User Manual

32

Data Flash Access Library

FDL - T06

Data Flash Access Library for RC03F Flash based V850 devices

Renesas Electronics
www.renesas.com

R01US0045ED, Rev. 1.01
Mar 15, 2013

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.	
Organisation	This document describes the following sections: • Architecture • Implementation and Usage • API	
Note	Additional remark or tip	
Caution	Item deserving extra attention	
Numeric notation	Binary:	xxxx or xxxB
	Decimal:	xxxx
	Hexadecimal	xxxxH or 0x xxxx
Numeric prefixes	Representing powers of 2 (address space, memory capacity):	
	K (kilo): 2^{10} = 1024	
	M (mega):	$2^{20} = 1024^2$ = 1,048,576
	G (giga):	$2^{30} = 1024^3$ = 1,073,741,824
Register contents	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 Manual

(1) Purpose and Target Readers

This manual is designed to provide the user with an understanding of the library itself and the functionality provided by the library. It is intended for users designing applications using libraries provided by Renesas. A basic knowledge of software systems as well as Renesas microcontrollers is necessary in order to use this manual. The manual comprises an overview of the library, its functionality and its structure, how to use it and restrictions in using the library.

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 Usage Notes 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
API	Application Programming Interface
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.
Data Set	Instance of data written to the Flash by the EEPROM Emulation Library (EEL), identified by the Data Set ID
DS	Short for Data Set
Dual Operation	Dual operation is the capability to access flash memory during reprogramming another flash memory range. Dual operation is available between Code Flash and Data Flash. 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.
FAL	Flash Access Library (Flash access layer)
FCL	Code Flash Library (Code Flash access layer)
FDL	Data Flash Library (Data Flash access layer)
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 nonvolatile memory. The difference to ROM is, that this type of memory can be re-programmed several times.
Flash Area	Area of Flash consists of several coherent Flash Blocks

Flash Block	A flash block is the smallest erasable unit of the flash memory.
Flash Macro	A certain number of Flash blocks is grouped together in a Flash macro.
FW	Firmware
HWd	Half Word (16bit) data
ID	Identifier of a Data Set instance in the Renesas EEPROM Emulation
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
REE	Renesas Electronics Europe GmbH
REL	Renesas Electronics Japan
ROM	“Read only memory” - nonvolatile memory. The content of that memory can not be changed.
Segment / Section	Segment of Flash is a part of the flash that might consist of several blocks. Important is, that this segment can be protected against manipulation.
Self-Programming	Capability to reprogram the embedded flash without external programming tool only via control code running on the microcontroller.
Serial programming	The onboard programming mode is used to program the device with an external programmer tool.

Table of Contents

Chapter 1 Introduction

This user’s manual describes the internal structure, the functionality and software interfaces (API) of the Renesas V850 Data Flash Access Library (FDL) type T06. The library type T06 is suitable for all Renesas V850 Flash based on the RC03F Flash technology.

Caution Do not use this library for devices based on other Flash technologies than RC03F, as this might lead to unwanted behaviour or demolition of the device.

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 behaviour and programming faults might be the result.

The development environments of the companies Green Hills (GHS), IAR and Renesas 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 manual and other device dependent information can be downloaded from the following URL:

http://www.renesas.eu/update

Chapter 2 FDL Architecture

2.1 Flash Infrastructure

2.1.1 Complementary Read Flash

Based on the different application needs, the Flash implementation used for Data Flash differs from the Code Flash implementation. In order to achieve the required high endurance (erase cycles), Renesas decided for a Complementary Read (CR) Flash implementation on Data Flash. Each data bit is realized by two Flash cells, which are programmed to the opposite direction data bit. The cell value difference is read to judge the data value:

Data bit	level of Flash cell 1	level of Flash cell 2
0	high	low
1	low	high

Resulting from the implementation, erased Flash (both Flash cells with same/similar level) has a very small differential level. The resulting data bit judgement has an undefined result, but with a tendency to formerly written data. This need to be considered on interpretation of the read values:

- The lower level library FDL provides a blank check to distinguish between erased and written Flash on read level.
- When inspecting the Data Flash contents (e.g. using a debugger), the debugger need to provide the information on the Flash status (erased/written).

2.1.2 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, the 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!

2.1.3 Flash granularity

The Data Flash can be erased in 32 Byte units.

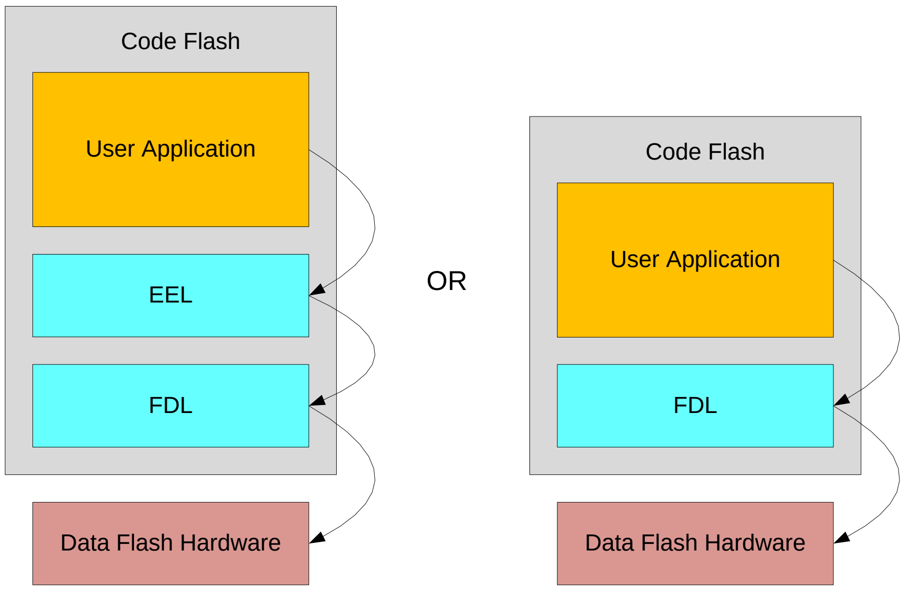
The Data Flash can be written and read in 2 Byte units. As the CPU is able to handle 4 Byte units as one “Word”, this document often refers to the “Half Word” (HWd) as 2 Byte units.

2.2 Layered software 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 2-1



Rough relationship between functional system blocks of the system

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.

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 FDL_ in order to distinguish the source/object modules for Code and Data Flash.

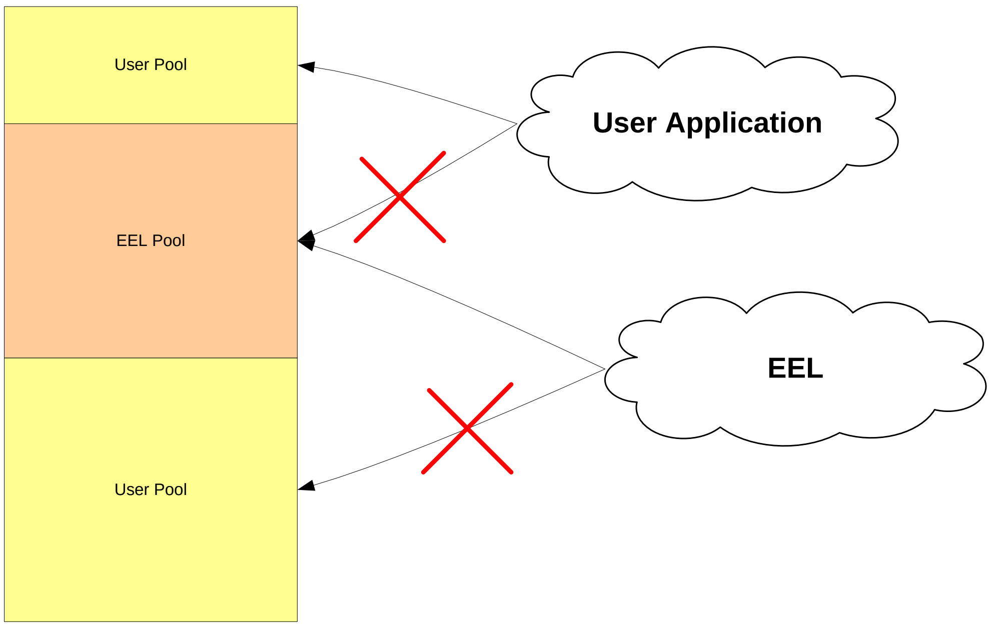
2.3 Data Flash Pools

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”.

Figure 2-2



Data Flash / FDL Pool

FDL-pool The 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 device depending physical Flash addresses (e.g. 0x02000000….0x0200FFFF or 0xFE000000….0xFE00FFFF) are transformed into a linear address room 0x0000….0xFFFF used by the FDL.

EEL-pool The 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 The 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.

2.4 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.

2.5 Bit error checks

Independent from the Flash manufacturer or Flash technology, Bit errors in the Flash 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 Renesas qualification and production quality and by Flash ECC (Error correction code), one major issue need to be 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 Renesas Data Flash hardware can correct single/double bit errors in a 16bit data word (+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.

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

Chapter 3 FDL Implementation

3.1 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 dataset 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.

3.1.1 Overview

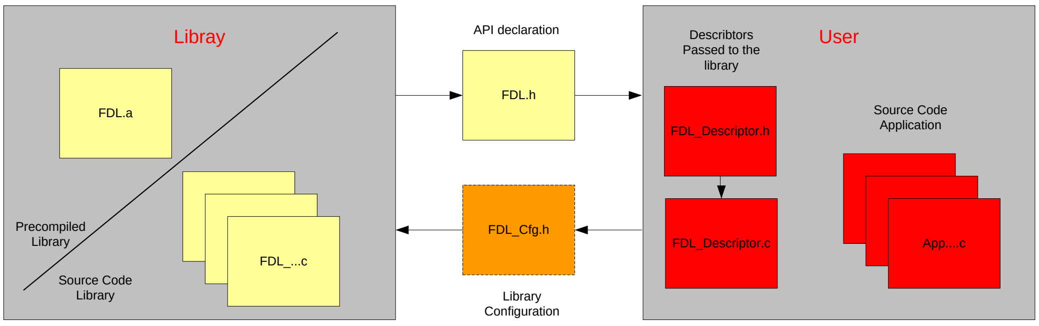
The following picture contains the library and application related files:

Figure 3-1

🟨 Library Files – Fix, may not be touched by the user

🟧 Library pre-compile configuration (Only on souce code delivery) – File name fix, File content user configurable

🟥 Application (User) Code – Completely in the hand of the user



Library and application file structure

The library code consists of different source files, starting with FDL/EEL_...The files shall 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 interface parameters and types are defined in the file FDL/EEL_Types.h.

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.

Caution **Wrong configuration of the EEL/FDL might lead to undefined results.**

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 4.2, “Run-time configuration” and 4.4.1.1, “FAL_Init”).

The structure of the descriptor is passed to the user application by FDL/EEL_Types.h, while 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.

3.1.2 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 and EEL (in the separate directories for EEL and FDL)

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_Main.c	Main source code
EELApp_Control.c	EEPROM emulation sample code
target.h	target device and application related definitions

root\<device name>\<compiler>\sample		
device header files	GHS	df<device number>.h
		df<device number>_irq.h
		io_macros_v2.h
	IAR	io_70f< device number>.h
		io_macros.h
		lxx.h
		cfi.h
startup file	GHS	DF<dev. num.>_startup.850
	IAR	l07.s85
		cstartup.s85
	REC	cstart.asm
linker directive file	GHS	df<dev. num.>.ld
	IAR	lnk70f<dev. num.>.xcl
	REC	df<dev. num.>.dir

root\<device name>\<compiler>\sample\FDL	
FDL.h	Header file containing function prototypes of the library user interface.
FDL_Types.h	Header file containing calling structures and error enumerations of the library user interface.
FDL_Descriptor.h	Descriptor file header with the run-time FDL configuration. To be edited by the user.
FDL_Descriptor.c	Descriptor file with the run-time FDL configuration. Using the defines of FDL_Descriptor.h. Should not be edited by the user.
FDL_User.c	Library related functions, which may be edited by the user
FDL_Cfg.h	Header file with definitions for library setup at compile time.
root\<device name>\<compiler>\sample\FDL\lib	
FDL_Env.h	Library internal defines for accessing the Flash programming hardware and Data Flash related definitions.
FDL_Global.h	Library internal defines, function prototypes and variables.
FDL_HWAccess.c	Source code for the library HW interface.
FDL_UserIF.c	Source code for the library user interface and service functions.

root\<device name>\<compiler>\sample\EEL	
EEL.h	Header file containing all function prototypes of the library user interface.
EEL_Types.h	Header file containing calling structures and error enumerations of the library user interface.
EEL_Cfg.h	Header file with definitions for library setup at compile time.
EEL_Descriptor.c	Descriptor file with the run-time EEL configuration. Using the defines of EEL_Descriptor.h and should not be edited by the user.
EEL_Descriptor.h	Descriptor file header with the run-time EEL configuration. To be edited by the user.
root\<device name>\<compiler>\sample\EEL\lib	
EEL_Global.h	Library internal defines, function prototypes and variables
EEL_BasicFct.c	EEL internal functions & state machine
EEL_UserIF.c	EEL user interface functions

3.2 FDL Linker sections

The following sections are Data Flash Access Library related.

Data sections

- FAL_DATA

  This section contains the variables required for FDL.

Code sections

- FAL_Text

  This section contains the hardware und user interface.

- FAL_Const

  This section contains all FDL library internal constant data.

3.3 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. Remaining findings are commented in the code while the QAC checker machine is set to silent mode in the concerning code lines.

Chapter 4 User Interface (API)

4.1 Pre-compile configuration

The pre-compile configuration of the FDL may be located in the FDL_cfg.h. The user has to configure all parameters and attributes by adapting the related constant definition in that header-file.

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

During initialization, the library needs to activate the device internal firmware for a short time in order to have access to certain data. This results in disabling the Code Flash. During that time, the Code Flash as well as Data Flash access is disabled. So, only Code from RAM can be executed, interrupts, NMIs and exceptions need to be relocated to RAM or to be disabled.
The critical code part with disabled Flash access is called critical section. The library provides the possibility to execute callback routines to disable, enable or relocate interrupts and exceptions at begin and end of the critical section.

Call back routine at critical section start (disable interrupts and exceptions):

```
#define FDL_CRITICAL_SECTION_BEGIN
```

Call back routine at critical section end (disable interrupts and exceptions):

```
#define FDL_CRITICAL_SECTION_END
```

4.2 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.

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

FAL_CPU_FREQUENCY_MHZ

This configuration element is set to the CPU frequency. A frequency fractional part need to be rounded up, e.g.: 25.3MHz need to be rounded up to 26MHz.

CPU frequency setting condition:
The Flash programming hardware is provided with a clock, derived from the CPU frequency. The frequency divider of this derived clock is device family dependent. The resulting $f_{\text{Flash hardware}}$ must be in the range of 8 to 50MHz.

E.g.: Fx4-L, Px4-L:
$f_{\text{Flash hardware}} = f_{\text{Cpu}} / 2$
⇨ 16MHz <= f_{Cpu} <= mimimum of <100MHz> or <maximum device frequency>

Caution: The CPU frequency must be set correctly. If not, malfunction may occur such as unstable Flash data without data retention, programming failure, operation blocking.

```
#define FAL_CPU_FREQUENCY_MHZ                48
```

FAL_EEL_VIRTUALBLOCKSIZE

The physical erase unit of the Data Flash is 32Byte. However, based on the compatibility to former Data Flash implementations, the EEL groups together each 64 erasable Blocks to a 2kB virtual block, which are then handled by the EEL. So, EEL-Pool start and size must be aligned to the 2kB boundary. For FAL-Pool size this is not necessary and may be changed accordingly, but in this sample configuration also FAL-Pool size is aligned to 2kB.

```
#define FAL_EEL_VIRTUALBLOCKSIZE             64
```

FAL_FAL_POOL_SIZE

Defines the number of physical Flash 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.

E.g.:
Data Flash size = 32kb, block size = 32Bytes → FAL_FAL_POOL_SIZE 1024

```
#define FAL_FAL_POOL_SIZE                   16*FAL_EEL_VIRTUALBLOCKSIZE
```

Value range:

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

FAL_EEL_POOL_START

Define to set the number of the first physical Data Flash block used as EEL-Pool. The value should be set to zero if the FAL is used only.

Caution The block number must be aligned to the virtual block size (2kB) used by the EEL! Following that, the block number must be a multiple of 64 (2kB = 64 * 32Byte)!

```
#define FAL_EEL_POOL_START                  1*FAL_EEL_VIRTUALBLOCKSIZE
```

Value range:

Min:	FAL Pool start block
Max:	sum of FAL_EEL_POOL_START and FAL_EEL_POOL_SIZE is less or equal than FAL_FAL_POOL_SIZE

FAL_EEL_POOL_SIZE

Defines the number of blocks used for the EEL-Pool.

Caution The block number must be aligned to the virtual block size (2kB) used by the EEL! Following that, the block number must be a multiple of 64 (2kB = 64 * 32Byte)!

```
#define FAL_EEL_POOL_SIZE                    6*FAL_EEL_VIRTUALBLOCKSIZE
```

Value range:

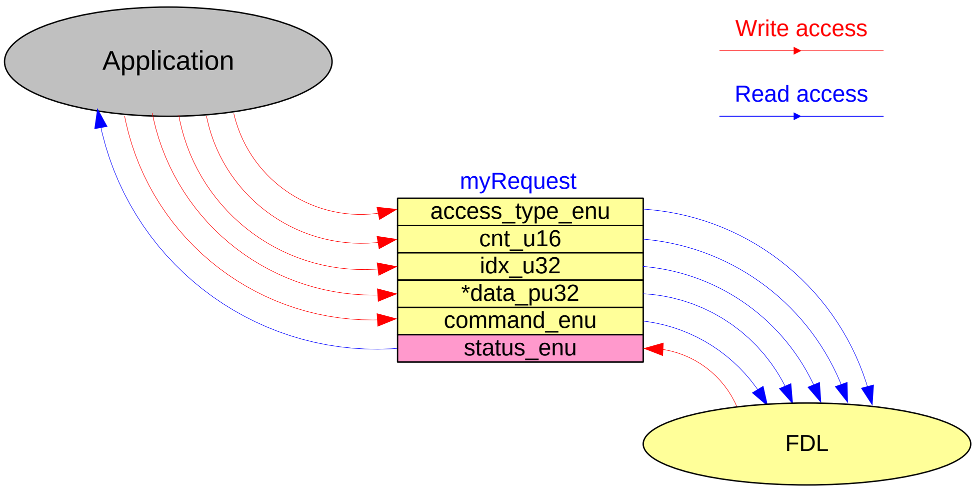
Min:	64 * 4 blocks (required for proper EEL operation)
Max:	FAL pool size, while the sum of FAL_EEL_POOL_START and FAL_EEL_POOL_SIZE is less or equal than FAL_FAL_POOL_SIZE

4.3 Data Types

4.3.1 User operation request structure

All user operations are initiated by a central initiation function (see chapter 4.4.4.1, “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 4-1



Request structure handling

```
/* FDL operations request structure, required for FAL_Execute */
typedef volatile struct FAL_REQUEST_T {
        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;
```

command_enu

User command to execute

```
/* FDL operation initiation command */
typedef enum FAL_COMMAND_T {
       FAL_CMD_ERASE,       /* Flash erase (Multiple blocks) */
       FAL_CMD_WRITE,       /* Flash write (Multiple words) */
       FAL_CMD_BLANKCHECK,  /* Flash blank check (Multiple words) */
       FAL_CMD_BITCHECK     /* Flash bit check (Multiple words) */
} fal_command_t;
```

All commands can be requested by using the FAL_Execute function.

Note All commands operate on virtual addresses (relative address that start from block 0 of the data Flash memory as address 0) and block numbers.

E.g.:

1st HWd of the Data Flash: 0x00000000

3rd HWd of the Data Flash: 0x00000004

dataAdd_u32

Address of the write buffer of the application (parameter only necessary during write commands)

idx_u32

Write	Destination byte index (address), relative to the DF start address
Erase	Block index of the 1st block to erase
Blank Check	Check start byte index (address), relative to the DF start address
Bit Error Check	Check start byte index (address), relative to the DF start address

cnt_u16

Write	Number of HWds to write
Erase	Number of blocks to erase
Blank Check	Number of HWds to check
Bit Error Check	Number of HWds to check

status_enu

Library return codes (see 4.3.2, “Status and Error Codes”)

accessType_enu

Access right definition

```
/* FDL operations originator defines */
typedef enum FAL_ACCESS_TYPE_T {
       FAL_ACCESS_NONE,     /* FDL internal value. Not used. */
       FAL_ACCESS_USER,     /* FDL operation by user application */
       FAL_ACCESS_EEL       /* FDL operation by EEL */
} fal_access_type_t;
```

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 FAL_Execute, this function will return a protection error.

4.3.2 Status and Error Codes

```
/* FDL status return values */
typedef enum FAL_STATUS_T {
   FAL_OK,                  /* Operation terminated successfully */
   FAL_BUSY,                /* Operation is still ongoing */
   FAL_SUSPENDED,           /* Operation is suspended */
   FAL_ERR_PARAMETER,       /* Wrong parameter in FDL function call */
   FAL_ERR_PROTECTION,      /* Operation blocked (wrong parameters) */
   FAL_ERR_REJECTED,        /* Flow error - other operation ongoing */
   FAL_ERR_WRITE,           /* Flash write error */
   FAL_ERR_ERASE,           /* Flash erase error */
   FAL_ERR_COMMAND,         /* Unknown command */
   FAL_ERR_BITCHECK,        /* Bit check error */
   FAL_ERR_INTERNAL         /* Library internal error */
} fal_status_t;
```

All status and error codes depend on the called command. The Library offers a set of commands that all can be requested by using the FAL_Execute function.

4.3.3 FAL_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 command is initiated by FAL_Execute and is executed by the sequencer to perform the physical erase. After the erase command has been initiated FAL_Handler must be called to complete it and to update the library status.

Status	Class	Background and Handling	
FAL_BUSY	normal	meaning	operation started successfully
		reason	no problems during execution
		remedy	call FAL_Handler until the Flash operation is finished, reported by the request structure status return value
FAL_OK	normal	meaning	operation finished successfully
		reason	no problems during execution
		remedy	nothing
FAL_SUSPENDED	normal	meaning	an ongoing Flash operation is suspended by user application request
		reason	FAL_SuspendRequest called and successfully finished
		remedy	start another operation or resume the suspended operation using FAL_ResumeRequest command
FAL_ERR_PARAMETER	heavy	meaning	current command is rejected
		reason	wrong parameters have been passed to the FAL: the range (start block) to (Start block + Number of blocks - 1) must be in the EEL/User-Pool

Status	Class	Background and Handling	
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_PROTECTION	heavy	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)
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_REJECTED	heavy	meaning	current command is rejected
		reason	any other operation is ongoing
		remedy	repeat the command when the preceding operation has finished
FAL_ERR_ERASE	heavy	meaning	at least one bit within the specified blocks is not erased
		reason	one or more Flash bits could not be erased completely
		remedy	a Flash block respectively the complete Data Flash should be considered as defect
FAL_ERR_INTERNAL	heavy	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
		remedy	refrain from further Flash operations and investigate in the root cause

4.3.4 FAL_CMD_BLANKCHECK

The blank-check command can be used by the requester to check if all cells in the specified target flash area are possibly blank, e.g. before writing data into it or reading data. The user can use the blank-check command freely as it is a non-destructive flash access.

Different from an Erase operation, which checks if the cells to be erased have really reached the erase level, the blank check does not check on this level but on a standard read level. It cannot ensure any electrical margin required for data retention but just give an indication if cells are possibly erased or possibly written. Therefore the blank check results need to be interpreted correctly:

Note **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. On blank check pass it is ensured that the cells are not completely written. This might result from successfully erased Flash blocks, but also from interrupted erase or write operations**

The check command is initiated by FAL_Execute and is executed by the sequencer. After that, FAL_Handler must be called frequently to complete the command and check the status.

Status	Class	Background and Handling	
FAL_BUSY	normal	meaning	operation started successfully
		reason	no problems during execution
		remedy	call FAL_Handler until the Flash operation is finished, reported by the request structure status return value
FAL_OK	normal	meaning	operation finished successfully
		reason	no problems during execution
		remedy	nothing
FAL_ERR_PARAMETER	heavy	meaning	current command is rejected
		reason	wrong parameters have been passed to the FAL: the range (start HWd) to (Start HWd + Number of HWds - 1) must be in the EEL/User-Pool
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_PROTECTION	heavy	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)
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_REJECTED	heavy	meaning	current command is rejected
		reason	any other operation is ongoing
		remedy	repeat the command when the preceding operation has finished
FAL_ERR_BLANKCHECK	heavy	meaning	at least one bit within the specified range is not blank
		reason	for any bit in the addressed flash range, the voltage level is below specification for an blank cell
		remedy	depending on the Blank Check usage concept.
FAL_ERR_INTERNAL	heavy	meaning	a library internal error occurred, which could not happen in case of normal application execution

Status	Class	Background and Handling	
		reason	application bug (e.g. program run-away, destroyed program counter) or hardware problem
		remedy	refrain from further Flash operations and investigate in the root cause

4.3.5 FAL_CMD_BITCHECK

A Flash cell, especially a not completely written or erased one (e.g. operation interrupted due to a power fail), might drift over time. Cyclic bit-checks on the data may detect possible data retention problems and the user can refresh the data if the correctness of the data is ensured.

The bit-check command bases on the Flash internal ECC. The ECC circuit is able to detect and correct a single bit error. Multiple bit errors will be detected only with a high confidence level but cannot be corrected. The bit-check command signalizes any detected and eventually corrected bit failure.

The user can use the bit-check command freely as it is a non-destructive flash access.

Status	Class	Background and Handling	
FAL_OK	normal	meaning	operation finished successfully
		reason	no problems during execution
		remedy	nothing
FAL_ERR_PARAMETER	heavy	meaning	current command is rejected
		reason	wrong parameters have been passed to the FAL: the range (start HWd) to (Start HWd + Number of HWds - 1) must be in the EEL/User-Pool
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_PROTECTION	heavy	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)
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_REJECTED	heavy	meaning	current command is rejected
		reason	any other operation is ongoing
		remedy	repeat the command when the preceding operation has finished
FAL_ERR_BITCHECK	normal	meaning	a bit error is found in at least one data word within the specified range

Status	Class	Background and Handling	
		reason	possible causes: 1) erased Flash cells normally have bit errors 2) not completely written Flash, e.g. caused by a power fail during a Flash operation 3) long time frame between erase or write and bit error check
		remedy	depending on the error reason: 1) do not execute bit error check on erased cells 2) refresh the data 3) refresh the data
FAL_ERR_INTERNAL	heavy	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
		remedy	refrain from further Flash operations and investigate in the root cause

4.3.6 FAL_CMD_WRITE

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

The write command is initiated by FAL_Execute and is executed by the sequencer to perform the physical write. After the write command has been initiated FAL_Handler must be called to complete it and to update the library status.

Status	Class	Background and Handling	
FAL_BUSY	normal	meaning	operation started successfully
		reason	no problems during execution
		remedy	call FAL_Handler until the Flash operation is finished, reported by the request structure status return value
FAL_OK	normal	meaning	operation finished successfully
		reason	no problems during execution
		remedy	nothing
FAL_SUSPENDED	normal	meaning	an ongoing Flash operation is suspended by user application request
		reason	FAL_SuspendRequest called and successfully finished

Status	Class	Background and Handling	
		remedy	start another operation or resume the suspended operation using FAL_ResumeRequest command
FAL_ERR_PARAMETER	heavy	meaning	current command is rejected
		reason	wrong parameters have been passed to the FAL: the range (start HWd) to (Start HWd + Number of HWds - 1) must be in the EEL/User-Pool
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_PROTECTION	heavy	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)
		remedy	refrain from further Flash operations and investigate in the root cause
FAL_ERR_REJECTED	heavy	meaning	current command is rejected
		reason	any other operation is ongoing
		remedy	repeat the command when the preceding operation has finished
FAL_ERR_WRITE	heavy	meaning	at least one HWd could not be written correctly
		reason	1) for any bit of the written area, the voltage levels are below specification for a written cell 2) a Flash write operation on a not blank cell failed
		remedy	a Flash block respectively the complete Data Flash should be considered as defect
FAL_ERR_INTERNAL	heavy	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
		remedy	refrain from further Flash operations and investigate in the root cause

4.4 Library Functions

4.4.1 Initialization

4.4.1.1 FAL_Init

Description

Function is executed before any execution of other FDL Flash operations.

The main functionality of the FAL_Init is:

- Initialization of the Flash programming hardware
- Set internal frequency
- Initialization of internal FDL variables

Note **FAL_Init need to access the device internal firmware in order to initialize the Flash programming hardware. During that time, the Code Flash is not accessible and so, no interrupts or exceptions can be served. The library can execute code to disable interrupts and/or exceptions during that time as user callback functions. The user must configure these by the library pre-compile configuration (see 4.1 "Pre-compile configuration").**

Interface

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

Arguments

Type	Argument	Description
fal_descriptor_t	descriptor_pstr	Pointer to the FDL run-time configuration descriptor in ROM

Return types / values

Type	Argument	Description
fal_status_t		Operation status when returned from function call: • FAL_OK • FAL_ERR_PARAMETER • FAL_ERR_INTERNAL

Pre-conditions

None

Post-conditions

None

Example

```
fal_status_t  ret;

/* Initialze FDL */
ret = FAL_Init( &fal_RTCONFIG_enu );

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

4.4.2 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.

Note 1 **When FAL_SuspendRequest has already suspended a Flash Erase, another Flash erase is not possible. If FAL_SuspendRequest has already suspended a Flash Write, another Flash Erase or Write is not possible.**

Note 2 **The suspend / resume mechanism can not be nested. Therefore, the following sequence is not allowed:**

Any Flash operation → suspend → any Flash operation → suspend

4.4.2.1 FAL_SuspendRequest

Description

This function requests suspending a Flash operation in order to be able to do other Flash operations or Flash read.

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_ResumeRequest.

In case of any violation of the correct function usage, the function will return FAL_ERR_REJECTED.

Interface

```
fal_status_t FAL_SuspendRequest( void )
```

Arguments

None

Return types / values

Type	Argument	Description
fal_status_t		Operation status when returned from function call: • FAL_OK • FAL_ERR_REJECTED

Pre-conditions

- A Flash operation must have been started.

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

Post-conditions

Call FAL_Handler until the library is suspended

Example

```
fal_status_t  srRes_enu;
fal_request_t myReq_str;
fal_u32       i;

/* Start Erase operation */
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 );

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

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

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

while( FAL_SUSPENDED != myReq_str.status_enu )
{
    FAL_Handler();
}


/* FAL is suspended. Handle other operations or read the Flash */
/* ... */


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

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

/* Finish the erase */
while( myReq_str.status_enu == FAL_SUSPENDED )
{
    FAL_Handler();
}
while( myReq_str.status_enu == FAL_BUSY )
{
    FAL_Handler();
}

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

4.4.2.2 FAL_ResumeRequest

Description

This function requests to resume the FAL operation after suspending. The resume is just requested by this function. Resume handling is done by the FAL_Handler function.

In case of any violation of the correct function usage, the function will return FAL_ERR_REJECTED.

Interface

```
fal_status_t FAL_ResumeRequest( void )
```

Arguments

None

Return types / values

Type	Argument	Description
fal_status_t		Operation status when returned from function call: • FAL_OK • FAL_ERR_REJECTED

Pre-conditions

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

Post-conditions

Call FAL_Handler until the library is resumed

Example

See FAL_SuspendRequest

4.4.3 Standby / Wakeup

The stand-by functionality shall suspend ongoing Flash operations asynchronously to the normal FDL handling flow, e.g. by using a high priority interrupt function.

It 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 suspend-able). So, the function FAL_StandBy tries to suspend the Flash operation and returns FAL_BUSY as long as a Flash operation is ongoing. If suspend was not possible (e.g. blank check operation), FAL_BUSY is returned until the operation is finished normally.
So, in order to be sure to have no Flash operation ongoing, the function must be called continuously until the function does no longer return FAL_BUSY or until a timeout occurred.

After stand-by, it is mandatory to call FAL_WakeUp before entering normal FAL operation again. The prescribed sequence in case of using FAL_StandBy/WakeUp is:

1. any FAL operation
2. FAL_StandBy
3. FAL_Handler until operation in standby
4. device power safe
5. device wake-up
6. FAL_WakeUp
7. continue FAL operations

Caution **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.**

4.4.3.1 FAL_StandBy

Description

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

In case of any violation of the correct function usage, the function will return FAL_ERR_REJECTED.

Interface

```
fal_status_t FAL_StandBy( void )
```

Arguments

None

Return types / values

Type	Argument	Description
fal_status_t		Operation status when returned from function call: • FAL_OK • FAL_BUSY • FAL_ERR_REJECTED

Pre-conditions

- The library must be initialized
- The following sequence is not allowed:

  Flash Erase → FAL suspend → Flash Write → FAL standby
- The FAL is not in standby mode

Post-conditions

- Continue calling FAL_StandBy until it no longer returns FAL_BUSY
- Execute FAL_WakeUp as next FAL function

Example

```
fal_status_t   fdlRet_enu;
fal_request_t  myReq_str;

/* Start Erase operation */
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 );

...
do
{
    fdlRet = FAL_StandBy();
}
while( FAL_BUSY == fdlRet );
if( FAL_OK != fdlRet )
{
    /* error handler */
}

...
/* device enters power safe mode */
...
/* device recovers from power safe mode */
...

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


/* Finish the erase */
while( myReq_str.status_enu == FAL_BUSY )
{
    FAL_Handler();
}

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

4.4.3.2 FAL_WakeUp

Description

This function wakes-up the library from the former entered Standby.

In case of any violation of the correct function usage, the function will return FAL_ERR_REJECTED.

Interface

```
fal_status_t FAL_WakeUp( void )
```

Arguments

None

Return types / values

Type	Argument	Description
fal_status_t		Operation status when returned from function call: • FAL_OK • FAL_ERR_REJECTED

Pre-conditions

- The library must be initialized
- The FAL is in standby mode

Post-conditions

None

Example

See FAL_StandBy

4.4.4 Operational functions

4.4.4.1 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.

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.

Depending on the used command, different combinations of operation and parameter are possible.

Command	Description
FAL_CMD_ERASE	Performs erase for each block of the specified range. The following arguments must be set for execution: • idx_u32: Start block index (block number) • cnt_u16: Numbers of blocks to erase • accessType_enu: Access rights (see 4.3.1)
FAL_CMD_WRITE	Write the data placed in the write input buffer to the Data Flash at the specified relative starting address for the specified number of HWds. The following arguments must be set for execution: • idx_u32: Start byte index (relative address) • cnt_u16: Number of HWds to check • accessType_enu: Access rights (see 4.3.1)

Command	Description
FAL_CMD_BLANKCHECK	Performs internal checks starting from the specified beginning address of the Data Flash for the area in the execution range. The following arguments must be set for execution: • idx_u32: Start byte index (relative address) • cnt_u16: Number of HWds to check • accessType_enu: Access rights (see 4.3.1)
FAL_CMD_BITCHECK	Performs internal checks starting from the specified beginning address of the Data Flash for the area in the execution range. The following arguments must be set for execution: • idx_u32: Start byte index (relative address) • cnt_u16: Number of HWds to check • accessType_enu: Access rights (see 4.3.1)

Interface

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

Arguments

Type	Argument	Description
fal_request_t	request_pstr	Address of the structure specifying the command to be executed.

Return types / values

Type	Argument	Description
fal_request_t	request_pstr.status_enu	Operation status when returned from function call: • FAL_BUSY • FAL_ERR_PARAMETER • FAL_ERR_REJECTED • FAL_ERR_COMMAND • FAL_OK • FAL_ERR_BITCHECK

Pre-conditions

The library must be initialized

Post-conditions

Call FAL_Handler until the Flash operation is finished. This is reported by the request structure status return value.

Example

- Erase blocks 0 to 3:

```
myRequest.command_enu       = FAL_ERASE
myRequest.idx_u32           = 0
myRequest.cnt_u16           = 4
myRequest.accessType_enu    = FDL_ACCESS_USER;

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

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

- Write Data to addresses 0x100 to 0x107:

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

myRequest.command_enu       = FAL_WRITE
myRequest.idx_u32           = 0x100
myRequest.cnt_u16           = 4
myRequest.dataAdd_u32       = &data[0];
myRequest. accessType_enu   = FDL_ACCESS_USER;

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

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

- Blank check on the address range 0x100 to 0x107:

```
myRequest.command_enu       = FAL_CMD_BLANKCHECK
myRequest.idx_u32           = 0x100
myRequest.cnt_u16           = 4
myRequest. accessType_enu   = FDL_ACCESS_USER;

FAL_Execute( &myRequest );

while( myRequest.status_enu == FAL_BUSY )
{
    FAL_Handler();
}

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

- Bit check on the address range 0x100 to 0x107:

```
myRequest.command_enu       = FAL_CMD_BITCHECK
myRequest.idx_u32           = 0x100
myRequest.cnt_u16           = 4
myRequest. accessType_enu   = FDL_ACCESS_USER;

FAL_Execute( &myRequest );

while( myRequest.status_enu == FAL_BUSY )
{
    FAL_Handler();
}

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

4.4.4.2 FAL_Handler

Description

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

The function checks the operation status and updates the request structure status_enu variable when the operation has finished. By that, the operations end can be polled.

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

Interface

```
void FAL_Handler( void )
```

Arguments

None

Return types / values

Type	Argument	Description
fal_status_t		Operation status when returned from function call: • FAL_OK • FAL_BUSY • FAL_SUSPENDED • FAL_ERR_PROTECTION • FAL_ERR_WRITE • FAL_ERR_ERASE • FAL_ERR_BLANKCHECK • FAL_ERR_BITCHECK • FAL_ERR_INTERNAL

Pre-conditions

- The library must be initialized
- FAL_Execute must be executed

Post-conditions

None

Example

See FAL_Execute

4.4.5 Administrative functions

4.4.5.1 FAL_GetVersionString

Description

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

Interface

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

Arguments

None

Return types / values

Type	Argument	Description	
fal_u08		Pointer to version string Version string format: "DV850T06xxxxxYZabcD"	
		xxxxx	Coded information about the supported compiler. If no information is coded, the library is a generic library valid for different compiler.
		Y	Coded information about the used memory/register model. If no information is coded, the library is a generic library valid for all memory/register models.
		Z	"E" for engineering version
			"V" for normal version
		abc	Library version number a.bc
		optional:	
		D	optional character, identifying different engineering versions

Pre-conditions

None

Post-conditions

None

Example

```
/* Read library version */
fal_u08   *version_pu08;

version_pu08 = FAL_GetVersionString( );
```

Chapter 5 Integration into the user application

5.1 First steps

It is very important to have theoretic background about the Data Flash and the EEL and 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 application sample.

5.2 Application sample

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

Note: **Before the first compile run, the compiler path must be configured in the application sample file "makefile":**

Set the variable COMPILER_INSTALL_DIR to the correct compiler directory

5.3 Special considerations

5.3.1 Function reentrancy

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

5.3.2 Task switch, context change and synchronization between functions

Each function depends on global FDL 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.

5.3.3 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 partitioned 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.

5.3.4 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.

5.3.5 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.

5.3.6 Entering power safe mode

Entering power safe mode is not allowed at all during ongoing Data Flash operations. Use FAL_StandBy or wait until operations are no longer busy.

Revision History

Chapter	Page	Description
Rev. 1.00		
		Initial version
Rev. 1.01		
4.2	20	Updated frequency setting description

Data Flash Access Library

Renesas Electronics Corporation

R01US0045ED