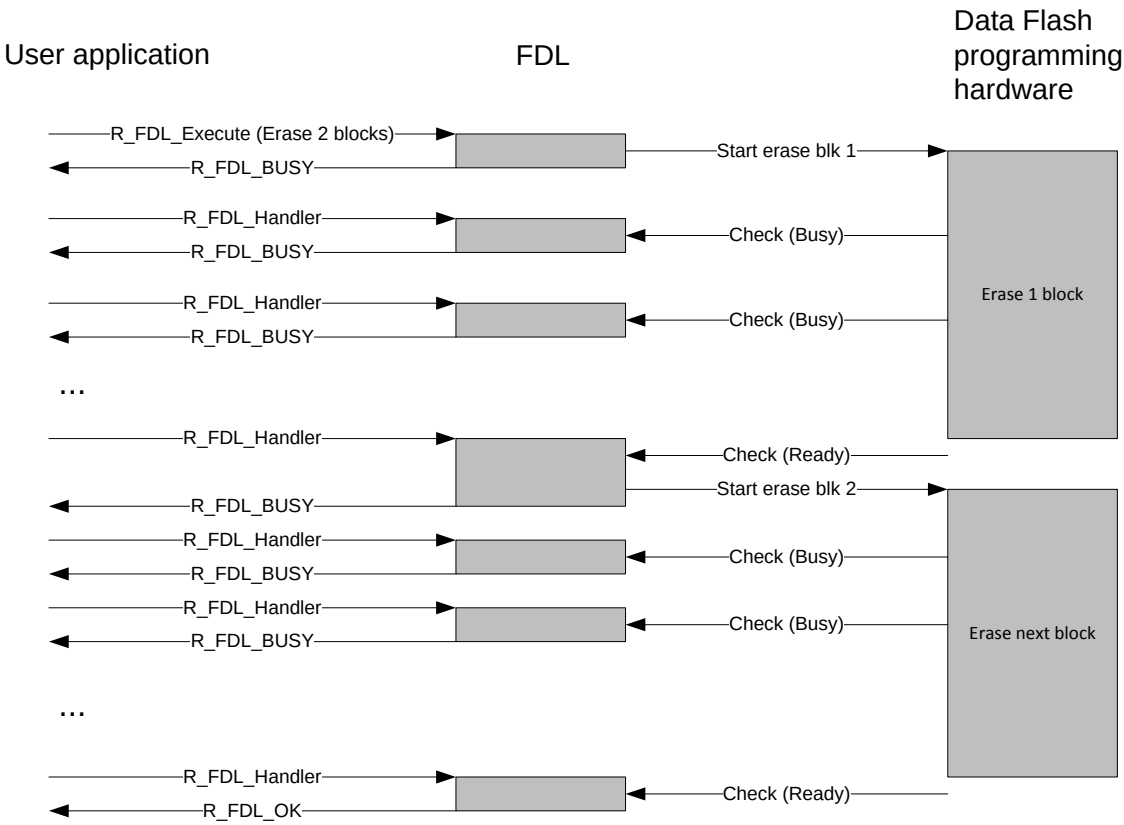



Figure 3: Flash erase sequence

## 3.2 Request-response oriented dialog

The FDL utilizes request-response architecture in order to initiate the commands. This means any "requester" (any tasks in the user application) has to fill a request structure and pass it by reference to the Data Flash Access Library using `R_FDL_Execute` function. The FDL interprets the content of the request variable, checks its plausibility and initiates the execution. The feedback is reflected immediately to the requester via the status member (`status_enu`) of the same request structure. The completion of an accepted request/command is done by calling `R_FDL_Handler` periodically as long as the request remains "busy".

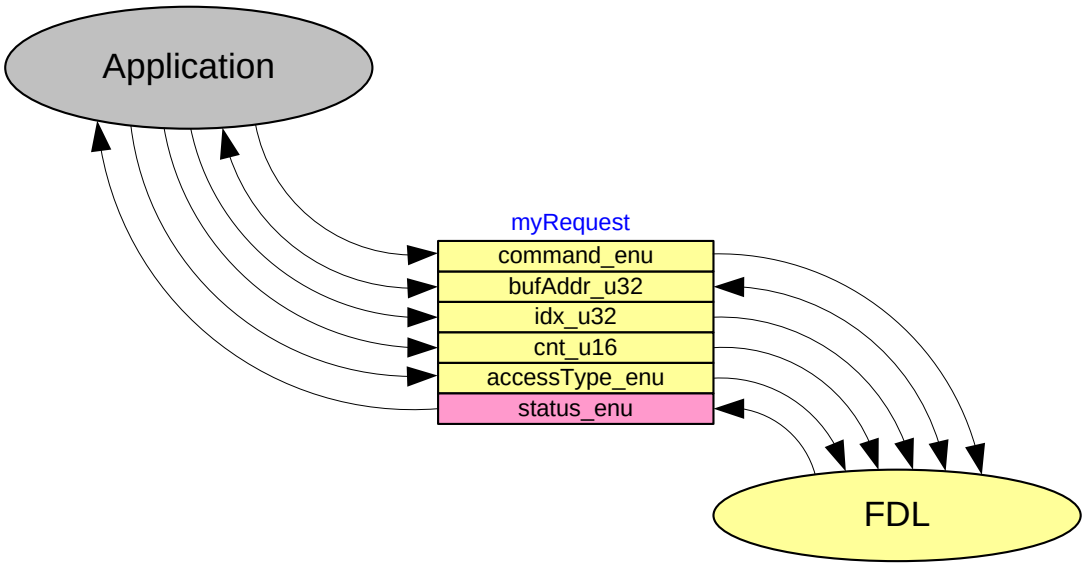



**Figure 4: Usage of the request structure**

Details on the request variable structure and its members are given later in section 4.3.3 “r_fdl_request_t”. Please also note that not all structure members are required for all commands. The individual command descriptions in section 4.5 “Commands” provide the corresponding detailed information.

## 3.3 Background Operation

The flash technology provided by Renesas enables the application to write/erase the Data Flash in parallel to the CPU execution. In order to satisfy the operation in concurrent or distributed systems, the command execution is divided into two steps:

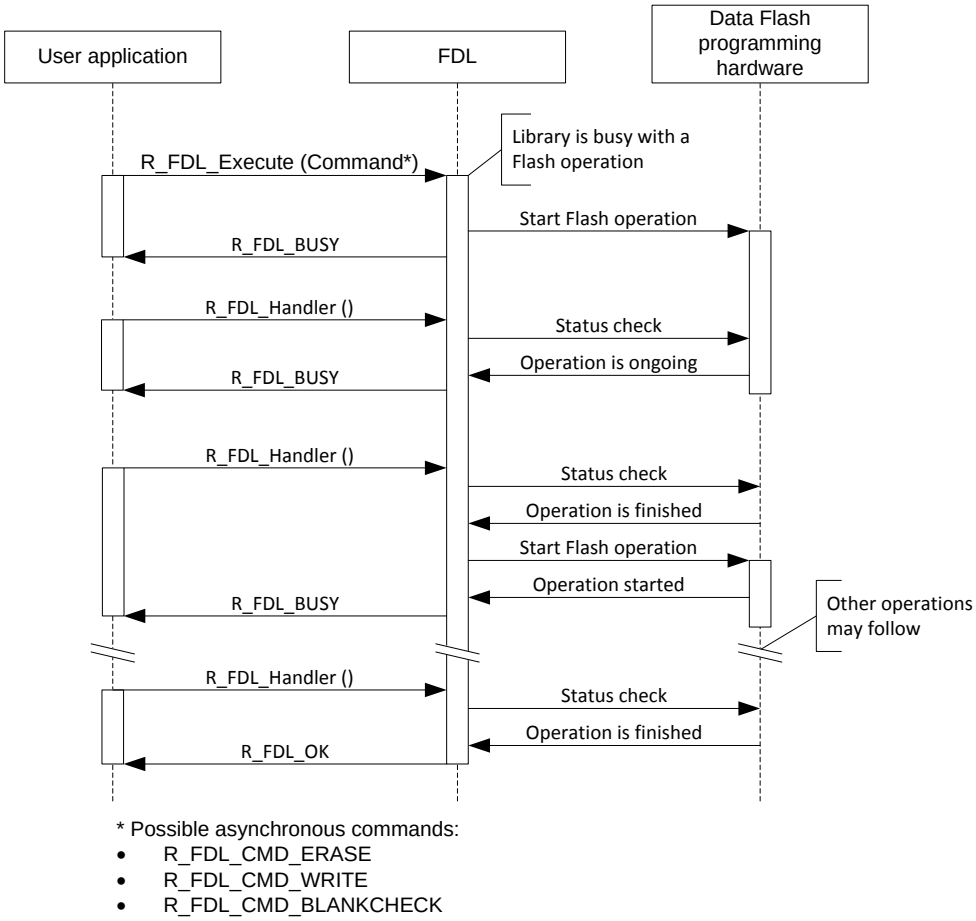
1. Initiation of the command execution using `R_FDL_Execute`
2. Processing of the requested command state by using `R_FDL_Handler`

This approach comes with one important advantage: Command processing can be done centrally at one place in the target system (normally the idle-loop or the scheduler loop), while the status of the requests can be polled locally within the requesting tasks.

Please note that `R_FDL_Execute` only initiates the command execution and returns immediately with the request-status "busy" after execution of the first internal state (or an error in case the request cannot be accepted).

The device flash hardware is responsible for executing the operation in the background. The device hardware operation might be divided into multiple operations, each performed on a separate occasion, depending on the number of blocks and data items. The first operation is conducted by calling the `R_FDL_Execute` function. The second and subsequent operations are triggered by calling the `R_FDL_Handler` function. Thus, there is a need to call the `R_FDL_Handler` function multiple times. Processing is suspended from the time each separate operation is completed until the next one is triggered. Therefore, as the time interval between `R_FDL_Handler` functions call increases, so does the overall processing time.

An exception to this background operation is `R_FDL_CMD_READ` command that is executed synchronously during `R_FDL_Execute` function.




* Possible asynchronous commands:

- R_FDL_CMD_ERASE
- R_FDL_CMD_WRITE
- R_FDL_CMD_BLANKCHECK

**Figure 5: Background operation**

## 3.4 Flash Access Protection

The FDL Flash Access Protection shall protect Flash accesses to unintended addresses. The protection distinguishes EEL-Pool Flash blocks from User-Pool blocks (refer to chapter 2.2 “Pool Definitions” for more information). An access as user application will be allowed to all configured Flash blocks outside the EEL-Pool, while an access from EEL will be allowed to the EEL-Pool only.

Generally, on any Data Flash operation initiation, the access type must be defined in the operation request structure variable. Setting this variable enables the access either to the EEL-Pool or to the Data Flash blocks outside the EEL-Pool (User-Pool). If the variable is not initialized appropriately or if the wrong pool shall be accessed, a protection error is returned.

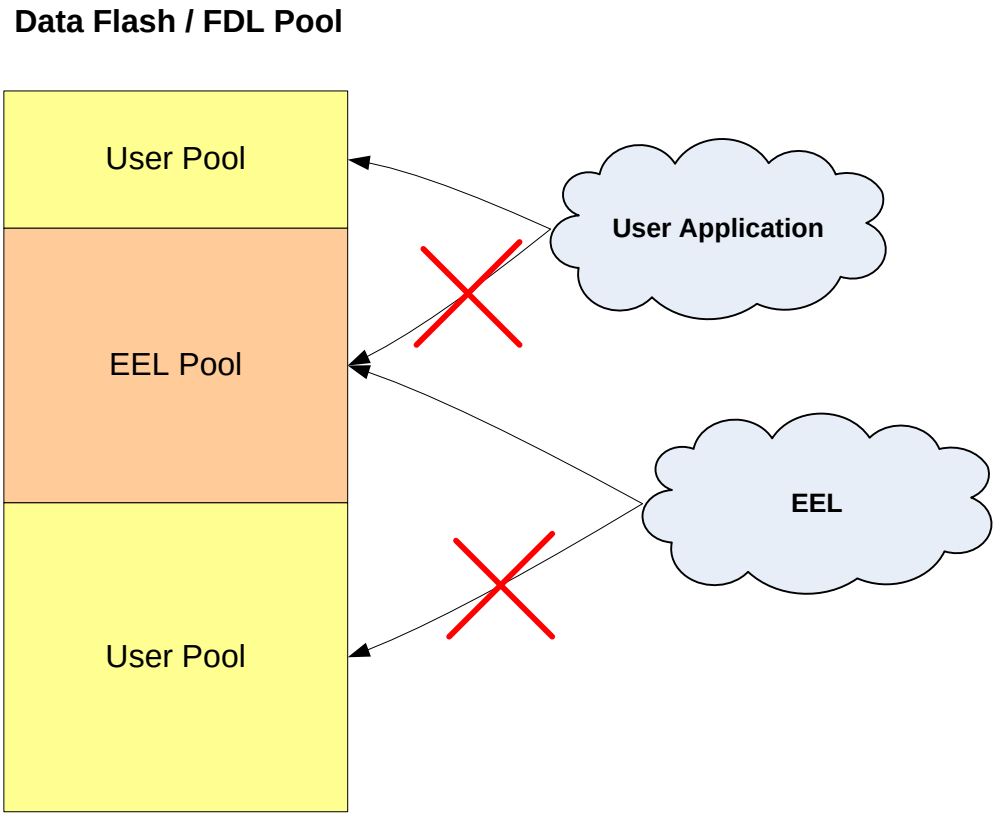



Figure 6: Flash Access Rights

## 3.5 Suspend / Resume mechanism

Some Data Flash operations can last a long time especially multiple erase and write. The user application cannot always wait for the operation end because other operations have higher priority. So, from user point of view current operation is suspend-able and can be resumed after finishing the other Flash accesses.

From software point of view an on-going operation always ends in suspended state unless the resume is requested beforehand. In case the Flash hardware has already finished an operation but its end result has not already been processed by the library, the library returns the suspended status. The final operation result is returned after successful resume request.

The FDL contains special functions to suspend and resume an ongoing operation. Please refer to chapter 4.4.3.1 “R_FDL_SuspendRequest”.

Examples of Erase or Write Suspend-Resume flow:

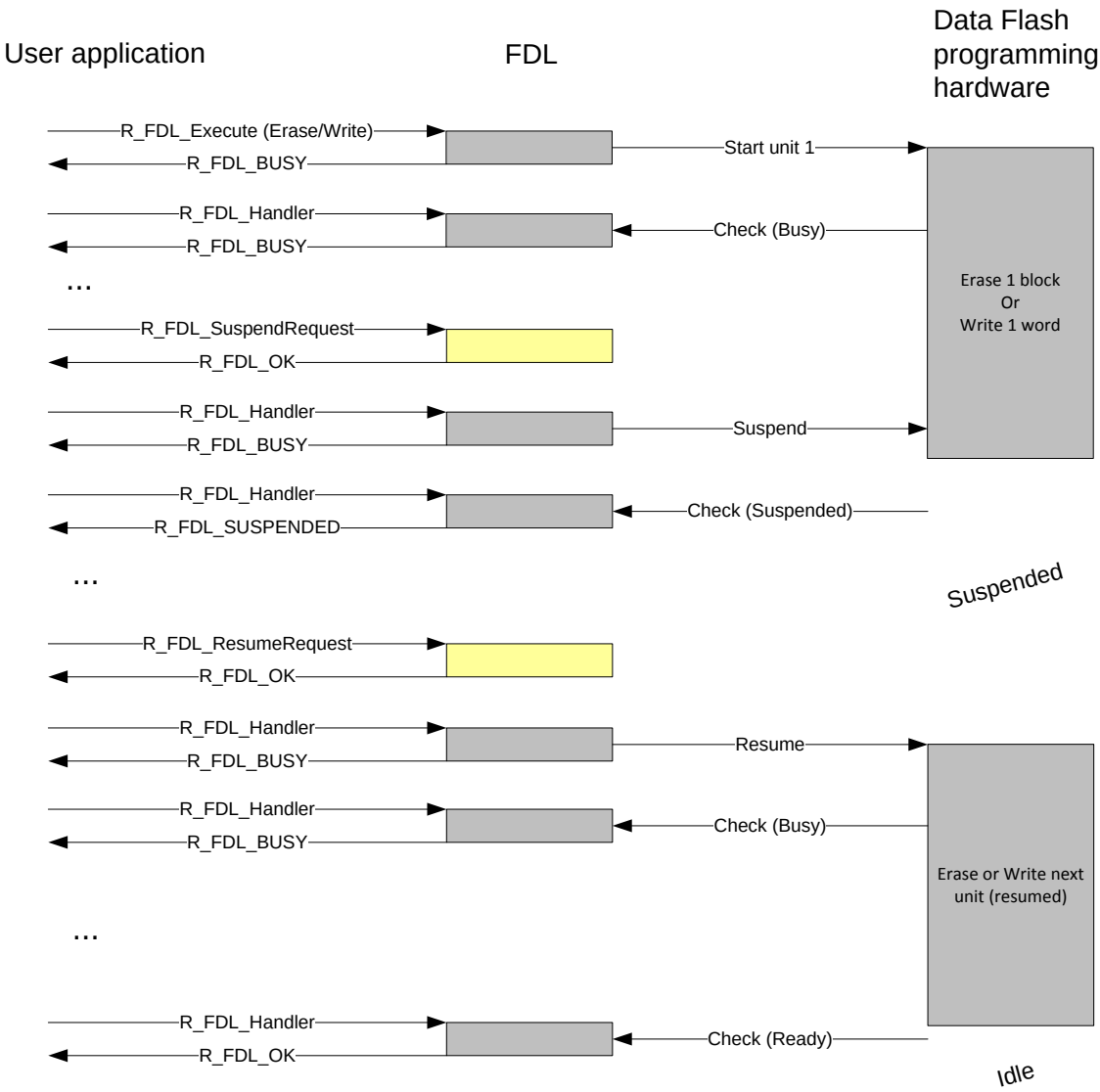



**Figure 7: Erase/Write Suspend Resume Flow**

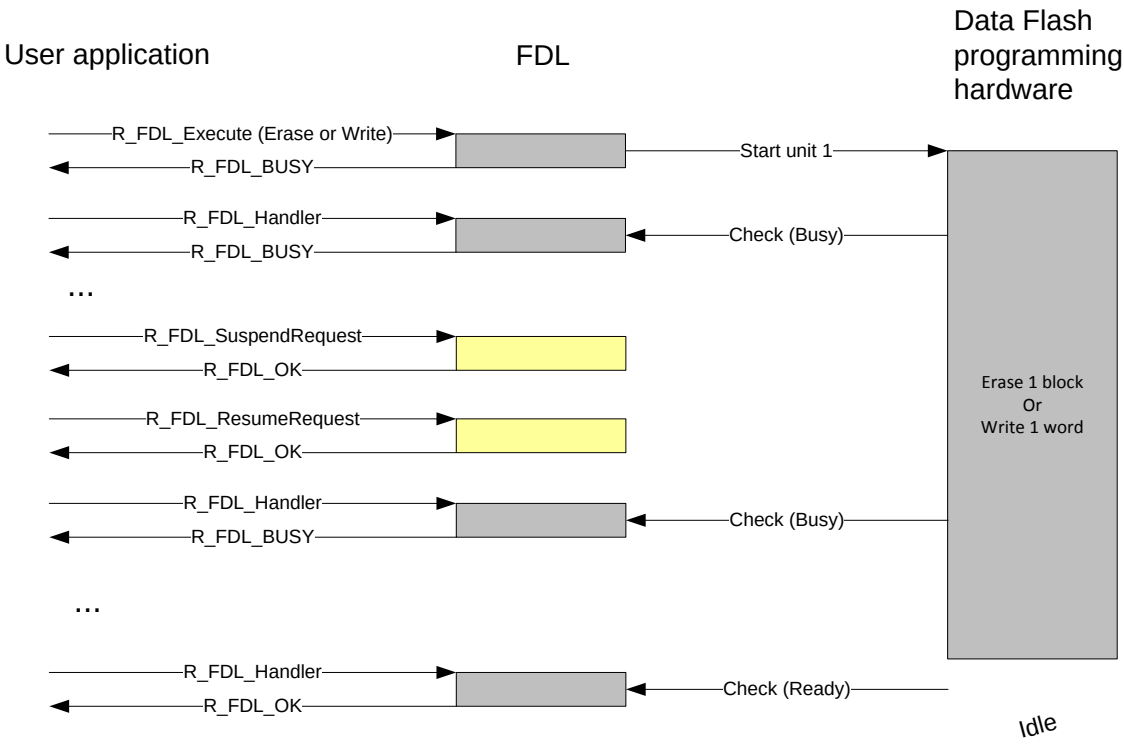



**Figure 8: Erase/Write Suspend + Immediate Resume**

Blankcheck operation will not be interrupted by a suspend request unless the operation reaches a Flash Macro boundary (any multiple of 0x4000 bytes) or it will be finished:

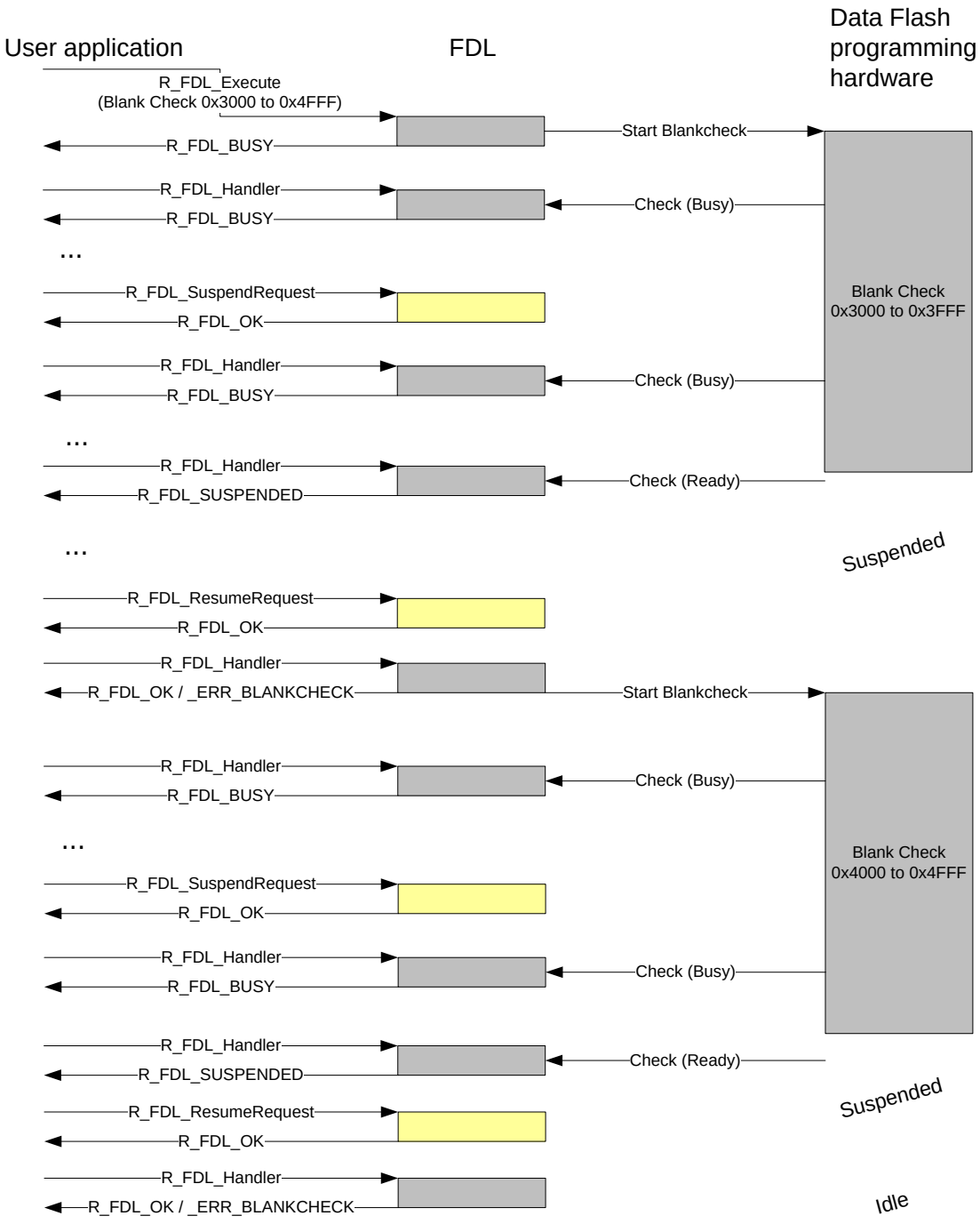



**Figure 9: Suspend/Resume a Blankcheck operation**

**Notes:**

When Erase processing is suspended and resumed, this is not considered as an additional erase with respect to the specified Flash erase endurance.

The suspend / resume mechanism cannot be nested. Therefore, the following sequence is not allowed: Erase Flash ► Suspend ► Start another erase flash ► Suspend.

## 3.6 Stand-by and Wake-up functionality

Entering a device power save (stand-by) mode is not allowed, when a Data Flash operation is on-going. Due to that, especially Data Flash Erase operation can delay entering a power save mode significantly. In order to allow fast entering of such mode, the functions `R_FDL_StandBy` and `R_FDL_WakeUp` have been introduced. The main functionality of the functions is to suspend a possibly on-going Data Flash Erase or Write operation (`R_FDL_StandBy`) and resume it after waking up from power save mode (`R_FDL_WakeUp`).

Once started, stand-by processing must always end in stand-by status. Calling the `R_FDL_StandBy` does not necessarily immediately suspend any Data Flash operation, as suspend might be delayed by the device internal hardware or might not be supported at all (only Erase and Write are suspend-able). In this case, the `R_FDL_StandBy` function must be called repeatedly until the stand-by status is reached.

Blank Check and Read Data Flash operations are suspendable from software point of view, but the library will wait for the operation to be finished by hardware while suspend is processed and the result will be presented after resuming. This wait however is not that important because blank check and read operations are much faster than erase or write.

In case the FDL is in an idle state (no on-going Data Flash operations), by calling the `R_FDL_StandBy` the FDL will immediately enter the Stand-By state. By calling the `R_FDL_WakeUp`, the FDL will return to previous state (in this case the idle state).

The following pictures describe the library behaviour in case a stand-by request is issued during FDL operation:

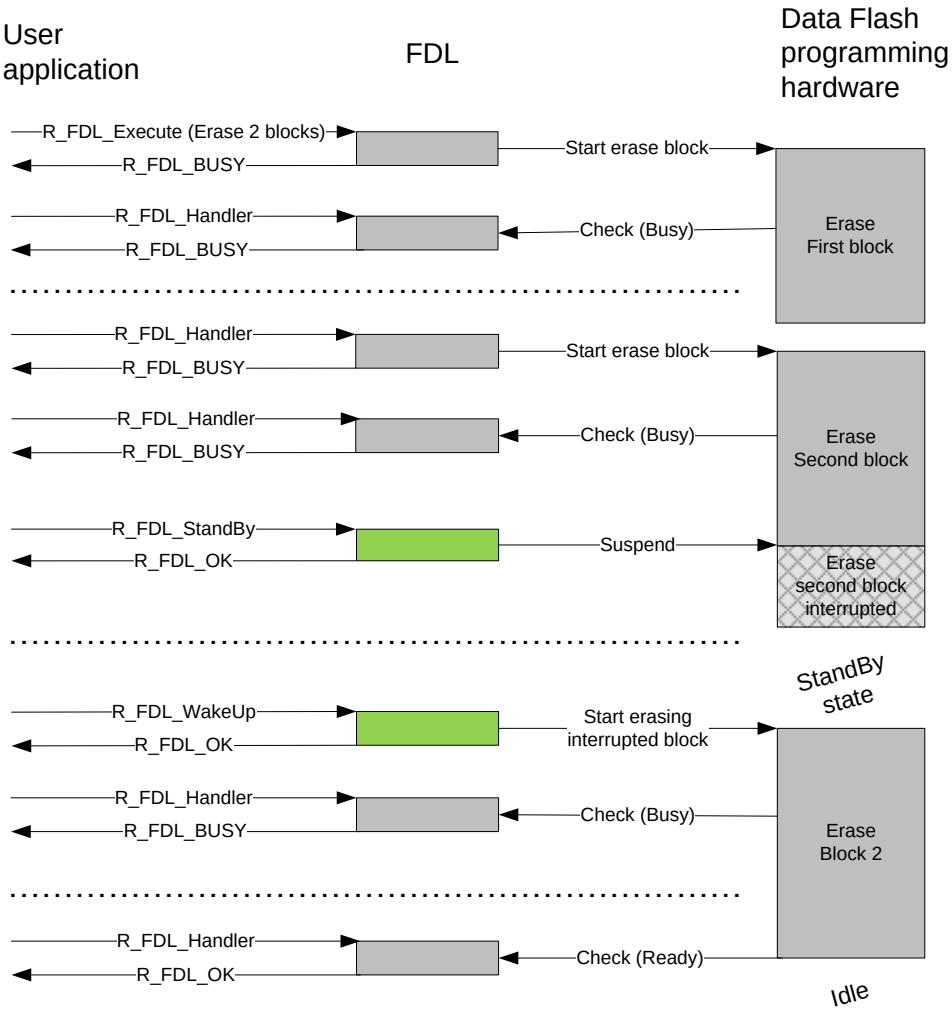



**Figure 10: Stand-By processing on a Data Flash Erase operation**

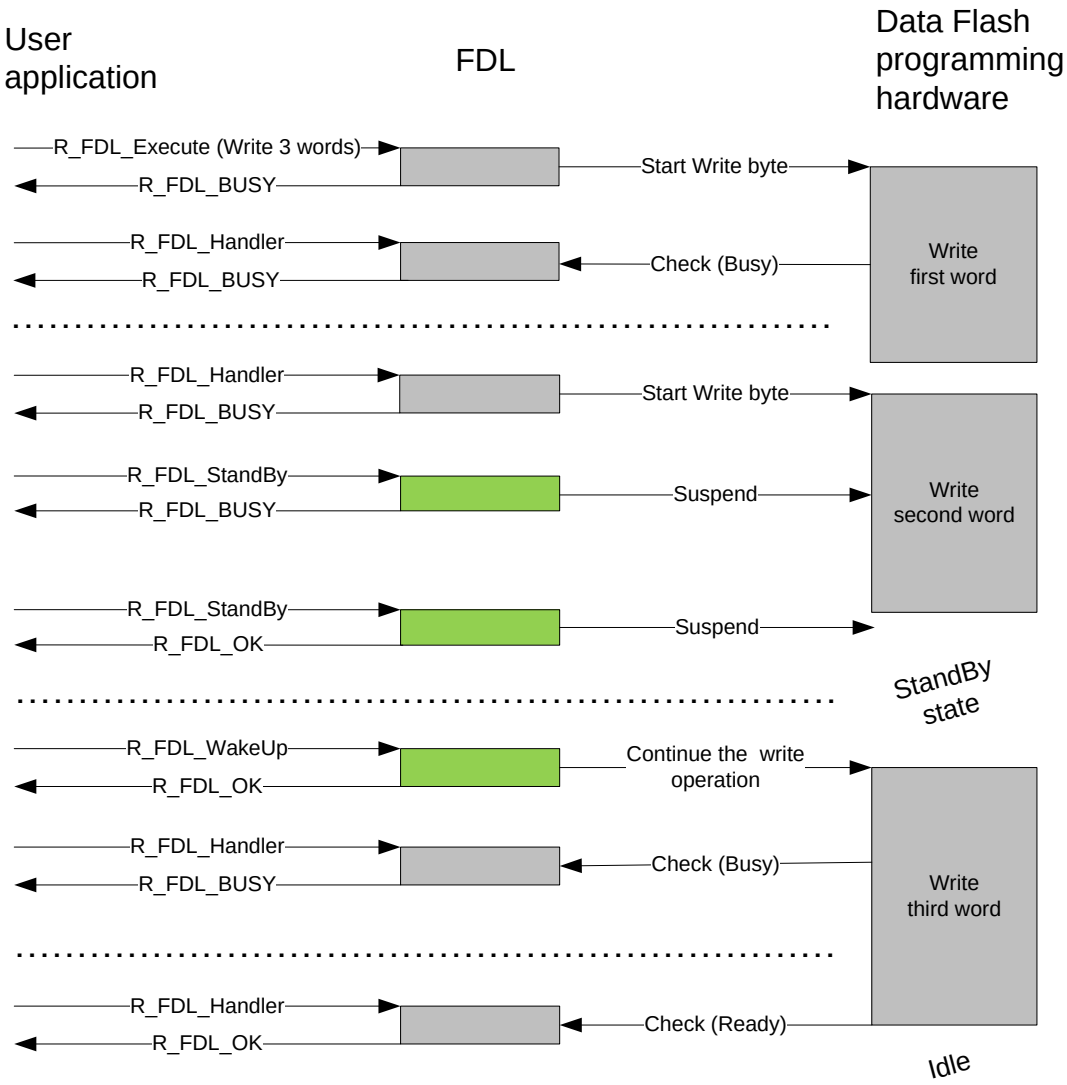



**Figure 11: Stand-By processing on a Data Flash Write operation**

# Chapter 4 User Interface (API)

This chapter provides the formal description of the application programming interface of the Flash Data Library Type T01 (FDL). It is strongly advised to read and understand the previous chapters presenting the concepts and structures of the library before continuing with the API details.

## 4.1 Pre-compile configuration

The pre-compile configuration has a direct impact on the object file generated by the compiler. Hence it is used for conditional compilation (e.g. solve device dependencies of the code).

The configuration is done in the module `fdl_cfg.h`. The user has to configure all parameters and attributes by adapting the related constant definitions in that header-file.

The following configuration options are available:

1. Critical section

   One configuration element is the critical section handling of the library. The function `R_FDL_Init` needs to activate the device internal special memory for a short time in order to have access to certain data. This results in disabling the Code Flash. During that time, code from Code Flash cannot be executed as well as data cannot be read. The library provides the possibility to execute call-back routines in order for the user to handle the implications of disabling the Code flash (for the impact on the application, please refer to Chapter 6 Cautions). The call-back routines are executed at the begin and end of the critical section. The defines to set the call back routines are described in the following:

   `FDL_CRITICAL_SECTION_BEGIN`: Possibility to execute a call back routine at critical section start (e.g. disable interrupts and exceptions)

   `FDL_CRITICAL_SECTION_END`: Possibility to execute a call back routine at critical section end (e.g. enable interrupts and exceptions)

   Implementation in the sample application:

```
#define FDL_CRITICAL_SECTION_BEGIN FDL_User_CriticalSetionBegin();
#define FDL_CRITICAL_SECTION_END   FDL_User_CriticalSetionEnd();
```

2. Device family

   The macro `FDL_CFG_E1X_P1X_PLATFORM` must be defined for E1x and P1x and must be left undefined for F1x and R1x devices.

## 4.2 Run-time configuration

The FDL configuration can be changed dynamically at runtime. It contains important FDL related information (e.g. CPU frequency, number of blocks used by library, authentication code) and EEL information (e.g. EEL pool size and EEL starting block number).

The run-time configuration is stored in a descriptor structure (see `r_fdl_descriptor_t`), which is declared in `r_fdl_types.h`, but defined in the user application and passed to the library by the function `R_FDL_Init`.

The file `fdl_descriptor.c` shall show an example of the descriptor structure definition and filling, while the `fdl_descriptor.h` shall show an example of the definitions required to fill in the structure.

In fact, the file `fdl_descriptor.h` might be modified according to the user applications needs and might be added to the user application project together with the `fdl_descriptor.c`. The descriptor files (.c and .h) are part of the library installation package.

The following settings should be configured by user:

1. **AUTHENTICATION_ID:** A 16 byte access ID code.

   The ID code is used to secure access to:

   1. On Chip Debug unit
   2. Serial programming
   3. Self-programming only on older F1x devices. Newer devices will simply ignore a wrong ID code so self-programming feature is always available.

      On older devices where ID code is still checked, a wrong code will not result in a failed initialization but FDL commands will fail to operate with a `R_FDL_ERR_PROTECTION` error status.

2. **CPU_FREQUENCY_MHZ:** This defines the internal CPU frequency in MHz unit, rounded up to the nearest integer, e.g. for 24.3 MHz set `CPU_FREQUENCY_MHZ` to 25. Please check the Device Manual for limit values
3. **FDL_POOL_SIZE:** It defines the number of blocks to be accessed by the FDL for user access and EEL access. Usually it is set to the total number of blocks physically available on the device. For example, if the device is equipped with 32 KB of Data Flash and the block size is 64 bytes, then FDL_POOL_SIZE can be any value up to 512
4. **EEL_POOL_START:** It defines the starting block of the EEL-Pool. If FDL is used without EEL on top, the value should be set to 0
5. **EEL_POOL_SIZE:** It defines the number of blocks used for the EEL-Pool. If FDL is used without EEL on top, the value should be set to 0

FDL block size is always equal to the physical block size of Data Flash.

Example of descriptor when FDL is used alone:

```
/* default access code */
#define AUTHENTICATION_ID       { 0xFFFFFFFF, \
                                  0xFFFFFFFF, \
                                  0xFFFFFFFF, \
                                  0xFFFFFFFF }
#define CPU_FREQUENCY_MHZ       (80)
/* FDL pool will use 512 blocks * 64 bytes = 32KB, no EEL pool */
#define FDL_POOL_SIZE           (512)
#define EEL_POOL_START          (0)
#define EEL_POOL_SIZE           (0)
```

Example of descriptor when EEL is used:

```
/* default access code */
#define AUTHENTICATION_ID       { 0xFFFFFFFF, \
                                  0xFFFFFFFF, \
                                  0xFFFFFFFF, \
                                  0xFFFFFFFF }
#define CPU_FREQUENCY_MHZ       (80)
/* FDL pool will use 32KB, EEL pool occupies fist 16 KB */
#define FDL_POOL_SIZE           (512)
#define EEL_POOL_START          (0)
#define EEL_POOL_SIZE           (256)
```

EEL may be configured to use virtual blocks. A virtual block size is an integer multiple of physical block size and it is aligned to physical blocks. Please consult EEL documentation for details.

Example of descriptor when EEL is used with virtual block size 32 times the size of physical block size:

```
/* default access code */
#define AUTHENTICATION_ID       { 0xFFFFFFFF, \
                                  0xFFFFFFFF, \
                                  0xFFFFFFFF, \
                                  0xFFFFFFFF }
#define CPU_FREQUENCY_MHZ       (80)
#define FDL_POOL_SIZE           (512)
/* FDL pool will use 32KB, from wich EEL pool occupies area:
START:       1 * 32 * 64 = 2048 till
END:         6 * 32 * 64 + 2048 = 14335 */
#define EEL_VIRTUALBLOCKSIZE    (32u)
#define EEL_POOL_START          (1u * EEL_VIRTUALBLOCKSIZE)
#define EEL_POOL_SIZE           (6u * EEL_VIRTUALBLOCKSIZE)
```

## 4.3 Data types

This section describes all data definitions used and offered by the FDL. In order to reduce the probability of type mismatches in the user application, please make strict usage of the provided types.

### 4.3.1 Library specific simple type definitions

**Type definition:**

```
typedef signed char         int8_t;
typedef unsigned char       uint8_t;
typedef signed short        int16_t;
typedef unsigned short      uint16_t;
typedef signed long         int32_t;
typedef unsigned long       uint32_t;
typedef unsigned char       rBool;
```

**Description:** These simple types are used throughout the complete library API. All library specific simple type definitions can be found in file `r_typedefs.h`, which is part of the library installation package.

### 4.3.2 r_fdl_descriptor_t

**Type definition:**

```
typedef struct R_FDL_DESCRIPTOR_T
{
    uint32_t    id_au32[4];
    uint16_t    cpuFrequencyMHz_u16;
    uint16_t    fdlPoolSize_u16;
    uint16_t    eelPoolStart_u16;
    uint16_t    eelPoolSize_u16;
} r_fdl_descriptor_t;
```

**Description:** This type is the run-time configuration (see chapter 4.2 “Run-time configuration”). A variable of this type is read during initialization phase then hardware and internal variables are set according to the configuration.

**Member / Value:**

| Member / Value | Description |
|---|---|
| id_au32[4] | Authentication ID array code |
| cpuFrequencyMHz_u16 | CPU frequency in MHz |
| fdlPoolSize_u16 | FDL pool size in number of blocks |
| eelPoolStart_u16 | Number of first block of the EEL pool |
| eelPoolSize_u16 | Last block of the EEL pool |

### 4.3.3 r_fdl_request_t

**Type definition:**

```
typedef volatile struct R_FDL_REQUEST_T
{
    r_fdl_command_t     command_enu;
    uint32_t            bufAddr_u32;
    uint32_t            idx_u32;
    uint16_t            cnt_u16;
    r_fdl_accessType_t  accessType_enu;
    r_fdl_status_t      status_enu;
} r_fdl_request_t;
```

**Description:** This structure is the central type for the request-response-oriented dialog for the command execution (see section 3.2 “Request-response oriented dialog”). Not every element of this structure is required for each command. However, all members of the request variable must be initialized once before usage. Please refer to section 4.5 “Commands” for a more detailed description of the structure elements command-specific usage.

For simplification, `idx_u32` structure member is a virtual address that starts at `0x0` and not at the address at which Data Flash is mentioned in the hardware user manual.

**Member / Value:**

| Member / Value | Description |
|---|---|
| command_enu | User command to execute:<br>• R_FDL_CMD_ERASE<br>• R_FDL_CMD_WRITE<br>• R_FDL_CMD_BLANKCHECK<br>• R_FDL_CMD_READ |
| bufAddr_u32 | Source/Destination buffer address for Write/Read operations |
| idx_u32 | Bidirectional:<br>• start block number when starting block based commands (erase) or<br>• start word address when starting address based commands (write, blank check, read) or<br>• failure address in case of blank check or ECC read commands |
| cnt_u16 | Number of blocks (64 bytes) to operate in case of erase command. Number of words (4 bytes) to operate for all the other commands. |
| accessType_enu | Data Flash access originator:<br>• R_FDL_ACCESS_USER or<br>• R_FDL_ACCESS_EEL |
| status_enu | Status/Error codes returned by the library, see 4.3.6 "r_fdl_status_t" |

### 4.3.4 r_fdl_command_t

**Type definition:**

```
typedef enum R_FDL_COMMAND_T
{
    R_FDL_CMD_ERASE,
    R_FDL_CMD_WRITE,
    R_FDL_CMD_BLANKCHECK,
    R_FDL_CMD_READ
} r_fdl_command_t;
```

**Description:** User command to execute. This type is used within the structure `r_fdl_request_t` (see section 4.3.3 "r_fdl_request_t") in order to specify which command shall be executed via the function `R_FDL_Execute`. A detailed description of each command can be found in section 4.5 “Commands”.

**Member / Value:**

| Member / Value | Description |
|---|---|
| R_FDL_CMD_ERASE | Erase Data Flash block(s) |
| R_FDL_CMD_WRITE | Write Data Flash word(s) |
| R_FDL_CMD_BLANKCHECK | Blank check certain Data Flash area |
| R_FDL_CMD_READ | Read from Data Flash and return data and possible ECC errors |

### 4.3.5 r_fdl_accessType_t

**Type definition:**

```
typedef enum R_FDL_ACCESS_TYPE_T
{
    R_FDL_ACCESS_NONE,
    R_FDL_ACCESS_USER,
    R_FDL_ACCESS_EEL
} r_fdl_accessType_t;
```

**Description:** In order to initiate a Data Flash operation, the access type to the Data Flash must be set depending on the configured pool that will be accessed. The pool ranges are defined in the FDL descriptor, passed to the `R_FDL_Init` function (please check Figure 6: “Flash Access Rights”).

After each operation the access right is reset to `R_FDL_ACCESS_NONE` to prevent accidental access.

**Member / Value:**

| Member / Value | Description |
|---|---|
| R_FDL_ACCESS_NONE | FDL internal value. Not used by the application |
| R_FDL_ACCESS_USER | Application wants to execute an FDL operation in the User-pool Data Flash area |
| R_FDL_ACCESS_EEL | Application wants to execute an FDL operation in the EEL-pool Data Flash area |

### 4.3.6 r_fdl_status_t

**Type definition:**

```
typedef enum R_FDL_STATUS_T
{
    R_FDL_OK,
    R_FDL_BUSY,
    R_FDL_SUSPENDED,
    R_FDL_ERR_CONFIGURATION,
    R_FDL_ERR_PARAMETER,
    R_FDL_ERR_PROTECTION,
    R_FDL_ERR_REJECTED,
    R_FDL_ERR_WRITE,
    R_FDL_ERR_ERASE,
    R_FDL_ERR_BLANKCHECK,
    R_FDL_ERR_COMMAND,
    R_FDL_ERR_ECC_SED,
    R_FDL_ERR_ECC_DED,
    R_FDL_ERR_INTERNAL
} r_fdl_status_t;
```

**Description:** This enumeration type defines all possible status and error-codes that can be generated by the FDL. Some error codes are command specific and are described in detail in section 4.5 “Commands”.

**Member / Value:**

| Member / Value | Description |
|---|---|
| R_FDL_OK | FDL operation successfully finished |
| R_FDL_BUSY | FDL operation is still ongoing |
| R_FDL_SUSPENDED | Data Flash operation is suspended |
| R_FDL_ERR_CONFIGURATION | The FDL configuration (descriptor) was wrong |
| R_FDL_ERR_PARAMETER | An error was found in the given parameter(s) |
| R_FDL_ERR_PROTECTION | FDL operation stopped due to hardware error, wrong access rights or wrong conditions |
| R_FDL_ERR_REJECTED | A flow error occurred (e.g. library not initialized, other operation on-going) |
| R_FDL_ERR_WRITE | Data Flash write error |
| R_FDL_ERR_ERASE | Data Flash erase error |
| R_FDL_ERR_BLANKCHECK | The blank check command was stopped because the specified area is not blank |
| R_FDL_ERR_COMMAND | Unknown command |
| R_FDL_ERR_ECC_SED | Single bit error detected by ECC |
| R_FDL_ERR_ECC_DED | Double bit error detected by ECC |
| R_FDL_ERR_INTERNAL | The current FDL command stopped due to an library internal error (e.g. hardware errors that should never occur or library errors which were not expected and might result from library data manipulation by the application) |

## 4.4 Functions

The API functions, grouped by their role in the interface:

Initialization:

- R_FDL_Init

Flash Operations:

- R_FDL_Execute
- R_FDL_Handler

Operation control:

- R_FDL_SuspendRequest
- R_FDL_ResumeRequest
- R_FDL_StandBy
- R_FDL_WakeUp

Administration:

- R_FDL_GetVersionString

The following sub-chapters describe the Flash operations that can be initiated and controlled by the library. The operations are initiated by a library function R_FDL_Execute and later on controlled by the library function R_FDL_Handler.

All FDL interface functions are prototyped in the header file r_fdl.h.

## 4.4.1 Initialization

### 4.4.1.1 R_FDL_Init

**Outline:** Initialization of the Data Flash Access Library.

**Interface:** C Interface

```
r_fdl_status_t R_FDL_Init (const r_fdl_descriptor_t * descriptor_pstr);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| descriptor_pstr | r_fdl_descriptor_t * | R | FDL configuration descriptor (see section 4.3.2 “r_fdl_request_t”) |

Return value

| Type | Description |
|---|---|
| r_fdl_status_t | • R_FDL_OK<br>Operation finished successfully.<br>• R_FDL_ERR_CONFIGURATION<br>Wrong parameters have been passed to the FDL:<br>  • Descriptor address is NULL<br>  • FDL-pool is zero<br>  • EEL-pool ends beyond FDL-pool edge<br>  • Specified CPU clock is outside limits for this device<br>• R_FDL_ERR_INTERNAL<br>Initialization failed due to various factors (insufficient stack space, unknown hardware or software issues) |

**Pre-conditions:** Interrupt execution shall be disabled for a brief time during execution of this function. This must either be done in advance by the user, or the user must properly configure provided callback macro functions in fdl_cfg.h (see description and example below).

**Post-conditions:** None

**Description:** This function is executed before any execution of FDL Flash operation.

Function checks the input parameters and initializes the hardware and software.

**Note:**

**This function will temporarily disable Code Flash. Please refer to Chapter 6** Cautions **for limitations that must be considered.**

**Example:**

```
const r_fdl_descriptor_t  sampleApp_fdlConfig_enu =
{
    AUTHENTICATION_ID,
    CPU_FREQUENCY_MHZ,
    FDL_POOL_SIZE,
    EEL_POOL_START,
    EEL_POOL_SIZE
};

r_fdl_status_t ret;

ret = R_FDL_Init (&sampleApp_fdlConfig_enu);

if (ret != R_FDL_OK)
{
    /* Error handler */
}
```

**Example:** for setting the protected section with callbacks provided in the sample application

```
#define FDL_CRITICAL_SECTION_BEGIN      FDL_User_CriticalSetionBegin();
#define FDL_CRITICAL_SECTION_END        FDL_User_CriticalSetionEnd();
```

## 4.4.2 Flash Operations

### 4.4.2.1 R_FDL_Execute

**Outline:** Initiate a Data Flash operation.

**Interface:** C Interface

```
void R_FDL_Execute (r_fdl_request_t * request_pstr);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| request_pstr | r_fdl_request_t * | RW | This argument points to a request structure defining the command, command parameters and also the execution results.<br>A more detailed description of request structure can be found in section 4.3.3 “r_fdl_request_t”. |

Return value

| Type | Description |
|---|---|
| none | |

**Pre-conditions:** `R_FDL_Init` must have been executed successfully.

**Post-conditions:** Call `R_FDL_Handler` until the Flash operation is finished. This is reported by the request structure status return value (value changes from `R_FDL_BUSY` to a different value).

The user application must not modify members of the request structure while the command is in operation.

**Description:** The execute function initiates all Flash modification operations. The operation type and operation parameters are passed to the FDL by a request structure, the status and the result of the operation are returned to the user application also by the same structure. The required parameters as well as the possible return values depend on the operation to be started.

This function only starts a hardware operation according to the command to be executed. The command processing must be controlled and stepped forward by the handler function `R_FDL_Handler`.

Possible commands, parameters and return values are described into chapter 4.5 “Commands“.

**Example:** Erase blocks 0 to 3.

```
r_fdl_request_t myRequest;

myRequest.command_enu      = R_FDL_CMD_ERASE;
myRequest.idx_u32          = 0;
myRequest.cnt_u16          = 4;
myRequest.accessType_enu   = R_FDL_ACCESS_USER;

R_FDL_Execute (&myRequest);
while (myRequest.status_enu == R_FDL_BUSY)
{
    R_FDL_Handler ();
}

if (myRequest.status_enu != R_FDL_OK)
{
    /* Error handler */
}
```

**Example:** Write 8 bytes starting from addresses 0x10.

```
r_fdl_request_t myRequest;

uint32_t data[]            = { 0x11223344, 0x55667788 };

myRequest.command_enu      = R_FDL_CMD_WRITE;
myRequest.idx_u32          = 0x10;
myRequest.cnt_u16          = 2;
myRequest.bufAddr_u32      = (uint32_t)&data[0];
myRequest.accessType_enu   = R_FDL_ACCESS_USER;

R_FDL_Execute (&myRequest);
while (myRequest.status_enu == R_FDL_BUSY)
```

```
{
    R_FDL_Handler ();
}

if (myRequest.status_enu != R_FDL_OK)
{
    /* Error handler */
}
```

**Example:** Blank Check addresses from 0x10 to 0x17.

```
r_fdl_request_t myRequest;

myRequest.command_enu      = R_FDL_CMD_BLANKCHECK;
myRequest.idx_u32          = 0x10;
myRequest.cnt_u16          = 2;
myRequest.accessType_enu   = R_FDL_ACCESS_USER;

R_FDL_Execute(&myRequest);

while (myRequest.status_enu == R_FDL_BUSY)
{
    R_FDL_Handler();
}

if (myRequest.status_enu != R_FDL_OK)
{
    /* Error handler */
}
```

**Example:** Read two words starting from address 0x10.

```
r_fdl_request_t myRequest;

uint32_t data[2];

myRequest.command_enu      = R_FDL_CMD_READ;
myRequest.idx_u32          = 0x10;
myRequest.cnt_u16          = 2;
myRequest.bufAddr_u32      = (uint32_t)&data[0];
myRequest.accessType_enu   = R_FDL_ACCESS_USER;

R_FDL_Execute(&myRequest);

if (myRequest.status_enu != R_FDL_OK)
{
    /* Error handler */
}
```

### 4.4.2.2 R_FDL_Handler

**Outline:** This function needs to be called repeatedly in order to drive pending commands and observe their progress.

**Interface:** C Interface

```
void R_FDL_Handler (void);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| None | | | |

Return value

| Type | Description |
|---|---|
| None | |

**Pre-conditions:** `R_FDL_Init` and `R_FDL_Execute` must have been executed successfully.

**Post-conditions:** The status of a pending FDL command may be updated, i.e. the `status_enu` member of the corresponding request structure is written.

**Description:** The function needs to be called regularly in order to drive pending commands and observe their progress. Thereby, the command execution is performed state by state. When a command execution is finished, the request status variable (structural element `status_enu` of `r_fdl_request_t`) is updated with the status/error code of the corresponding command execution.

**Note:**

When no command is being processed, `R_FDL_Handler` consumes few CPU cycles.

**Example:**

```
while (true)
{
    R_FDL_Handler();
    User_Task_A();
    User_Task_B();
    User_Task_C();
    User_Task_D();
}
```

## 4.4.3 Operation control

### 4.4.3.1 R_FDL_SuspendRequest

**Outline:** This function requests suspending a Flash operation in order to be able to do other Flash operations.

**Interface:** C Interface

```
r_fdl_status_t R_FDL_SuspendRequest (void);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| None | | | |

Return value

| Type | Description |
|---|---|
| r_fdl_status_t | • R_FDL_OK<br>Operation finished successfully<br>• R_FDL_ERR_REJECTED<br>Wrong library handling flow:<br>◦ No operation is ongoing<br>◦ FDL is already in suspended state |

**Pre-conditions:** A Flash operation must have been started and not yet finished (request structure status value is `R_FDL_BUSY`). The FDL must not be processing another suspend request.

**Post-conditions:** Call `R_FDL_Handler` until the library is suspended (status `R_FDL_SUSPENDED`)

If the function returned successfully, no further error check of the suspend procedure is necessary, as a potential error is saved and restored on `R_FDL_ResumeRequest`.

The request structure used before suspend shall not be modified by the command(s) issued during suspended state.

**Description:** This function requests suspending a Flash operation in order to be able to do other Flash operations.

**Example:**

```
r_fdl_status_t  srRes_enu;
r_fdl_request_t myReq_str_str;
uint32_t        i;

/* Start Erase operation */
myReq_str_str.command_enu     = R_FDL_CMD_ERASE;
myReq_str_str.idx_u32         = 0;
myReq_str_str.cnt_u16         = 4;
myReq_str_str.accessType_enu  = R_FDL_ACCESS_USER;

R_FDL_Execute (&myReq_str_str);
```

```
/* Now call the handler some times */
i = 0;
while ( (myReq_str_str.status_enu == R_FDL_BUSY) && (i < 10) )
{
    R_FDL_Handler ();
    i++;
}

/* Suspend request and wait until suspended */
srRes_enu = R_FDL_SuspendRequest ();

if (R_FDL_OK != srRes_enu)
{
    /* error handler */
    while (1)
        ;
}

while (R_FDL_SUSPENDED != myReq_str_str.status_enu)
{
    R_FDL_Handler ();
}


/* Now the FDL is suspended and we can handle other operations or read the Data
Flash ... */

/* Erase resume */
srRes_enu = R_FDL_ResumeRequest();

if (R_FDL_OK != srRes_enu)
{
    /* Error handler */
}

/* Finish the erase */
while (myReq_str_str.status_enu == R_FDL_SUSPENDED)
{
    R_FDL_Handler();
}
while (myReq_str_str.status_enu == R_FDL_BUSY)
{
    R_FDL_Handler();
}

if (R_FDL_OK != myReq_str_str.status_enu)
{
    /* Error handler */
}
```

### 4.4.3.2 R_FDL_ResumeRequest

**Outline:** This function requests to resume the FDL operation after suspending.

**Interface:** C Interface

```
r_fdl_status_t R_FDL_ResumeRequest (void);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| None | | | |

Return value

| Type | Description |
|---|---|
| r_fdl_status_t | • R_FDL_OK<br>Operation finished successfully<br>• R_FDL_ ERR_REJECTED<br>Wrong library handling flow: FDL is not in suspended or suspend pending state |

**Pre-conditions:** The library must be in suspended state.

**Post-conditions:** Call `R_FDL_Handler` until the library operation is resumed.

**Description:** This function requests to resume the FDL operation after suspending. The resume is just requested by this function. Resume handling is done by the `R_FDL_Handler` function.

**Example:** See `R_FDL_SuspendRequest`.

### 4.4.3.3 R_FDL_StandBy

**Outline:** This function suspends an ongoing flash operation.

**Interface:** C Interface

```
r_fdl_status_t R_FDL_StandBy (void);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| None | | | |

Return value

| Type | Description |
|---|---|
| r_fdl_status_t | • R_FDL_OK<br>FDL operation finished successfully<br>• R_FDL_BUSY<br>The started Flash operation is still on-going<br>• R_FDL_ ERR_REJECTED<br>Flow error:<br>  • Library is not initialized<br>  • Library is already in stand-by mode |

**Pre-conditions:** R_FDL_Init must have been executed successfully.

FDL is not in stand-by mode.

**Post-conditions:** Repeat the execution of the R_FDL_StandBy function until the state indicated by the function changes from R_FDL_BUSY.

Do not execute functions R_FDL_Execute, R_FDL_SuspendRequest, R_FDL_ResumeRequest or R_FDL_StandBy when FDL is in stand-by state.

**Description:** This function suspends an ongoing flash operation and brings FDL into stand-by state. The system can then change to special states (e.g. change power mode).

Function does not necessarily immediately suspend any Flash operation, as suspend might be delayed by the device internal hardware or might not be supported at all (only Erase and Write are suspendable). So, the function R_FDL_StandBy tries to suspend the Flash operation and returns R_FDL_BUSY as long as a Flash operation is on-going. If suspend was not possible (e.g. blank check operation), R_FDL_BUSY is returned until the operation is finished normally.

So, in order to be sure to have no Flash operation on-going, the function must be called continuously until the function does no longer return R_FDL_BUSY or until a timeout occurred.

After stand-by, it is mandatory to call R_FDL_WakeUp to resume normal FDL operation again. The prescribed sequence in case of using R_FDL_StandBy/R_FDL_WakeUp is:

- any FDL command is in operation
- call R_FDL_StandBy until it does no longer return R_FDL_BUSY
- put device in power save (stand-by) mode
- device wake-up
- call R_FDL_WakeUp
- continue with initial FDL command

**Note:**

**Please consider not entering a power save mode (e.g. Deep Stop mode) which resets the Flash hardware, alter stack or library variables, because a resume of the previous operation is not possible afterwards. The library is not able to detect this failure.**

**Example:**

```
r_fdl_status_t  fdlRet_enu;
r_fdl_request_t myReq_str_str;
```

```
/* Start Erase operation */
myReq_str_str.command_enu     = R_FDL_CMD_ERASE;
myReq_str_str.idx_u32         = 0;
myReq_str_str.cnt_u16         = 4;
myReq_str_str.accessType_enu  = R_FDL_ACCESS_USER;

R_FDL_Execute (&myReq_str_str);

...
do
{
    fdlRet = R_FDL_StandBy ();
}
while (R_FDL_BUSY == fdlRet);
if (R_FDL_OK != fdlRet)
{
    /* error handler */
}

...
/* device enters power save mode */
...

...
/* device recovers from power save mode */
...

fdlRet = R_FDL_WakeUp ();
if (R_FDL_OK != fdlRet)
{
    /* error handler */
}

/* Finish erase command */

while (myReq_str_str.status_enu == R_FDL_BUSY)
{
    R_FDL_Handler ();
}

if (R_FDL_OK != myReq_str_str.status_enu)
{
    /* Error handler */
    while (1)
        ;
}
```

### 4.4.3.4 R_FDL_WakeUp

**Outline:** This function wakes-up the library from Stand-by.

**Interface:** C Interface

```
r_fdl_status_t R_FDL_WakeUp (void);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| None | | | |

Return value

| Type | Description |
|---|---|
| r_fdl_status_t | • R_FDL_OK<br>Operation finished successfully<br>• R_FDL_ ERR_REJECTED<br>Wrong library handling flow: FDL is not in stand-by state |

**Pre-conditions:** The library must be in stand-by mode.

The hardware conditions (CPU frequency, voltage, etc...) must be restored to the state before issuing the stand-by request.

**Post-conditions:** None

**Description:** The main purpose of this function is to wake-up the library from the stand-by mode and resume Flash hardware. For more information see chapter 3.6 “Stand-by and Wake-up functionality”.

**Example:** See `R_FDL_StandBy`.

### 4.4.4 Administration

#### 4.4.4.1 R_FDL_GetVersionString

**Outline:** This function returns the pointer to the null terminated library version string.

**Interface:** C Interface

```
(const uint8_t*) R_FDL_GetVersionString (void);
```

**Arguments:** Parameters

| Argument | Type | Access | Description |
|---|---|---|---|
| None | | | |

Return value

| Type | Description |
|---|---|
| const uint8_t * | The library version is a string value in the following format: “DH850T01xxxxxYZabcD”<br>Please check function description below for details. |

**Pre-conditions:** None

**Post-conditions:** None

**Description:**

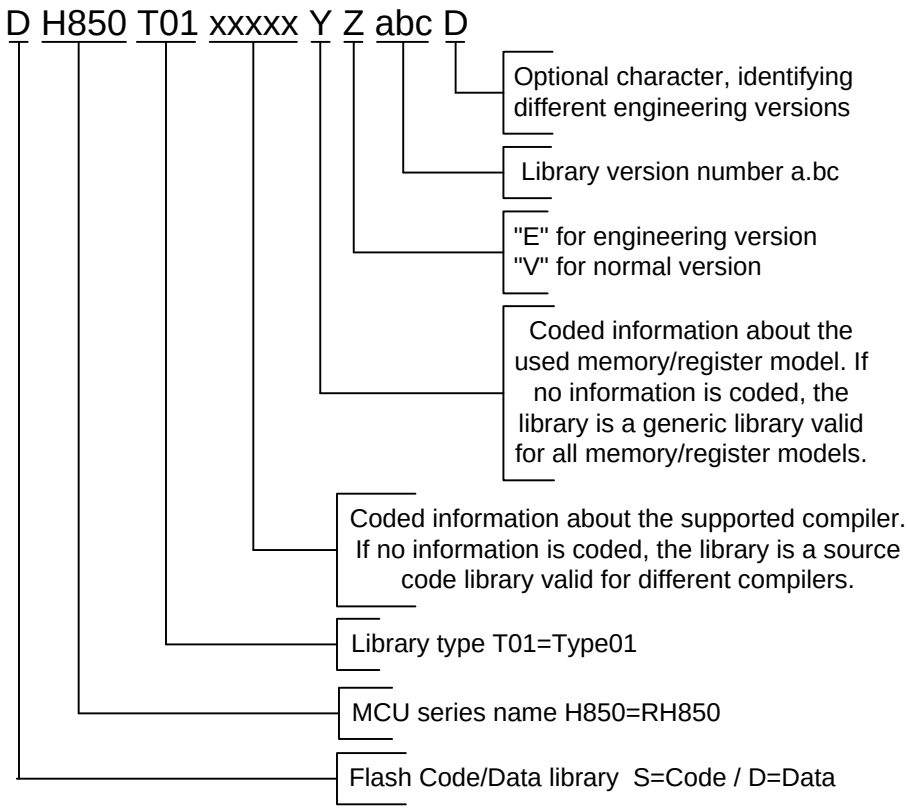



**Figure 12: Version string**

**Example:**

```
uint8_t * vstr = (uint8_t *)R_FDL_GetVersionString ();
```

## 4.5 Commands

The following sub-chapters describe the Flash operations that can be initiated and controlled by the library.

In general, all FDL commands can be handled in the same way as illustrated in Figure 13:

1. The requester fills up the private request variable `my_request` (command definition).
2. The requester tries to initiate the command execution by `R_FDL_Execute(&my_request)`.
3. The requester has to call `R_FDL_Handler` to proceed the FDL command execution as long the request is being processed (i.e. `my_request.status_enu == R_FDL_BUSY`).
4. After finishing the command (i.e. `my_request.status_enu != R_FDL_BUSY`) the requester has to analyse the status to detect potential errors.

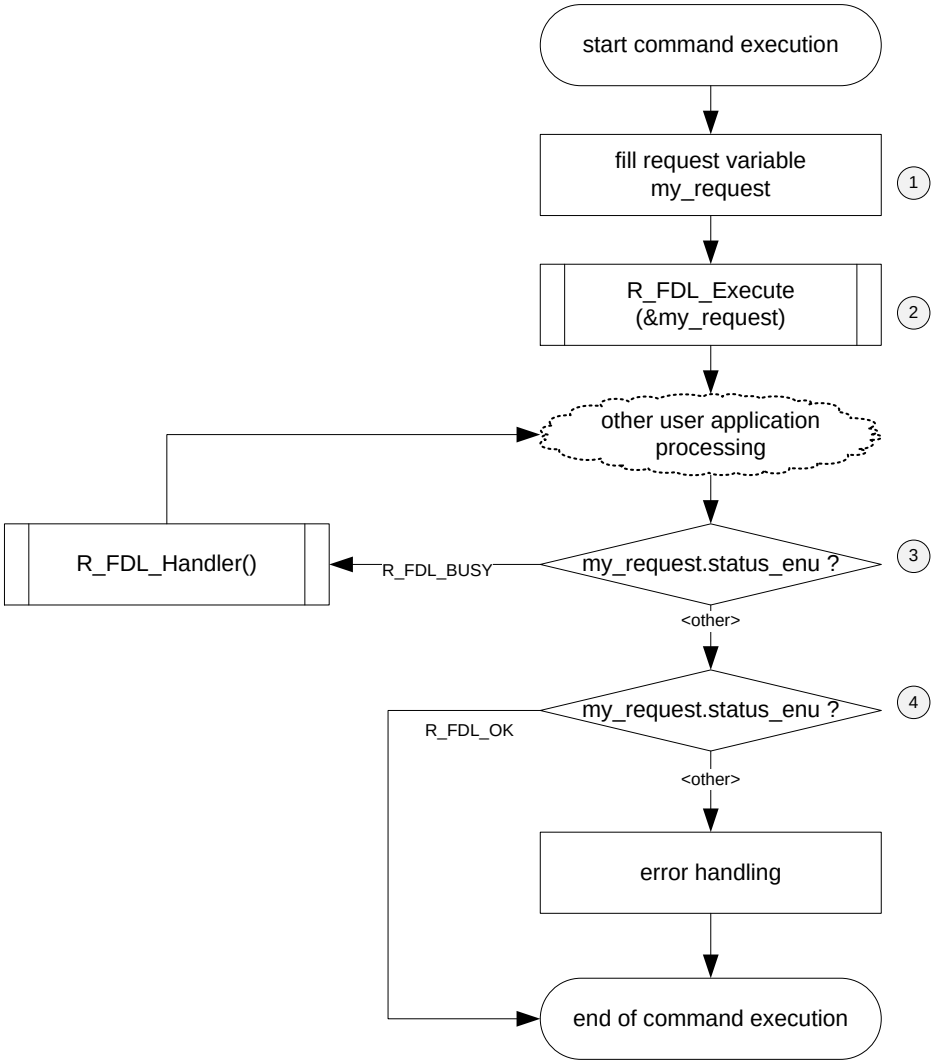



**Figure 13: Generic command execution flow**

### 4.5.1 R_FDL_CMD_ERASE

The erase command can be used to erase a number of Flash blocks defined by a start block and the number of blocks.

The members of the request structure given to `R_FDL_Execute` are described in the following table:

**Table 3: Request structure usage for erase command**

| Structure member | Value | Description |
|---|---|---|
| command_enu | R_FDL_CMD_ERASE | Request a block erase operation |
| bufAddr_u32 | - | Not used |
| idx_u32 | {uint32_t number} | Number of the first block to be erased. Flash blocks are defined by the erase granularity that is 64 Bytes, e.g.:<br>block 0: 0x00 ... 0x3F<br>block 1: 0x40 ... 0x7F<br>... |
| cnt_u16 | {uint16_t number} | Numbers of blocks to erase |
| accessType_enu | R_FDL_ACCESS_USER / R_FDL_ACCESS_EEL | Selects the Flash pool in which the command will be able to operate. |
| status_enu | - | This is an output member. It contains the status of the operation during and after the execution. Possible values are described in the next table. |

The following table describes all possible status returns:

**Table 4: Erase operation returned status**

| Status | Background and Handling | |
|---|---|---|
| R_FDL_BUSY | meaning | Operation started successfully |
| | reason | No problems during execution |
| | remedy | Call `R_FDL_Handler` until the Flash operation is finished, reported by the request structure status return value |
| R_FDL_OK(1) | meaning | Operation finished successfully |
| | reason | No problems during execution |
| | remedy | Nothing |
| R_FDL_SUSPENDED(1) | meaning | An on-going Flash operation was successfully suspended |
| | reason | Suspend processing successfully finished |
| | remedy | Start another operation or resume the suspended operation |
| R_FDL_ERR_PARAMETER(2) | meaning | Current command is rejected |
| | reason | Wrong command parameters:<br>• access is made outside of physically available Data Flash<br>• command shall operate in User-pool but `accessType_enu` is not `R_FDL_ACCESS_USER`<br>• command shall operate in EEL-pool but `accessType_enu` is not `R_FDL_ACCESS_EEL`<br>• `cnt_u16` is 0 or it is too big |

| Status | Background and Handling | |
|---|---|---|
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_PROTECTION | meaning | Current command is rejected |
| | reason | • To gain robustness, the parameter check is repeated right before Flash modification and returns the protection error in case of a violation (e.g. due to an unwanted variable modification)<br>• Other device specific protection mechanisms (e.g. security unit like ICU or FHVE protection mechanisms prevent Flash operations. |
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_REJECTED(2) | meaning | Current command is rejected |
| | reason | Another operation is ongoing |
| | remedy | Request again the command when the preceding command has finished |
| R_FDL_ERR_ERASE(1) | meaning | At least one bit within the specified blocks is not erased |
| | reason | Hardware defect: one or more Flash bits could not be erased completely |
| | remedy | A Flash block respectively the complete Data Flash should be considered as defect |
| R_FDL_ERR_INTERNAL(1) | meaning | A library internal error occurred, which could not happen in case of normal application execution |
| | reason | • Application bug (e.g. program run-away, destroyed program counter) or hardware problem<br>• Only on older F1x devices: failed ID code authentication supplied in the device descriptor. See section 4.2 ”Run-time configuration” for details about ID code |
| | remedy | Refrain from further Flash operations and investigate in the root cause |

(1) `R_FDL_Execute` will never set this status code

(2) `R_FDL_Handler` will never set this status code

### 4.5.2 R_FDL_CMD_WRITE

The write command can be used to write a number of data words located in the RAM into the Data Flash at the location specified by the virtual target address.

**Note:**

It is not allowed to “Overwrite” data, which means writing data to already partly or completely written Flash area. Please always erase the targeted area before writing into it.

The members of the request structure given to `R_FDL_Execute` are described in the following table:

**Table 5: Request structure usage for write command**

| Structure member | Value | Description |
|---|---|---|
| command_enu | R_FDL_CMD_WRITE | Request a write operation |

| Structure member | Value | Description |
|---|---|---|
| bufAddr_u32 | {uint32_t number} | Address of the buffer containing the source data to be written. |
| idx_u32 | {uint32_t number} | The virtual start address for writing in Data Flash aligned to word size (4 Bytes). |
| cnt_u16 | {uint16_t number} | Number of words to write. |
| accessType_enu | R_FDL_ACCESS_USER / R_FDL_ACCESS_EEL | Selects the Flash pool in which the command will be able to operate. |
| status_enu | - | This is an output member. It contains the status of the operation during and after the execution. Possible values are described in the next table. |

The following table describes all possible status returns:

**Table 6: Write operation returned status**

| Status | Background and Handling | |
|---|---|---|
| R_FDL_BUSY | meaning | Operation started successfully |
| | reason | No problems during execution |
| | remedy | Call `R_FDL_Handler` until the Flash operation is finished, reported by the request structure status return value |
| R_FDL_OK(1) | meaning | Operation finished successfully |
| | reason | No problems during execution |
| | remedy | Nothing |
| R_FDL_SUSPENDED(1) | meaning | An on-going Flash operation was successfully suspended |
| | reason | Suspend processing successfully finished |
| | remedy | Start another operation or resume the suspended operation |
| R_FDL_ERR_PARAMETER(2) | meaning | Current command is rejected |
| | reason | Wrong command parameters:<br>• access is made outside of physically available Data Flash<br>• command shall operate in User-pool but `accessType_enu` is not `R_FDL_ACCESS_USER`<br>• command shall operate in EEL-pool but `accessType_enu` is not `R_FDL_ACCESS_EEL`<br>• `cnt_u16` is 0 or it is too big<br>• flash writing address is not aligned with granularity (4 bytes) |
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_PROTECTION | meaning | Current command is rejected |

| Status | Background and Handling | |
|---|---|---|
| | reason | • To gain robustness, the parameter check is repeated right before Flash modification and returns the protection error in case of a violation (e.g. due to an unwanted variable modification)<br>• Other device specific protection mechanisms (e.g. security unit like ICU or FHVE protection mechanisms prevent Flash operations. |
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_REJECTED[2] | meaning | Current command is rejected |
| | reason | Another operation is ongoing |
| | remedy | Request again the command when the preceding command has finished |
| R_FDL_ERR_WRITE[1] | meaning | At least one data could not be written correctly |
| | reason | • User flow defect: tried to “overwrite” data (write into non-erased cells)<br>• Hardware defect: one or more Flash bits could not be written |
| | remedy | Erase write area before writing.<br>A Flash block respectively the complete Data Flash should be considered as defect |
| R_FDL_ERR_INTERNAL[1] | meaning | A library internal error occurred, which could not happen in case of normal application execution |
| | reason | • Application bug (e.g. program run-away, destroyed program counter) or hardware problem<br>• Only on older F1x devices: failed ID code authentication supplied in the device descriptor. See section 4.2 ”Run-time configuration” for details about ID code |
| | remedy | Refrain from further Flash operations and investigate in the root cause |

[1] `R_FDL_Execute` will never set this status code

[2] `R_FDL_Handler` will never set this status code

### 4.5.3 R_FDL_CMD_BLANKCHECK

The blank check command can be used by the requester to check whether a specified amount of memory starting from a specified address is written. This command will stop at the first memory location that is not erased with status `R_FDL_ERR_BLANKCHECK`.

**Notes:**

1. On blank check fail, the cells are surely not blank. This might result from successfully written cells, but also from interrupted erase or write operations.
2. On blank check pass, the cells are surely not written. This might result from successfully erased cells, but also from interrupted erase or write operations.
3. Depending on the Flash operations use case (e.g. EEPROM emulation) it may be necessary to log the Flash operations results in order to be sure that Flash cells are correctly written or erased. The way of logging depends on the use case (e.g. as part of a EEPROM Emulation concept)

4. Internally blankcheck operation is split into smaller operations every time the operation crosses a 0x4000 bytes boundary. This means that time to suspend is not going to exceed the time to fully perform a blankcheck on 0x4000 bytes.

The members of the request structure given to `R_FDL_Execute` are described in the following table:

**Table 7: Request structure usage for blank check command**

| Structure member | Value | Description |
|---|---|---|
| command_enu | R_FDL_CMD_BLANKCHECK | Request a blank check operation |
| bufAddr_u32 | - | Not used |
| idx_u32 | {uint32_t number} | Input: The virtual start address for performing blank check in data flash. Must be word (4 bytes) aligned.<br>Output: Fail address in case of blank check error, unchanged if the operation finishes with `R_FDL_OK`. |
| cnt_u16 | {uint16_t number} | Number of words (4 bytes) to check |
| accessType_enu | R_FDL_ACCESS_USER / R_FDL_ACCESS_EEL | Selects the Flash pool in which the command will be able to operate. |
| status_enu | - | This is an output member. It contains the status of the operation during and after the execution. Possible values are described in the next table. |

The following table describes all possible status returns:

**Table 8: Blank check operation returned status**

| Status | Background and Handling | |
|---|---|---|
| R_FDL_BUSY | meaning | Operation started successfully |
| | reason | No problems during execution |
| | remedy | Call `R_FDL_Handler` until the Flash operation is finished, reported by the request structure status return value |
| R_FDL_OK(1) | meaning | Operation finished successfully |
| | reason | No problems during execution |
| | remedy | Nothing |
| R_FDL_SUSPENDED(1) | meaning | An on-going Flash operation was successfully suspended |
| | reason | Suspend processing successfully finished |
| | remedy | Start another operation or resume the suspended operation |
| R_FDL_ERR_PARAMETER(2) | meaning | Current command is rejected |

| Status | Background and Handling | |
|---|---|---|
| | reason | Wrong command parameters:<br>• access is made outside of physically available Data Flash<br>• command shall operate in User-pool but `accessType_enu` is not `R_FDL_ACCESS_USER`<br>• command shall operate in EEL-pool but `accessType_enu` is not `R_FDL_ACCESS_EEL`<br>• `cnt_u16` is 0 or it is too big<br>• flash blank check address is not aligned with granularity (4 bytes) |
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_PROTECTION | meaning | Current command is rejected |
| | reason | • To gain robustness, the parameter check is repeated right before Flash modification and returns the protection error in case of a violation (e.g. due to an unwanted variable modification)<br>• Other device specific protection mechanisms (e.g. security unit like ICU or FHVE protection mechanisms prevent Flash operations. |
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_REJECTED[2] | meaning | Current command is rejected |
| | reason | Another operation is ongoing |
| | remedy | Request again the command when the preceding command has finished |
| R_FDL_ERR_BLANKCHECK[1] | meaning | At least one bit within the specified blocks is not blank |
| | reason | For any bit in the specified range the voltage level is below specification for a blank cell |
| | remedy | Remedy depends on the usage:<br>• nothing to do if only checking the area<br>• perform an erase if needed or<br>• consider the checked area as defect if `R_FDL_CMD_BLANKCHECK` command fails when executed immediately after an erase operation on the same area |
| R_FDL_ERR_INTERNAL[1] | meaning | A library internal error occurred, which could not happen in case of normal application execution |
| | reason | • Application bug (e.g. program run-away, destroyed program counter) or hardware problem<br>• Only on older F1x devices: failed ID code authentication supplied in the device descriptor. See section 4.2 ”Run-time configuration” for details about ID code |
| | remedy | Refrain from further Flash operations and investigate in the root cause |

[1] `R_FDL_Execute` will never set this status code

(2) R_FDL_Handler will never set this status code

### 4.5.4 R_FDL_CMD_READ

The read operation will read a certain address range in the Data Flash and copy the data to the specified target buffer.

A higher level EEPROM Emulation library may want to read Data Flash addresses which are possibly not completely written or erased. Reading those addresses would most probably result in an ECC error interrupt request which must be handled by the user application. This behaviour is usually not intended as an emulation library would have to deal with the errors.

Based on these considerations, the read operation of the FDL temporarily disables the interrupt generation for ECC errors. The status of ECC interrupt generation is restored when the operation is finished. Errors detected during read operation are signalled to the user application by the request structure status_enu variable and idx_u32 variable.

In case of single bit error the data read will be continued and the 1st occurrence of the ECC error will be returned. In case of double bit error, the read operation is stopped and the fail address is returned. In case of a previous single bit error detected, the fail address of the single bit error is overwritten.

Read command execution is synchronous to execution of R_FDL_Execute function. Therefore this command cannot be suspended and does not need to be processed by R_FDL_Handler function.

The members of the request structure given to R_FDL_Execute are described in the following table:

**Table 9: Request structure usage for read command**

| Structure member | Value | Description |
|---|---|---|
| command_enu | R_FDL_CMD_READ | Request a read operation |
| bufAddr_u32 | {uint32_t number} | Data destination buffer address in RAM.<br>**Note:** The buffer must be 32 bit aligned! |
| idx_u32 | {uint32_t number} | Data Flash virtual address from where to read. Must be word (4 bytes) aligned. |
| cnt_u16 | {uint16_t number} | Numbers of words (4 bytes) to read |
| accessType_enu | R_FDL_ACCESS_USER / R_FDL_ACCESS_EEL | Selects the Flash pool in which the command will be able to operate. |
| status_enu | - | This is an output member. It contains the status of the operation during and after the execution. Possible values are described in the next table. |

The following table describes all possible status returns:

**Table 10: Read operation returned status**

| Status | Background and Handling | |
|---|---|---|
| R_FDL_OK | meaning | Operation finished successfully |
| | reason | No problems during execution |
| | remedy | Nothing |
| R_FDL_ERR_PARAMETER | meaning | Current command is rejected |

| Status | Background and Handling | |
|---|---|---|
| | reason | Wrong command parameters:<br>• access is made outside of physically available Data Flash<br>• command shall operate in User-pool but `accessType_enu` is not `R_FDL_ACCESS_USER`<br>• command shall operate in EEL-pool but `accessType_enu` is not `R_FDL_ACCESS_EEL`<br>• `cnt_u16` is 0 or it is too big<br>• flash read address is not aligned with granularity (4 bytes) |
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_PROTECTION | meaning | Current command is rejected |
| | reason | • To gain robustness, the parameter check is repeated right before Flash modification and returns the protection error in case of a violation (e.g. due to an unwanted variable modification)<br>• Other device specific protection mechanisms (e.g. security unit like ICU or FHVE protection mechanisms prevent Flash operations. |
| | remedy | Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_REJECTED | meaning | Current command is rejected |
| | reason | Another operation is ongoing |
| | remedy | Request again the command when the preceding command has finished |
| R_FDL_ERR_ECC_SED | meaning | A data word contains a single bit ECC error. Single bit errors are automatically corrected by the ECC logic.<br>Note: The first occurrence of the fail address is returned. |
| | reason | • Not completely written or erase Flash<br>• Cell level degradation by time<br>• Hardware defect |
| | remedy | A single bit error is acceptable if resulting from degradation by time. Depending on the data handling concept on top, the affected data should be refreshed (Erased and rewritten) in order to remove the error. |
| R_FDL_ERR_ECC_DED | meaning | A data word contains a multiple bit ECC error. This error cannot be corrected by the ECC logic.<br>Note: The read operation will stop at the failing address and the fail address is returned. |
| | reason | • Not completely written or erase Flash<br>• Cell level degradation by time.<br>• Hardware defect |

| Status | Background and Handling | |
|---|---|---|
| | remedy | A multiple bit error can appear when caused by not completely written or erased Flash. The reaction depends on the data handling concept. In case of expected completely written Flash a multiple bit error would mean loss of data. Refrain from further Flash operations and investigate in the root cause |
| R_FDL_ERR_INTERNAL | meaning | A library internal error occurred, which could not happen in case of normal application execution |
| | reason | • Application bug (e.g. program run-away, destroyed program counter) or hardware problem<br>• Only on older F1x devices: failed ID code authentication supplied in the device descriptor. See section 4.2 ”Run-time configuration” for details about ID code |
| | remedy | Refrain from further Flash operations and investigate in the root cause |

The following figure shows the handling of ECC error registers during read command execution:

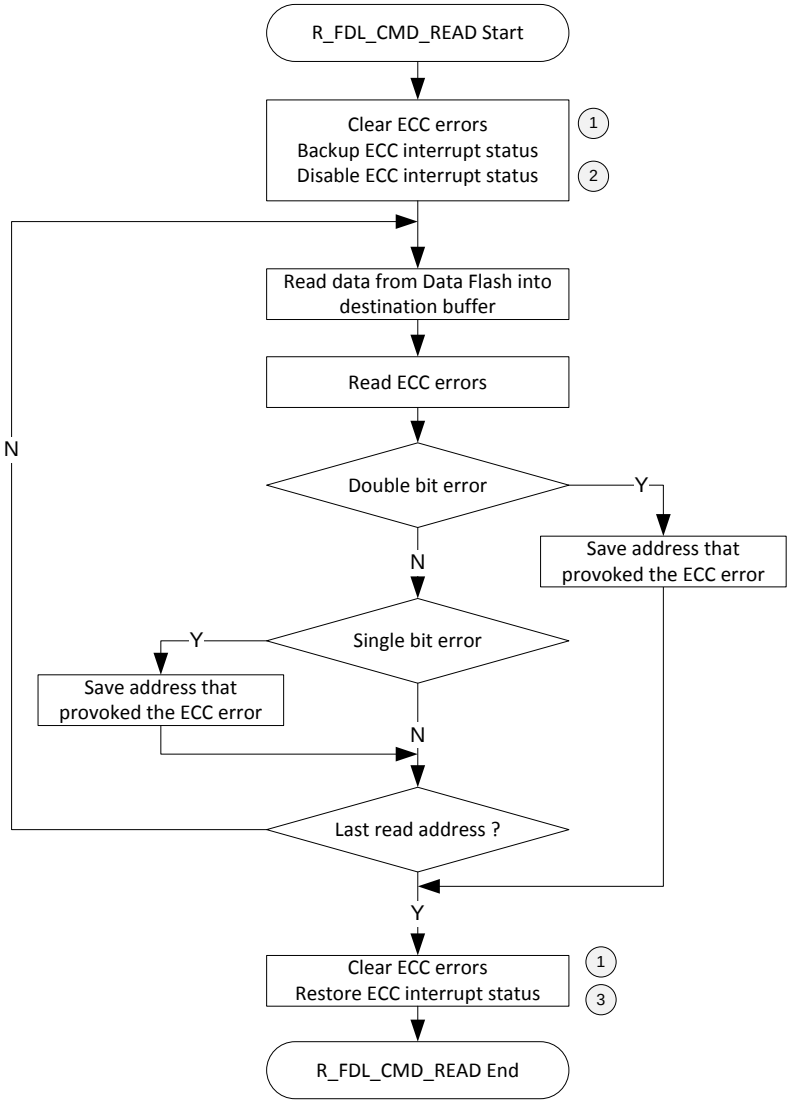



**Figure 14: Handling of ECC error registers during read command**

The user shall take into consideration that the following registers are modified:

1. DFERSTC register is written to clear any errors in DFFSTERSTR
2. DFERRINT register is backed up and cleared
3. DFERRINT register is restored

# Chapter 5 Library Setup and Usage

This chapter contains important information about how to put the FDL into operation and how to integrate it into your application. Please read this chapter carefully — and also especially Chapter 6 “Cautions” — in order to avoid problems and misbehaviour of the library. Before integrating the library into your project however, please make sure that you have read and understood how the FDL works and which basic concepts are used (see Chapter 2 “Architecture” and Chapter 3 “Functional Specifications”).

## 5.1 Obtaining the library

The FDL is provided by means of an installer via the Renesas homepage at

http://www.renesas.eu/update

Please follow the instructions of the installer carefully. Please ensure to always work on the latest version of the library.

## 5.2 File structure

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

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

### 5.2.1 Overview

The following picture contains the library and the application related files:

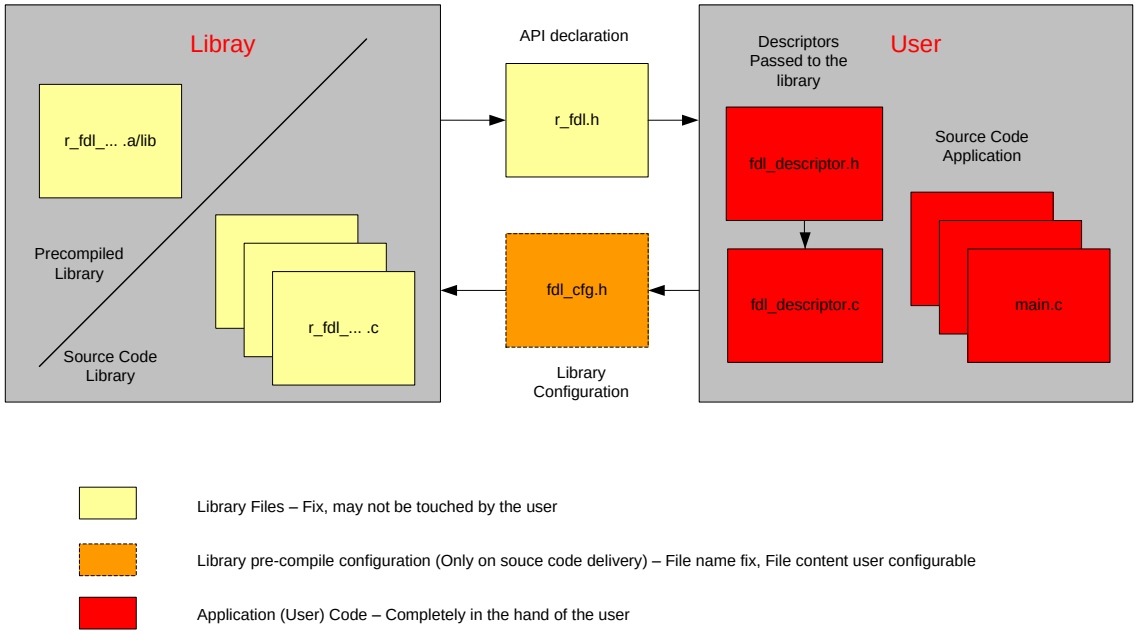



**Figure 15: File structure of library and sample application**

The library must be configured for compilation. The file `fdl_cfg.h` contains defines for that. As it is included by the library source files, the file contents may be modified by the user, but the file name may not.

These files reflect an example, how the library descriptor variable can be built up and passed to the function `R_FDL_Init` for run-time configuration. The structure of the descriptor is defined in `r_fdl_types.h` which needs to be included in the user application. The value definition should be done in the file `fdl_descriptor.h`. The constant variable definition and value assignment should be done in the file `fdl_descriptor.c`. If adding the files `r_fdl_ descriptor.c/h` to the application, only the file `fdl_descriptor.h` needs to be adapted by the user, while `fdl_descriptor.c` may remain unchanged. For usage please refer to chapter 4.2 “Run-time configuration”.

### 5.2.2 Delivery package directory structure and files

The following table contains all files installed by the library installer:

- Files in red belong to the build environment, controlling the compile, link and target build process
- Files in blue belong to the sample application
- Files in green are description files only
- Files in black belong to the FDL

**Table 11: File structure of the FDL package**

| File | | Description |
|---|---|---|
| **<installation_folder>** | | |
| Release.txt | | Library package release notes. |
| **<installation_folder>/Make** | | |
| GNUPublicLicense.txt | | GNU Make utility license file |
| Readme.txt | | Extra information for source code of GNU Make |
| make.exe | | Minimal installation of GNU Make utility |
| libiconv2.dll | | |
| libintl3.dll | | |
| setup.exe | | GNU Make installer package |
| **<installation_folder>/<device_name>/<compiler>** | | |
| Build.bat | | Batch file to build the FDL sample application |
| Clean.bat | | Batch file to clean the FDL sample application |
| Makefile | | Make file that controls the build and clean process |
| **<installation_folder>/<device_name>/<compiler>/Sample** | | |
| dr7f701035_startup.850(2) | <for GHS compiler> | Device and compiler specific start-up code |
| cstart.asm(2) | <for REC compiler> | |
| dr7f701035.ld(2) | <for GHS compiler> | Compiler specific linker directives |
| dr7f701035.dir(2) | <for REC compiler> | |
| dr7f701035_0.h(2)<br>dr7f701035_irq.h(2)<br>io_macros_v2.h(2) | <for GHS compiler> | Definitions of IO registers, interrupt and exceptions vector table, for RH850 device |
| iodefine.h(2)<br>vecttbl.asm(2) | <for REC compiler> | |

| File | Description |
|---|---|
| eel_cfg.h[1] | EEL pre-compile definitions |
| eel_descriptor.c[1] | EEL descriptor used in the sample application |
| eel_descriptor.h[1] | |
| sampleapp.h | Sample application code |
| sampleapp_control.c | |
| sampleapp_main.c | |
| fdl_cfg.h | FDL pre-compile definitions |
| fdl_descriptor.c | FDL descriptor used in the sample application |
| fdl_descriptor.h | |
| fdl_user.c | User defined code for handling interrupts and library pre-initialization |
| fdl_user.h | |
| target.h | Initialization code for target microcontroller |
| r_typedefs.h | C types used by FDL library |
| **<installation_folder>/<device_name>/<compiler>/Sample/EEL**[1] | |
| r_eel.h[1] | EEL API definitions |
| r_eel_mem_map.h[1] | Section mapping definitions |
| r_eel_types.h[1] | User interface type definitions, error and status codes |
| **<installation_folder>/<device_name>/<compiler>/Sample/EEL/lib**[1] | |
| r_eel_basic_fct.c[1] | EEL main source code |
| r_eel_user_if.c[1] | |
| r_eel_global.h[1] | Global variables and settings |
| **<installation_folder>/<device_name>/<compiler>/Sample/FDL** | |
| r_fdl.h | FDL API definitions |
| r_fdl_mem_map.h | Section mapping definitions |
| r_fdl_types.h | User interface type definitions, error and status codes |
| **<installation_folder>/<device_name>/<compiler>/Sample/FDL/lib** | |
| r_fdl_env.h | Internal FDL definitions |
| r_fdl_global.h | Global variables and settings |
| r_fdl_hw_access.c | FDL main source code |
| r_fdl_user_if.c | |

(1) These files are not available if the EEL layer is not part of the delivered package

(2) File names are dependent on the chosen device. Shown filenames are valid for F1L devices

## 5.3 Library resources

### 5.3.1 Linker sections

The following sections are related to the Data Flash Access Library and need to be defined in the linker file (please see sample linker directive file for an example):

**Data sections:**

- R_FDL_DATA

This section contains all FDL internal variables. It can be located either in internal or external RAM.

**Code sections:**

- R_FDL_CONST

This section contains library internal constant data. It can be located anywhere in the code flash.

- R_FDL_TEXT

FDL code section containing the library code. It can be located anywhere in the code flash.

### 5.3.2 Stack and Data Buffer

The FDL utilizes the same stack as specified in the user application. It is the developer's duty to reserve enough stack for the operation of both, user application and FDL. With source code library it is not possible to give an exact value for stack consumption. However, an estimate value for the FDL library is: 268 bytes for GHS compiler and 316 bytes for Renesas compiler.

The data buffer used by the FDL refers to the RAM area in which data is located that is to be written into the data flash. This buffer needs to be allocated and managed by the user.

**Note:**

In order to allocate the stack and data buffer to a user-specified address, please utilize the link directives of your framework.

## 5.4 MISRA Compliance

The FDL code has been tested regarding MISRA™ compliance.

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

**Note:**

"MISRA" is a registered trademark of MIRA Ltd, held on behalf of the MISRA Consortium.

"QA C" is a registered trademark of Programming Research Ltd.

## 5.5 Sample Application

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

After a first compile run, it will be worth playing around with the library in the debugger. By that you will get a feeling for the source code files and the working mechanism of the library. After this exercise it might be easier to understand and follow the recommendations and considerations of this document.

**Note:**

Before the first compile run, the compiler path must be configured in the “Makefile” of the sample application: set the variable `COMPILER_INSTALL_DIR` to the correct compiler directory.

## 5.6 Library configuration

Before using the Data Flash Access library, the library has to be configured and adapted to a certain degree in order to fit the requirements of the user application. For information about configuration settings and handling, please refer to chapter 4.2 “Run-time configuration”.

## 5.7 Basic Reprogramming Flow

The following flow chart shows the basic reprogramming flow for a certain Data Flash range.

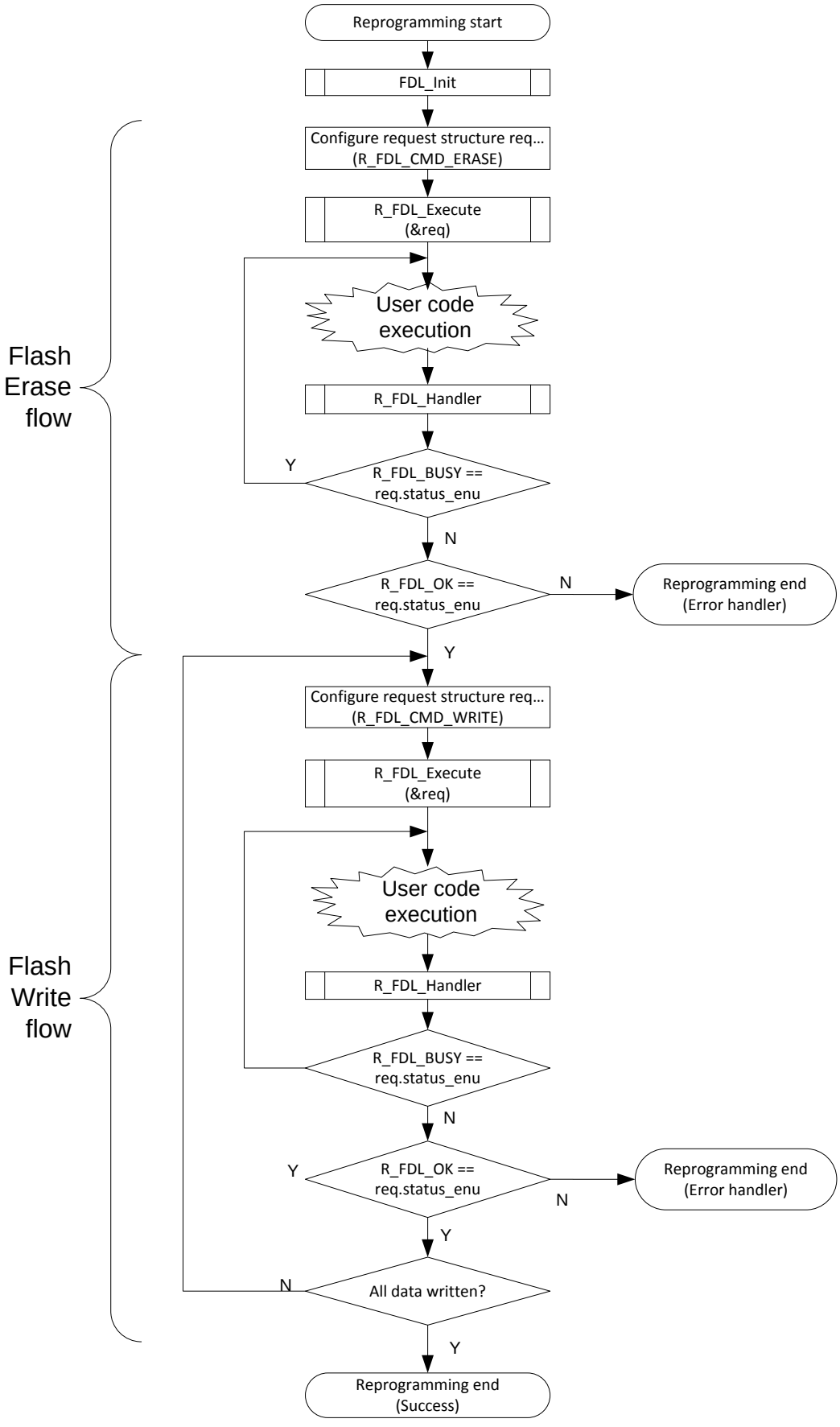



**Figure 16: Basic reprogramming flow**

Error treatment of the FDL functions themselves is not detailed described in the flow charts for simplification reasons.

For details on enabling or disabling access to the Data Flash, refer to the user's manual for the hardware. An example is given by the sample application, file `sample_app_main.c`, functions `FDL_Open` and `FDL_Close`.

## 5.8 R_FDL_Handler calls

Once initiated FDL operations need to be driven forward by successive handler calls. The frequency of these handler calls does have an impact on the FDL operation performance and needs to be adapted to the target application.

In the following, different approaches for calling the `R_FDL_Handler` are compared with respect to their advantages and disadvantages:

- Calling `R_FDL_Handler` repeatedly after starting an operation execution: This approach is also utilized in most of the code examples you can find in this manual. Typically realized in a loop waiting for the operation status not to be busy anymore, this approach results in the best FDL operation performance. However, the CPU is fully loaded and blocked for other tasks as long as the FDL operation is being executed.
- Calling `R_FDL_Handler` in a timed task: By calling the `R_FDL_Handler` periodically, FDL commands can be driven forward while other tasks are processed by the CPU. The period between the status check calls can have significant impact on the FDL operation performance. Shorter calling intervals result in better FDL performance, but also increase the CPU load by the FDL. Due to this trade off, a general advice for the calling interval cannot be given. It needs to be analysed and tailored individually for each target application.
- Calling `R_FDL_Handler` in the idle task: If it is ensured that the idle task is called often enough, this method might result in a good FDL performance, as the handler can be called continuously. However, this approach is not deterministic in case of a high CPU load by the application itself.

Due to the individual requirements of each application, a general advice for selecting a strategy to call the `R_FDL_Handler` cannot be given. Please also consider that mixtures of the above mentioned approaches can be meaningful depending on the target scenario.

Note:

When evaluating concepts for calling the `R_FDL_Handler`, please be aware that all FDL functions are not re-entrant. That means it is not allowed to call an FDL function from interrupt service routines while another FDL function is already running.

# Chapter 6 Cautions

Before starting the development of an application using the FDL, please carefully read and understand the following cautions:

1. CPU operating frequency configuration:

   Correct frequency configuration is essential for Flash programming quality and stability. Wrong configuration could lead to loss of data retention or Flash operation fail.

   The limits for CPU frequency are device dependent. Please consult Device Manual for correct range.

   If the CPU frequency is a fractional value, round up the value to the nearest integer.

   Do not change power mode (voltage or CPU clock) while FDL is performing a Data Flash operation. If power mode must change the user can:

   - put current operation into stand-by mode and wait until hardware conditions are restored
   - wait until operations are no longer busy or
   - reinitialize the library with proper CPU frequency value

2. CPU mode:

   The initialization function `R_FDL_Init` must be executed in CPU supervisor mode (register bit PSW.UM = 0).

3. Function re-entrancy:

   All functions are not re-entrant. So, re-entrant calls of any FDL function must be avoided.

4. Task switch, context change, synchronization between functions:

   Each function depends on global available information and is able to modify this information. In order to avoid synchronization problems, it is necessary that at any time only one FDL function is executed. So, it is not allowed to start an FDL function, then switch to another task context and execute another FDL function while the last one has not finished.

5. Entering power save (stand-by) mode:

   Entering power save mode is not allowed at all during on-going Data Flash operations. Use `R_FDL_StandBy` or wait until operations are no longer busy.

6. Different power save (stand-by) modes:

   Other power save modes than HALT will result in Flash hardware internal data loss. So, don't enter power save modes except HALT when further FDL operations are intended after wake-up. If entering other modes, the FDL need to be re-initialized by `R_FDL_Init`.

7. Initialization:

   The FDL library initialization by means of calling `R_FDL_Init` must be performed before calling most of the library functions. Exception is `R_FDL_GetVersionString` function that can be called anytime.

8. Critical Section handling:

   The `R_FDL_Init` function temporarily disables Code Flash. During this time, since the Code Flash is not available, the library is executing code from internal RAM (allocated space on stack). Please ensure that:

   - Code execution is done from other locations (e.g. internal RAM).
   - No access to Code Flash is allowed, e.g. by jump to interrupt/exception functions, direct Code Flash Read/Execution from the CPU, DMA accesses to Code Flash. The user can configure the provided callback macro functions in `fdl_cfg.h`. , in order to handle e.g.

interrupt & exception disable, DMA,... .The sample application provides examples on how to disable and restore interrupts and exceptions using the callback routines.

9. Interrupted flash operations:

   In case of Flash modification operation (Erase / Write) interruption, the electrical conditions of the affected Flash range (Flash block on erase, Flash write unit on Write) get undefined. It is impossible to give a statement on the read value after the interruption. Furthermore, the resulting read value is not reliable; the electrical margin for the specified data retention may not be given. In such case, erase and re-write the affected Flash block(s) to ensure data integrity and retention.

10. Write operation:

    Before executing a write operation, please make sure the given address range is erased.

11. Reading Data Flash:

    Data Flash on RH850 devices is made with differential cells for storage. This means that reading erased Data Flash areas directly (bypassing FDL) will produce undefined data with a tendency to the previously written data and it will most probably cause ECC error exceptions. To avoid this exceptions use `R_FDL_CMD_READ` command.

    DMA transfers from Data Flash are permitted, but need to be synchronized with the FDL.

    During command execution Data Flash is not available. Any direct read during command execution will result in invalid data therefore it must be avoided.

12. Dual operation:

    It is not possible to modify the Code Flash in parallel to a modification of the Data Flash or vice versa due to shared hardware resources.

13. Reusing the request command:

    Do not change the content of the request structure while an FDL command is operating because the library may crash or data loss can occur. Multiple requests, each using different request structures, do not have these adverse effects.

14. Workload and supervision:

    It is recommended to supervise the FDL operations and functions execution by timeout supervision (e.g. timer, counter, watchdog, etc.). In addition, the user of the library should evaluate the time necessary to perform a certain operation and divide long lasting operations to meet real-time system specifications.

15. Suspend and Stand-by nesting:

    It is not always possible to nest suspend and/or stand-by. E.g.:

    - Any operation ► suspend ► suspend – is not possible.
    - Any operation ► stand-by ► stand-by – is not possible.
    - Any operation ► stand-by ► suspend – is not possible.
    - Write or Erase ► suspend ► Erase operation – is not possible
    - Any operation ► suspend ► other operation ► suspend – is not possible
    - Write operation ► suspend ► other Write operation – is not possible

    It is recommended to avoid nesting as much as possible.

16. Stand-by:

    Do not continue FDL functions execution or start execution of any other function than `R_FDL_GetVersionString`, `R_FDL_WakeUp` or `R_FDL_Init` when the library is in stand-by mode.

17. Data alignment:

    Data Flash blocks are aligned to 64 bytes and Data Flash words are aligned to 4 bytes.

RH850 devices also add alignment restrictions for types larger than 8 bits. Please consult device hardware manual for details.

18. Precompile options

    The user must not use any pre-compile configuration options that are not documented in present manual.

19. Supported devices

    The RH850 FDL library is supported on 3 device families at the moment of writing of this manual. These families are E1x, F1x, P1x and R1x (where ‘x’ can be any letter depending on power consumption, peripherals, etc).

    Further device families may be added in the future.

# Revision History

| Chapter | Page | Description |
|---|---|---|
| | | Rev. 1.03:<br>Initial released document version |

Data Flash Access Library

Renesas Electronics Corporation

R01US0079ED0103