User's Manual

# Data Flash Access Library

## FDL - T05

---

## Data Flash Access Library for V850 Single Voltage Flash Devices

Document No. U20279EE1V4UM00
Date Published: 10.07.2013

To our customers,

## Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

# Legal Notes

**The information in this document is current as of July 2013. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**

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

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

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

(Note)

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

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

# Table of Contents

# Chapter 1 Introduction

This user's manual describes the internal structure, the functionality and software interfaces (API) of the NEC V850 Data Flash Access Library (FDL) type T05, designed for V850 Flash devices with Data Flash based on the UX6LF Flash technology

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

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

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

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

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

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

Caution:

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

The Data Flash Access Libraries together with the EEPROM emulation libraries, application samples, this application note and other device dependent information can be downloaded from the following URL:

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

## 1.1 Naming Conventions

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

These abbreviations shall be explained here:

| Abbreviations / Acronyms | Description |
|---|---|
| Block | Smallest erasable unit of a flash macro |
| Code Flash | Embedded Flash where the application code is stored. For devices without Data Flash EEPROM emulation might be implemented on that flash in the so called data area. |
| Data Flash | Embedded Flash where mainly the data of the EEPROM emulation are stored. Beside that also code operation might be possible. |
| Dual Operation | Dual operation is the capability to fetch code during reprogramming of the flash memory. Current limitation is that dual operation is only available between different flash macros. Within the same flash macro it is not possible! |
| EEL | EEPROM Emulation Library |
| 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. |
| FAL | Flash Access Library (Flash access layer) |
| FCL | Code Flash Library (Code Flash access layer) |
| FDL | Data Flash Library (Data Flash access layer) |
| Flash | “Flash EPROM” - Electrically erasable and programmable nonvolatile memory. The difference to ROM is, that this type of memory can be re-programmed several times. |
| Flash Block | A flash block is the smallest erasable unit of the flash memory. |
| Flash Macro | A flash comprises of the cell array, the sense amplifier and the charge pump (CP). For address decoding and access some additional logic is needed. |
| NVM | Non volatile memory. All memories that hold the value, even when the power is cut off. E.g. Flash memory, EEPROM, MRAM... |
| RAM | “Random access memory” - volatile memory with random access |
| ROM | “Read only memory” - nonvolatile memory. The content of that memory can not be changed. |
| Serial programming | The onboard programming mode is used to program the device with an external programmer tool. |
| Single Voltage | For the reprogramming of single voltage flashes the voltage needed for erasing and programming are generated onboard of the microcontroller. No external voltage needed like for dual- voltage flash types. |

# Chapter 2 UX6LF Data Flash

## 2.1 33-bit Implementation

The Data Flash of devices in UX6LF Flash technology is based on a standard 32-bit architecture. This means, that the data can be written and read in 32-bit units (read or write in 8-bit or 16-bit units is not possible!).

Additionally to every 32-bit data word a 33rd bit (Tag) is available for free usage.

While the 32 data bits can be read in a linear address room, the Tag can be read in another linear address room on a different address (every 32-bit address one tag can be read). The data address room starts from 0x02000000 while the Tag address room starts from 0x02100000

Furthermore, the Tag can be written independently from the other data and it is protected against bit failures separately. The FDL provides separate functions to write the data and the tags.

The Tags are completely in the hand of the user application. In the EEL concept, the Tag is used to write additional management data in order to ensure data consistency in case of write interruptions.

## 2.2 Dual operation

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

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

A standard architectural approach is the separation of the Flash into Code Flash and Data Flash. By that, it is possible to read from the Code Flash (to execute program code or read data) while Data Flash is modified, and vice versa. This allows implementation of EEPROM emulation concepts with Data storage on Data Flash while all program code is executed from Code Flash.

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

**Note:**

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

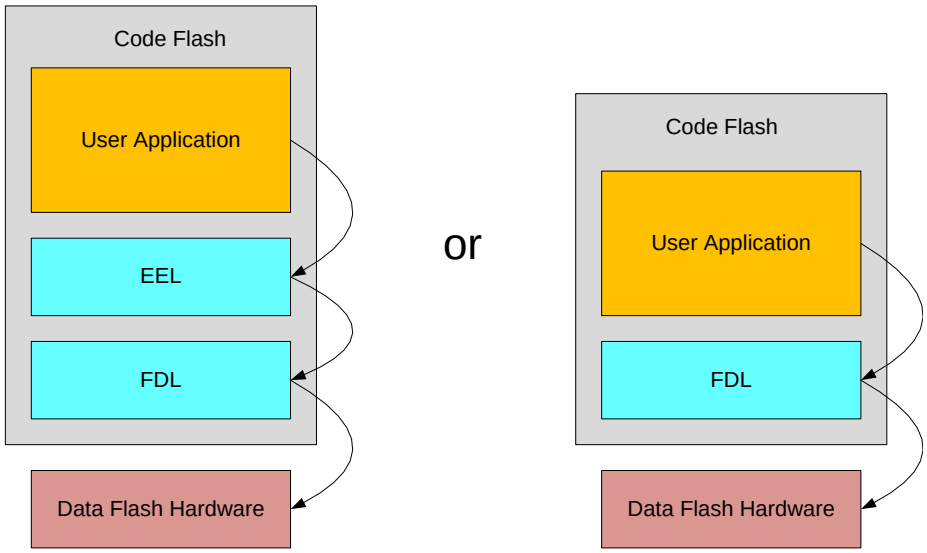
# Chapter 3 FDL Architecture

## 3.1 Layered Architecture

This chapter describes the function of all blocks belonging to the EEPROM Emulation and the Data Flash Access System.

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** Usage examples of the library layers




Application

The functional block “Application” should not use the functions offered by the FDL directly, in fact it is recommended to access the EEL API only.

Nevertheless, if the user intends to implement a proprietary EEPROM emulation, he may use the FDL functions for direct Data Flash accesses. Even combinations of both are possible, always considering the synchronization of these access paths.

EEPROM Emulation Library (EEL)

The functional block “EEPROM Emulation library” offers all functions and commands the “Application” can use in order to handle its own EEPROM data.

Data Flash Library (FDL)

The “Data Flash Library” offers an access interface to any user-defined Data Flash area, so called “FDL-pool” (described in next chapter). Beside the initialization function the FDL allows the execution of access-commands like write as well as a suspend-able erase command.

Note:
General requirement is to be able to deliver pre-compiled EEL libraries, which can be linked to either Data Flash libraries (FDL) or Code Flash libraries (FCL). To support this, a unique API towards the EEL must be provided by these libraries. Following that, the standard API prefix FDL_... which would usually be provided by the FDL library, is replaced by a standard Flash Access Layer prefix FAL_...

All functions, type definitions, enumerations etc. will be prefixed by FAL_ or fal_.
Independent from the API, the module names will be prefixed with FLD_ in order to distinguish the source/object modules for Code and Data Flash.

## 3.2 Data Flash Pools

The FDL pool defines the Flash blocks, which may be accessed by any FDL operation (e.g. write, erase). 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 (function FAL_Init), while usually the complete Data Flash is selected.

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”.

Pools details:

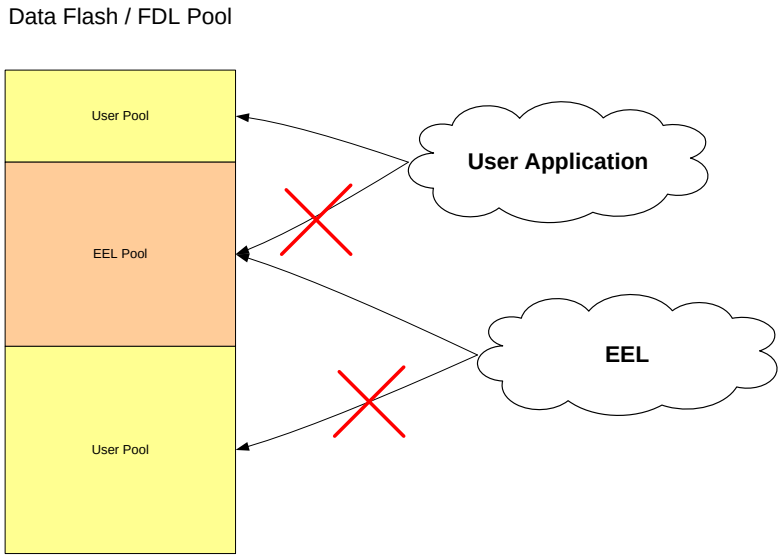
- **FDL-pool** is just a place holder for the EEL-pool. It does not allocate any flash memory. The FDL-pool descriptor defines the valid address space for FDL access to protect all flash outside the FDL-pool against destructive access (write/erase) by a simple address check in the library.

  To simplify function parameter passing between FDL and the higher layer the physical Flash addresses (e.g. 0x02000000….0x0200FFFF) are transformed into a linear address room 0x0000….0xFFFF used by the FDL.

- **EEL-pool** allocates and formats (virgin initialization) all flash blocks belonging to the EEL-pool. The header data are generated in proper way to be directly usable by the application.

- **User Pool** is completely in the hands of the user application. It can be used to build up an own user EEPROM emulation or to simply store constants.

**Figure 2 Data Flash / FDL Pool**



## 3.3 Safety Considerations

EEPROM emulation in the automotive market is not only operated under normal conditions, where stable function execution can be guaranteed. In fact, several failure scenarios should be considered.

Most important issue to be considered is the interruption of a function e.g. by power fail or Reset.

Differing from a normal digital system, where the operation is re-started from a defined entry point (e.g. Reset vector), the EEPROM emulation modifies Flash cells, which is an analogue process with permanent impact on the cells. Such an interruption may lead to instable electrical cell conditions of affected cells. This might be visible by undefined read values (read value != write value), but also to defined read values (blank or read value = write value). In each case the read margin of these cells is not given. The value may change by time into any direction.

This needs to be considered in any proprietary EEPROM emulation or simple data storage concept.

## 3.4 Bit error checks

Independent from the Flash manufacturer or Flash technology, Bit errors in the Flash (independent if occurring in data or Tags) might be caused by different conditions. Different measures are implemented or provided in order to handle such problems.

While device dependant causes like hardware defects or weak Flash cells are completely covered by the NEC qualification and production quality and by Flash ECC (Error correction code), one major issue need to considered additionally.

Interruption of Flash erase or write operations e.g. by power fails or Resets result in not completely charged or discharged Flash cells which results in Flash data without sufficient data retention. This need to be prevented by the operation conditions of the device or need to be detected by the software in order to ensure stable data storage conditions.

While prevention is often not possible, detection can be done by different mechanisms like checksums or special write sequences where one written word ensures that previous data write was completed successfully.

After having considered the mechanisms above, one method to additionally increase the system robustness is the check for bit errors in written data. This method assumes that multiple bit errors (by not completely charged/discharged Flash cells) don’t occur at once but by time. By special correction bits, the NEC Data Flash hardware can correct single/double bit errors in a 32bit data word (+correction bits) or a single bit error in a Tag (+correction bits) during run-time. Furthermore, it can signal this error to the application. By that, the user application can set-up a mechanism to refresh the data with the single bit error right on time before a multi bit error can occur that destroys the data.

For that purpose, the FDL provides a function to check a certain Flash address for bit errors on the data word + Tag.

It is recommended to cyclically execute the bit error check over the complete data range.

# Chapter 4 Implementation

## 4.1 File structure

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

The application sample initializes the *EEL* and does some dummy data set *Write* and *Read* operations.

Differing from former EEPROM emulation libraries, this one is realized not as a graphical IDE related specific sample project, but as a standard sample project which is controlled by makefiles.

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

The FDL and EEL files are strictly separated, so that the FDL can be used without the EEL. However, using EEL without FDL is not possible.

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

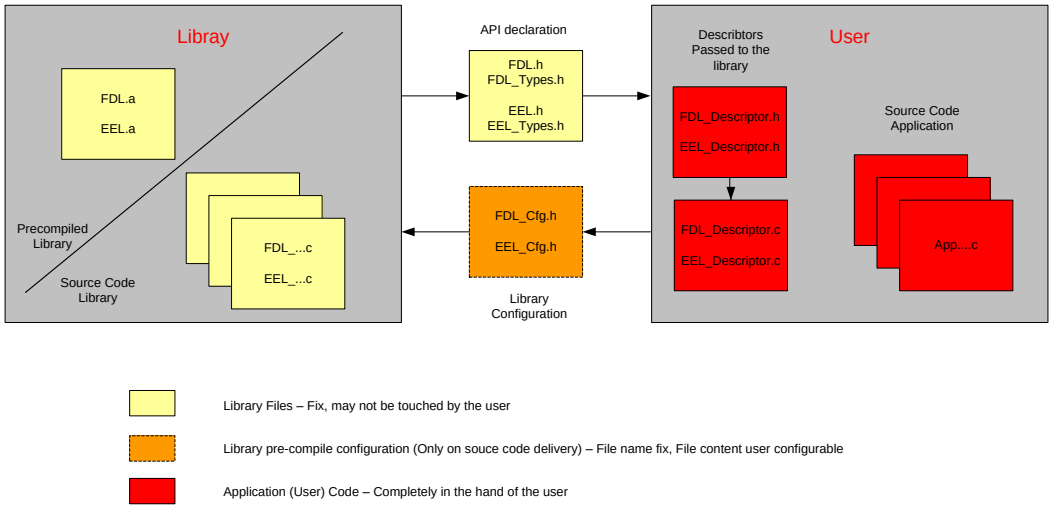
**Note:**

The application sample does not contain sample code for the FDL interface usage, but only for the EEL interface. Anyhow, as the EEL contains FDL functions calls, the usage of the FDL functions can be derived from that.

### 4.1.1 Overview

The following picture contains the library and application related files.

**Figure 3 Library and application file structure**




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

The file FDL/EEL.h is the library interface functions header file.

The file FDL/EEL_Types.h is the library interface parameters and types header file.

In case of source code delivery, the library must be configured for compilation. The file FDL/EEL_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.

FDL/EEL_Descriptor.c and FDL/EEL_Descriptor.h do not belong to the libraries themselves, but to the user application. These files reflect an example, how the library descriptor ROM variables can be built up which need to be passed with the functions FDL/EEL_Init to the FDL/EEL for run-time configuration (see chapter 5.2, “Run-time configuration” and 5.4.1.1, “FAL_Init”).

- The structure of the descriptor is passed to the user application by FDL/EEL_Types.h.
- The value definition should be done in the file FDL/EEL_Descriptor.h.
- The constant variable definition and value assignment should be done in the file FDL/EEL_Descriptor.c.

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

### 4.1.2 Delivery package directory structure and files

| | |
|---|---|
| [root] | |
| Release.txt | Installer package release notes |
| | |
| [root]\[make] | |
| GNUPublicLicense.txt | Make utility license file |
| libiconv2.dll | DLL-File required by make.exe |
| libintl3.dll | DLL-File required by make.exe |
| make.exe | Make utility |
| | |
| [root]\[<device name>]\[compiler] | |
| Build.bat | Batch file to build the application sample |
| Clean.bat | Batch file to clean the application sample |
| Makefile | Makefile that controls the build and clean process |
| | |
| [root]\[<device name>]\[<compiler>]\[sample] | |
| EELApp.h | Application sample header with function prototypes and collecting all includes |
| EELApp_Main.c | Main source code |
| EELApp_Control.c | Source code of the control program for EEEPROM emulation |
| target.h | Target device and application related definitions |
| ... device header files ... | ( GHS: df<device number>.h, io_macros.h, ... |
| | IAR: io_70f3xxx.h |
| | NEC: - ) |
| ... startup file ... | ( GHS: Startup_df<dev. num.>.850 |

| | | |
|---|---|---|
| | IAR: | DF3xxx_HWInit.s85 |
| | NEC: | tbd ) |
| ... linker directive file ... | ( GHS: | Df<device number>.ld |
| | IAR: | lnk70f3xxx.xcl |
| | NEC: | tbd ) |

[root]\[<device name>]\[<compiler>]\[sample]\[EEL]

| | |
|---|---|
| EEL_Cfg.h | Header file with definitions for library setup at compile time |
| EEL.h | Header file containing function prototypes |
| EEL_Types.h | Header file containing calling structures and error definitions |
| EEL_Descriptor.h | Descriptor file header with the run-time EEL configuration. To be edited by the user. |
| EEL_Descriptor.c | Descriptor file with the run-time EEL configuration. Should not be edited by the user. |

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

| | |
|---|---|
| EEL_Global.h | Library internal defines, function prototypes and variables |
| EEL_UserIF.c | Source code for the *EEL* internal state machine, service routines and initialization |
| EEL_BasicFct.c | Source code of functions called by the state machine |

[root]\[<device name>]\[<compiler>]\[sample]\[FDL]

| | |
|---|---|
| FDL_Cfg.h | Header file with definitions for library setup at compile time |
| FDL.h | Header file containing function prototypes |
| FDL_Types.h | Header file containing calling structures and error definitions |
| FDL_Descriptor.h | Descriptor file header with the run-time FDL configuration. To be edited by the user. |
| FDL_Descriptor.c | Descriptorfile with the run-time EEL configuration. Should not be edited by the user. |

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

| | |
|---|---|
| FDL_Env.h | Library internal defines for the Flash programming hardware |
| FDL_Global.h | Library internal defines, function prototypes and variables |
| FDL_UserIF.c | Source code for the library user interface and service functions |
| FDL_HWAccess.c | Source code for the libraries HW interface |

## 4.2 EEL Linker sections

The following sections are EEPROM emulation library related:

- FAL_Text
  FDL code section, containing the hardware interface and user interface.
- FAL_Data
  FDL Data section containing all FDL internal variables.
- EEL_Text
  EEL code section containing the state machine, user interface and FAL interface.
- FAL_Data
  FDL Data section containing all EEL internal variables.

## 4.3 MISRA Compliance

The EEL and FDL have been tested regarding MISRA compliance.

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

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

# Chapter 5 User Interface (API)

## 5.1 Pre-compile configuration

The pre-compile configuration of the FDL may be located in the FDL_cfg.h.

Based on the library design, there are no library concept related configurations to be done. Anyhow, depending on device differences, there may be FDL internal device dependant adaptations necessary. These adaptations are controlled by device dependant defines in this file.

Take care to follow the configurations done in the sample application in order to ensure correct FDL operation.

**The configuration elements sample:**

*#define FDL_DEVICESPECIFIC_CFG_3501*
By setting this define, umbrella chip specific adaptations of the FDL are activated.

**Implementation in FDL_Cfg.h (define not set):**

```
// #define FDL_DEVICESPECIFIC_CFG_3501
```

## 5.2 Run-time configuration

The overall EEL run-time configuration is defined by an EEL specific part (EEL run-time configuration) and by the FDL run-time configuration. Background of the splitting is that the FDL requires either common, by EEL and FDL used information (e.g. block size) or EEL related information (e.g. about the EEL pool size). So, this information is part of the FDL run-time configuration.

Both configurations of FDL and EEL are stored in descriptor structures which are declared in FDL_Types.h / EEL_Types.h and defined in FDL_Descriptor.c / EEL_Descriptor.c with header files FDL_Descriptor.h / EEL_Descriptor.h. The descriptor files (.c and .h) are considered as part of the user application.

The defined descriptor structures are passed to the libraries as reference by the functions FDL_Init and EEL_Init.

### 5.2.1 FDL run-time configuration elements

The descriptor contains the following elements; please also refer to chapter 3.2”Data Flash Pools”:

*blkSize:*
Defines the Data Flash block size in Bytes. This is just a configuration option reserved for future use. In all current Devices the Data Flash size is fixed to 2kB=0x800Bytes.

Value range: Currently fixed to 0x800

*falPoolSize:*
Defines the number of blocks used for the FAL pool, which means the User Pool + EEL Pool. Usually, the FAL pool size equals the total number of Flash blocks.

Value range: Min: EEL pool size
Max: Physical number of Data Flash blocks

*eelPoolStart:*
Defines the first Data Flash block number used as EEL pool.

Value range: Min: FAL Pool start block
Max: *eelPoolStart* + *eelPoolSize* <= *falPoolSize*

*eelPoolSize:*
Defines the number of blocks used for the EEL pool.

Value range: Min: 4 Blocks (required for proper EEL operation)
Max: FAL pool size, condition:
*eelPoolStart* + *eelPoolSize* <= *falPoolSize*

**Implementation:**

The descriptor structure is defined in the module FDL_Types.h

```
typedef struct {
                    fal_u16 blkSize_u16;
                    fal_u16 falPoolSize_u16;
                    fal_u16 eelPoolStart_u16;
                    fal_u16 eelPoolSize_u16;
                } fal_descriptor_t;
```

The descriptor variable definition and filling is part of the user application. The files FDL_Descriptor.h/.c give an example which shall be used by the user application. Only FDL_Descriptor.h need to be modified for proper configuration while FDL_Descriptor.c can be kept unchanged.

Example variable definition and filling in FDL_Descriptor.c:

```
const fal_descriptor_t  eelApp_fdlConfig =
                        {
                            FAL_CONFIG_BLOCK_SIZE,
                            FAL_CONFIG_DATAFLASH_SIZE,
                            EEL_CONFIG_BLOCK_START,
                            EEL_CONFIG_BLOCK_CNT
                        };
```

Example configuration in FDL_Descriptor.h:

Example 1)

Data Flash size is 32kB, separated into blocks of 2kB.
The EEL shall use the complete Data Flash for the EEL pool:

```
    #define FAL_CONFIG_DATAFLASH_SIZE               16
    #define FAL_CONFIG_BLOCK_SIZE                   0x800
    #define EEL_CONFIG_BLOCK_START                  0
    #define EEL_CONFIG_BLOCK_CNT                    16
```

Example 2)

Data Flash size is 32kB, separated into blocks of 2kB.
The EEL shall use blocks 2 to 11 for the EEL pool, while blocks 0 to 1 and 12 to 15 can be used as user pool:

```
    #define FAL_CONFIG_DATAFLASH_SIZE               16
    #define FAL_CONFIG_BLOCK_SIZE                   0x800
    #define EEL_CONFIG_BLOCK_START                  2
    #define EEL_CONFIG_BLOCK_CNT                    10
```

Example 3)

Data Flash size is 32kB, separated into blocks of 2kB; the EEL shall not be used at all. The complete Data Flash shall be used as user pool:

```
    #define FAL_CONFIG_DATAFLASH_SIZE               16
    #define FAL_CONFIG_BLOCK_SIZE                   0x800
    #define EEL_CONFIG_BLOCK_START                  0
    #define EEL_CONFIG_BLOCK_CNT                    0
```

## 5.3 Data Types

### 5.3.1 Error Codes

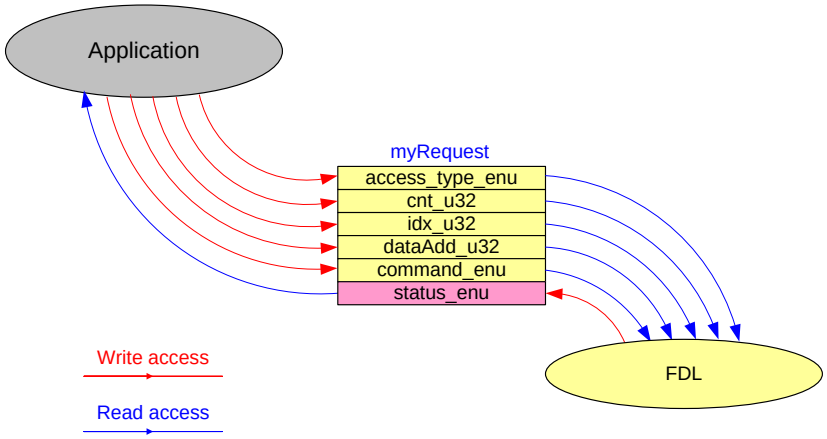
**Figure 4 FDL status & error codes**

| | Explanation | Root Cause Judgment | FAL Operation Impact | Recommended Application Reaction |
|---|---|---|---|---|
| FAL_OK | The operation finished successfully | Normal library behavior | None | None |
| FAL_BUSY | The operation has been started successfully | Normal library behavior | None | Continue operation |
| FAL_ERASE_SUSPENDED | An ongoing Flash erase operation is suspended by user application request | Normal library behavior | None | Continue operation, resume erase operation soon |
| FAL_ERR_PARAMETER | Wrong parameters have been passed to the FAL, e.g.: Wrong parameter in the request structure | Application bug | Current command is rejected | Refrain from further Flash operations and investigate in the root cause |
| FAL_ERR_PROTECTION | An erase or write operation to protected Flash blocks should be executed or the access rights variable was not set appropriately | Application bug | Current command is rejected | Refrain from further Flash operations and investigate in the root cause |
| FAL_ERR_REJECTED | A new operation should be initiated although the state machine is still busy with a preceding operation | Application bug or intended behavior | Current command rejected | Repeat the command when the preceding operation has finished. If not intended behavior, investigate in the root cause |
| FAL_ERR_WRITE | A flash word write or Tag write failed | Possible root causes:<br>- Flash word was not blank (application bug)<br>- Flash word or the complete Flash defect (Hardware defect) | The operation could not be finished successfully. | - If the Flash word was blank, the Flash word respectively the Data Flash should be considered as defect<br>- If the Flash word was not blank, erase the Flash block and re-write the word. An application bug should be considered. Investigate in the root cause |
| FAL_ERR_ERASE | A Flash Erase failed | Flash block or complete Data Flash defect (Hardware defect) | The operation could not be finished successfully. | The Flash block respectively the complete Data Flash should be considered as defect |
| FAL_ERR_COMMAND | The command to be executed is unknown | Application bug | Current command rejected | Refrain from further Flash operations and investigate in the root cause |
| FAL_ERR_BITCHECK | The bit check operation found a bit error in the data word or the Tag | At least on bit in the investigated data is wrong.<br>Possible cause:<br>- Not completely erased or written Flash, e.g. caused by a power fail during a Flash operation<br>- long time frame between Flash Erase and bit error check | None | Depending on the usage scenario:<br>1) If it is ensured that the erase and write operations on the checked data were completed successfully and the data retention time is not exceeded, this is a normal behavior. Up to two bits may fail as the data is ECC corrected<br>2) If above condition 1) is not fulfilled, but by cyclical checks it is ensured that only up to 2 bits are failed (we can assume that bits fail by time, not all at once), refresh the data into a new fresh erased block.<br>3) If neither 1) nor 2) are fulfilled, we cannot trust the correctness of the data. Reaction is then up to the application |
| FAL_ERR_INTERNAL | A library internal error occurred, which could not happen in case of normal application execution | - Application bug (e.g. Program run-away, destroyed program counter ...)<br>- Programming hardware problem | The operation could not be finished successfully. | Refrain from further Flash operations and investigate in the root cause |

### 5.3.2 User operation request structure

All different user operations are initiated by a central initiation function (FAL_Execute). All information required for the execution is passed to the FAL by a central request structure. Also the error is returned by the same structure:

**Figure 5 Request structure handling**




The following request elements are defined:

- command_enu: User command to execute:
  - EEL_CMD_ERASE — Erase a Flash block
  - EEL_CMD_WRITE — Write data words to Flash
  - EEL_CMD_WRITE_TAG — Write Tags in Flash
  - EEL_CMD_BITCHECK — Checks data and Tag bits on a certain address
- dataAdd_u32: Only required for Write command:
  Address of the write buffer of the application
- idx_u32: Write / Write Tag:
  destination word index
  Erase:
  block index of the 1st block to erase
- cnt_u32: Write / Write Tag:
  Number of words to write
  Erase:
  Number of blocks to erase
- status_enu: Status/Error codes returned by the library (see previous page)
- access_type_enu: Access right definition:
  - FDL_ACCESS_USER — Access the user pool
  - FDL_ACCESS_EEL — Access the EEL pool

Note: In order to initiate a Flash operation, the access right to the Flash must be set. The user application may only access the complete configured Data Flash range except the one configured for the EEL. The EEL may only access its range. The ranges are defined in the FAL descriptor, passed to the FAL_Init function. The access right is reset after each Flash operation. If not set again on calling EEL_Execute, this function will return a protection error.

**Type definition in FAL_Types.h**

```
typedef enum   {
                    FAL_CMD_ERASE,
                    FAL_CMD_WRITE,
                    FAL_CMD_WRITE_TAG,
                    FAL_CMD_BITCHECK
               } fal_command_t;
```

```
typedef enum   {
                    FAL_OK,
                    FAL_BUSY,
                    FAL_ERASE_SUSPENDED,
                    FAL_ERR_PARAMETER,
                    FAL_ERR_PROTECTION,
                    FAL_ERR_REJECTED,
                    FAL_ERR_WRITE,
                    FAL_ERR_ERASE,
                    FAL_ERR_COMMAND,
                    FAL_ERR_BITCHECK,
                    FAL_ERR_INTERNAL
               } fal_status_t;
```

```
typedef struct  {
                     fal_command_t       command_enu;
                     fal_u32             dataAdd_u32;
                     fal_u32             idx_u32;
                     fal_u16             cnt_u16;
                     fal_access_type_t   accessType_enu;
                     fal_status_t        status_enu;
                } fal_request_t;
```

## 5.4 FAL Functions

Functions represent the functional interface to the FAL which other SW can use.

### 5.4.1 Initialization / Shut down

#### 5.4.1.1 FAL_Init

**Description**

The *FAL_Init()* function is executed before any execution of FDL Flash operations.

The main purpose of the *FAL_Init()* is:

- Initializing the Flash access protection of the EEPROM emulation
- Initialization of internal FDL variables

**Interface**

```
fal_status_t FAL_Init( const fal_descriptor_t *fal_config_pstr)
```

**Arguments**

| Type | Argument | Description |
|---|---|---|
| fal_descriptor_t | fal_config_pstr | Pointer to the FDL run-time configuration descriptor structure in ROM |

**Return types/values**

| Type | Argument | Description |
|---|---|---|
| eel_status_t | - | Result of the function. Possible values are:<br>EEL_OK<br>EEL_ERR_PARAMETER |

The function checks the configuration in the descriptor variable for consistency. If a problem is found in the configuration, the error EEL_ERR_PARAMETER is returned:

- Descriptor pointer must be != zero
- Block size must be != zero
- FAL pool size must be != zero
- EEL pool must fit into the FAL pool

On check fail, all further FAL operations are locked.

**Pre-conditions**

None

**Post-conditions**

None

**Example**

fal_rtConfiguration is configured globally in FAL_Descriptor.c

```
fal_status_t res_enu;

res_enu = FAL_Init( &falConfig_str );

if( FAL_OK != res_enu )
{
    /* FAL error handler */
    while( 1 )
        ;
}
```

### 5.4.2 Suspend / Resume

The library provides the functionality to suspend and resume the library operation in order to provide the possibility to synchronize the EEL Flash operations with possible user application Flash operations, e.g. write/erase by using the FDL library directly or read by direct Data Flash read access.

#### 5.4.2.1 FAL_EraseSuspendRequest

**Description**

This function requests the Erase suspend in order to be able to do other Flash operations

**Interface**

```
fal_status_t FAL_EraseSuspendRequest( void );
```

**Arguments**

None

**Return types/values**

| Type | Argument | Description |
|---|---|---|
| fal_status_t | - | Result of the function. Possible values are:<br>EEL_OK<br>EEL_ERR_REJECTED |

**Pre-conditions**

- An erase operation must have been started.
- The started erase operation may not have been finished (request structure status value is FAL_BUSY).
- The library may not already be suspended.

On violation of any of the above conditions, the function will return EEL_ERR_REJECTED.

**Post-conditions**

- Call FAL_Handler until the library is suspended (status FAL_ERASE_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 FAL_EraseResume.

**Example**

```
fal_status_t  srRes_enu;
fal_request_t myReq_str_str;
fal_u32       i;

/* Start Erase operation */
myReq_str_str.command_enu     = FAL_CMD_ERASE;
myReq_str_str.idx_u32         = 0;
myReq_str_str.cnt_u16         = 4;
myReq_str_str.accessType_enu  = FAL_ACCESS_USER;

FAL_Execute( &myReq_str_str );

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

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

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

while( FAL_ERASE_SUSPENDED != myReq_str_str.status_enu )
{
    FAL_Handler();
}


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


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

if( FAL_OK != srRes_enu )
{
    /* Error handler */
    while( 1 )
        ;
}

/* Finish the erase */
while( myReq_str_str.status_enu == EEL_BUSY )
{
    FAL_Handler();
}

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

#### 5.4.2.2 FAL_EraseResume

**Description**

This function resumes the FAL operations after suspend

**Interface**

```
fal_status_t FAL_EraseResume( void );
```

**Arguments**

None

**Return types/values**

| Type | Argument | Description |
|---|---|---|
| fal_status_t | - | Result of the function. Possible values are:<br>FAL_OK<br>FAL_ERR_REJECTED |

**Pre-conditions**

- The library must be suspended. Call FAL_SuspendRequest before and wait until the suspend process finished.

On violation the function ends with FAL_ERR_REJECTED.

**Post-conditions**

None

**Example**

See FAL_Suspend

### 5.4.3 Stand-By / Wake-Up

The device system architecture prevents entering a device power safe mode, when a Flash operation is ongoing. By that, especially Flash Erase can delay a power safe mode significantly (several 10th ms). In order to allow fast entering of such mode, the functions FAL_StandBy and FAL_WakeUp have been introduced, which suspend a possibly ongoing Flash Erase operation (FAL_StandBy) and resume it after waking up from power safe mode (FAL_WakeUp). Any other Flash operation (e.g. Write) is not suspended as the execution time is considerably short.

FAL_StandBy immediately suspends a possible ongoing Flash Erase asynchronously to other FAL operations. So, it is mandatory to call FAL_WakeUp before entering normal FAL operation again. The prescribed sequence in case of using FAL_StandBy/WakeUp is:

→ any FAL operation
→ FAL_StandBy
→ device power safe
→ device wake-up
→ FAL_WakeUp
→ continue FAL operations

**Note: Please consider not to enter a power safe mode which resets the Flash hardware (e.g. Deep Stop mode), because a resume of the previous operation is not possible afterwards. The library is not able to detect this failure.**

**Note: When FAL_EraseSuspendRequest has already suspended a Flash Erase, another Erase suspend by FAL_StandBy is not possible. So, the following sequence is not allowed:**
**Erase → suspend → Erase(another block) → Stand-by**
**This applies independantly, if the 1st erase was issued by EEL or user application**

#### 5.4.3.1 FAL_StandBy

**Description**

This function suspends a possibly ongoing Flash Erase. Any other Flash operation is untouched

**Interface**

```
fal_status_t FAL_StandBy( void );
```

**Arguments**

None

**Return types/values**

| Type | Argument | Description |
|---|---|---|
| fal_status_t | - | Result of the function. Possible values are:<br>EEL_OK<br>EEL_ERR_REJECTED |

**Pre-conditions**

- The library must be initialized
- The sequence
  Flash Erase --> FAL suspend --> Flash Erase --> FAL Stand-by
  is not allowed
- The FAL is not in stand-by mode

On violation of any of the above conditions, the function will return EEL_ERR_REJECTED.

**Post-conditions**

- Execute FAL_WakeUp as next FAL function

**Example**

```
fal_status_t   fdlRet_enu;
fal_request_t myReq_str_str;

/* Start Erase operation */
myReq_str_str.command_enu     = FAL_CMD_ERASE;
myReq_str_str.idx_u32         = 0;
myReq_str_str.cnt_u16         = 4;
myReq_str_str.accessType_enu  = FAL_ACCESS_USER;

FAL_Execute( &myReq_str_str );

...

fdlRet = FAL_StandBy();
if( FAL_OK != fdlRet )
{
    /* error handler */
}

...
/* enter power safe mode */
...

fdlRet = FAL_WakeUp();
if( FAL_OK != fdlRet )
{
    /* error handler */
}


/* Finish the erase */
while( myReq_str_str.status_enu == EEL_BUSY )
{
    FAL_Handler();
}

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

#### 5.4.3.2 FAL_WakeUp

**Description**

This function wakes-up the library from stand-by.

**Interface**

```
fal_status_t FAL_WakeUp( void );
```

**Arguments**

None

**Return types/values**

| Type | Argument | Description |
|---|---|---|
| fal_status_t | - | Result of the function. Possible values are:<br>EEL_OK<br>EEL_ERR_REJECTED |

**Pre-conditions**

- The library must be initialized
- The library must be in stand-by mode

On violation of any of the above conditions, the function will return EEL_ERR_REJECTED.

**Post-conditions**

-

**Example**

See FAL_StandBy

### 5.4.4 Operational functions

#### 5.4.4.1 FAL_Execute

**Description**

The execute function initiates all Flash modification operations. The operation type and operation parameters are passed to the FAL 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.

Except for the “bit error check” command, 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 FAL_Handler (explained later on). The “bit error check” operation is executed by the FAL_Execute function alone. Further calls of FAL_Handler are not necessary.

Possible user commands are:

- Erase

  Erases a defined number of Flash blocks. The start block and the number of blocks can be defined.

  Required parameters from the request structure:

  - command_enu → FAL_CMD_ERASE for the Erase operation
  - idx_u32 → Start block index (block number)
  - cnt_u32 → Number of blocks to erase
  - access_type_enu → Access right, either FDL_ACCESS_USER or FDL_ACCESS_EEL

  The parameters are checked in EEL_Execute, resulting in EEL_ERR_PARAMETER error on violation. Independently in order to be robust against library external influences, the parameters are checked again by the access check functionality, then resulting in error EEL_ERR_PROTECTION.

  The check condition is:

  - The range (start block) to (Start block + Number of blocks - 1) must be in the EEL/User pool.

- Write

  Writing data from a user defined source buffer to a destination address in the Data Flash. The Tag bit aligned to the data is not affected.

  Required parameters from the request structure:

  - command_enu → FAL_CMD_WRITE for the Write operation
  - idx_u32 → Start byte index in the Data Flash (= relative address)
    The address is calculated relative to the Data Flash base address, e.g.:
    - 1st word of the Data Flash is addressed by 0x00000000
    - 3rd word of the Data Flash is addressed by 0x00000008

- cnt_u32 → Number of words to write (Word Count). Based on the Flash hardware implementation 1, 2, 3 or 4 words can be written
- dataAdd_u32 → Source data buffer address
- access_type_enu → Access right, either FDL_ACCESS_USER or FDL_ACCESS_EEL

The parameters are checked in EEL_Execute, resulting in EEL_ERR_PARAMETER error on violation. Independently in order to be robust against library external influences, the parameters are checked again by the access check functionality, then resulting in error EEL_ERR_PROTECTION.

The check condition is:

- The range (start word) to (Start word + Number of words - 1) is in the EEL/User pool.

- Write Tag

  Setting tag bits in the Data Flash to zero. The data words aligned to the tags are not affected.

  Required parameters from the request structure:

  - command_enu → FAL_CMD_WRITE_TAG for the Write Tag operation
  - idx_u32 → Start byte index in the Data Flash (= relative address)
    The address is calculated relative to the Data Flash base address, e.g.:
    - 1st tag of the Data Flash is addressed by 0x00000000
    - 3rd tag of the Data Flash is addressed by 0x00000008
  - cnt_u32 → Number of tags to set (Tag count). Based on the Flash hardware implementation 1, 2, 3 or 4 tags can be set
  - access_type_enu → Access right, either FDL_ACCESS_USER or FDL_ACCESS_EEL

  The parameters are checked in EEL_Execute, resulting in EEL_ERR_PARAMETER error on violation. Independently in order to be robust against library external influences, the parameters are checked again by the access check functionality, then resulting in error EEL_ERR_PROTECTION.

  The check condition is:

  - The range (start word) to (Start word + Number of words - 1) is in the EEL/User pool.

- Bit error check

  Check one Data Flash address for bit errors in the 32-bit data and the Tag.

  Required parameters from the request structure:

  - command_enu → FAL_CMD_BITCHECK for the Bit error check operation

- idx_u32 → Start byte index in the Data Flash (= relative address)
  The address is calculated relative to the Data Flash base address, e.g.:
  - 1st tag of the Data Flash is addressed by 0x00000000
  - 3rd tag of the Data Flash is addressed by 0x00000008
- access_type_enu → Access right, either FDL_ACCESS_USER or FDL_ACCESS_EEL

The parameters are checked in EEL_Execute, resulting in EEL_ERR_PARAMETER error on violation. Independently in order to be robust against library external influences, the parameters are checked again by the access check functionality, then resulting in error EEL_ERR_PROTECTION.

The check condition is:

- The relative address is in the EEL/User pool.

**Interface**

```
void FAL_Execute(fal_request_t *request_pstr);
```

**Arguments**

| Type | Argument | Description |
|---|---|---|
| fal_request_t | request_pstr | See chapter 5.3.2, “User operation request structure” |

**Return types/values**

| Type | Argument | Description |
|---|---|---|
| fal_request_t | request_str.<br>status_enu | The value is returned in the request structure error variable.<br><br>All commands:<br>FAL_ERR_REJECTED<br>FAL_ERR_PARAMETER<br>FAL_ERR_COMMAND<br>FAL_ERR_INTERNAL<br><br>All commands except bit error check:<br>FAL_BUSY<br><br>Only bit error check command:<br>FAL_ERR_BITCHECK<br>FAL_OK |

**Note:**
The user application can either react directly on the errors returned by the FAL_Execute function or call the handler function FAL_Handler and react on errors then. The errors set on FAL_Execute are not reset and the handler execution does not do additional operations in case of an error already set.

### Pre-conditions

- Call FAL_Init to initialize the library

### Post-conditions

- Call FAL_Handler to complete the initiated operation (“Except bit error check”)

### Example

Example erase blocks 0 to 3:

```
fal_request_t myReq_str;

myReq_str.command_enu    = FAL_CMD_ERASE;
myReq_str.idx_u32        = 0;
myReq_str.cnt_u16        = 4;
myReq_str.accessType_enu = FAL_ACCESS_USER;

FAL_Execute( &myReq_str );
while( myReq_str.status_enu == FAL_BUSY )
{
    FAL_Handler();
}

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

Example write Data to addresses 0x100 to 0x107

```
fal_request_t myReq_str;
fal_u32 data[] = { 0x12345678, 0x23456789 };

myReq_str.command_enu    = FAL_CMD_WRITE;
myReq_str.idx_u32        = 0x100;
myReq_str.cnt_u16        = 2;
myReq_str.dataAdd_u32    = (fal_u32)&data[0];
myReq_str.accessType_enu = FAL_ACCESS_USER;

FAL_Execute( &myReq_str );
while( myReq_str.status_enu == FAL_BUSY )
{
    FAL_Handler();
}

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

Example write Tags on addresses 0x100 to 0x107

```
fal_request_t myReq_str;

myReq_str.command_enu    = FAL_CMD_WRITE_TAG;
myReq_str.idx_u32        = 0x100;
myReq_str.cnt_u16        = 2;
myReq_str.accessType_enu = FAL_ACCESS_USER;

FAL_Execute( &myReq_str );
while( myReq_str.status_enu == FAL_BUSY )
{
    FAL_Handler();
}

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

Example check for a bit error on address 0x100

```
fal_request_t   myReq_str;

myReq_str.command_enu    = FAL_CMD_BITCHECK;
myReq_str.idx_u32        = 0x100;
myReq_str.accessType_enu = FAL_ACCESS_USER;

FAL_Execute( &myReq_str );

if( myReq_str.status_enu != FAL_OK )
{
    /* Bit error handling */
    while( 1 )
        ;
}
```

#### 5.4.4.2 FAL_Handler

**Description**

This function handles the command processing for the FAL Flash operations. After operation initiation by FAL_Execute, this function needs to be called frequently.

The function checks the operation status, handles library internal state machines and updates the request structure status_enu variable when the operation has finished. By that, the operation end can be polled.

**Note:**
FAL_Handler must be called until the Flash operation has finished in order to disable the Flash programming hardware. Only after deinitialization further operations can be started or Data Flash can be read.

**Interface**

```
void FAL_Handler( void );
```

**Arguments**

-

**Return types/values**

| Type | Argument | Description |
|---|---|---|
| fal_request_t | request_str. status_enu | The value is returned in the request structure error variable, passed to the FAL_Execute function. The possible return values depend on the operation that was started as well as on the errors of background operations.<br>This table describes not the errors set by operation invocation with the FAL_Execute function, but the errors, additionally set during operation execution. |
| | | All operations:<br>• FAL_OK<br>• FAL_BUSY<br>• FAL_ERR_INTERNAL<br>• FAL_ERR_PROTECTION |
| | | Additionally on Erase:<br>• FAL_ERR_ERASE |
| | | Additionally on Write & Write Tag:<br>• FAL_ERR_WRITE |
| | request_str. accessType_enu | Is reset to FAL_ACCESS_NONE after Flash operation end in order to avoid accidental repetition of the same command by the user application. On every new command invocation this variable must be set again. |

**Pre-conditions**

- Call FAL_Init to initialize the library
- Call FAL_Execute to initiate a FAL operation

**Post-conditions**

None

**Example**

See FAL_Execute

### 5.4.5 Administrative functions

#### 5.4.5.1 FAL_GetVersionString

**Description**

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

**Interface**

```
(const fal_u08*) FAL_GetVersionString( void );
```

**Arguments**

-

**Return types/values**

The library version is returned as string value in the following style:

"DV850T05xxxxyZabc"

with

x = supported compiler
y = compiler option
Z = "E" for engineering versions,
"V" for final versions
abc = Library version numbers according to version Va.b.c

**Pre-conditions**

None

**Post-conditions**

None

**Example**

```
fal_u08 *vstr_pu08;

vstr_pu08 = FAL_GetVersionString();
```

# Chapter 6 FDL Implementation into the user application

## 6.1 First steps

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.

## 6.2 Special considerations

### 6.2.1 Library handling by the user application

6.2.1.1 Function re-entrancy

All functions are not reentrant. So, reentrant calls of any EEL or FDL functions must be avoided.

6.2.1.2 Task switches, context changes and synchronization between FDL functions

All FDL functions depend on global FDL available information and are able to modify this. 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.

Example of not allowed sequence:

- Task1: Start an FDL operation with FDL_Execute
- Interrupt the function execution and switch to task 2, executing FDL_Handler function
- Return to task 1 and finish FDL_Execute function

As the FDL may not define critical sections which disable interrupts in order to avoid context changes and task switches, this synchronization need to be done by the user application.

### 6.2.2 Concurrent Data Flash accesses

Depending on the user application scenario, the Data Flash might be used for different purposes, e.g. one part is reserved for direct access by the user application and one part is reserved for EEPROM emulation by the Renesas EEL. The FDL is prepared to split the Data Flash into an EEL Pool and a User Pool.

On splitted Data Flash, the EEL is the only master on the EEL pool, accesses to this pool shall be done via the EEL API only.

Access to the user pool is done by using the FDL API functions for all accesses except read (e.g. FDL_Erase, FDL_Write, ...), while Data Flash read is directly done by the CPU.

The configuration of FDL pool and EEL pool (and resulting user pool) is done in the FDL descriptor.

6.2.2.1 User Data Flash access during active EEPROM emulation

Please refer to the EEL user manual regarding more detailed description of synchronization between EEPROM emulation and user accesses.

6.2.2.2 Direct access to the Data Flash by the user application by DMA

Basically, DMA transfers from Data Flash are permitted, but need to be synchronized with the EEL. Same considerations apply as mentioned in the last sub-chapter for accesses by the user application.

### 6.2.3 Entering power safe mode

Entering power safe modes is delayed by the device hardware until eventually ongoing Data Flash operations are finished.

# Chapter 7 Revision History

| Rev. | Date | Description | |
|---|---|---|---|
| | | Page | Summary |
| 1.00 | Mar 16, 2010 | — | First Edition issued |
| 1.1 | July 02, 2010 | 13 | Added new EEL and FDL sections (EEL_Const, FDL_Const) |
| | | 33 | Replaced 6.2.1.4 (“User Data Flash access during active EEPROM emulation using the EEL”) by 6.2.2 |
| | | 34 | Moved 6.2.1.3 to 6.2.3 |
| 1.2 | July 08, 2010 | 24 | changed heading “FAL_Resume” --> “FAL_EraseResume” |
| 1.3 | Oct. 11, 2010 | 27-29 | Added to describe FAL_Stand-by/Wake-up |
| 1.4 | Jul. 10, 2013 | 20, 31 | Fixed variablen name |