User Manual

16

Data Flash Access Library

Type T01, European Release

16 Bit Single-chip Microcontroller
78K0R/Fx3 Series

Installer: RENESAS_FDL_78K0R_T01E_Vx.xxx

Renesas Electronics
www.renesas.com

R01US0005ED0102
June 25, 2013

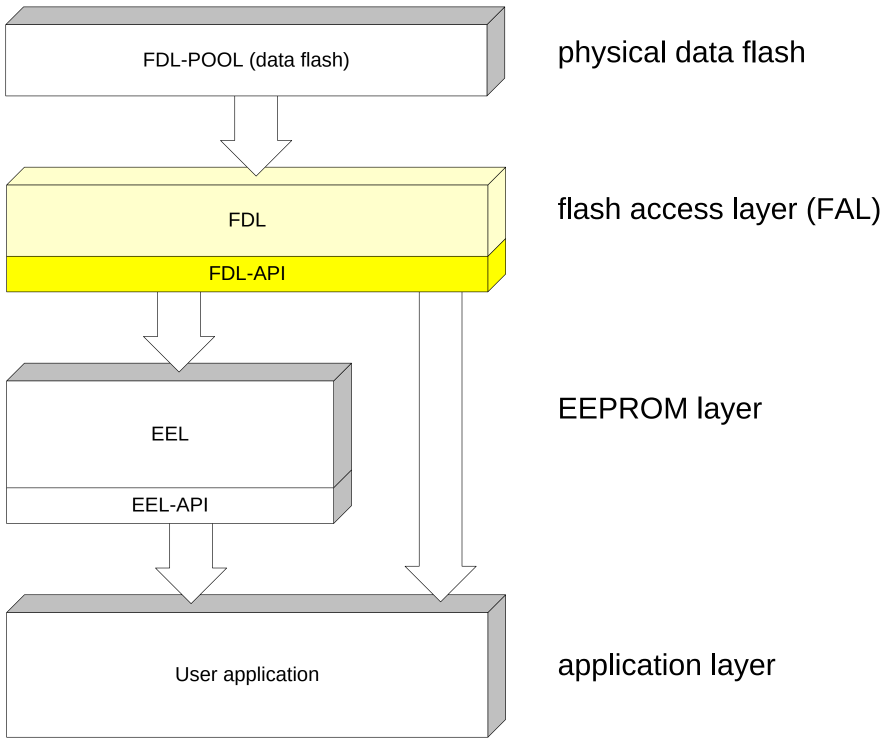
Table of Contents

Chapter 1 Introduction

This user's manual describes the overall structure, functionality and software interfaces (API) of the Data Flash Library (FDL) accessing the physical Data Flash separated and independent from the Code Flash. This library supports dual operation mode where the content of the Data Flash is accessible (read, write, erase) during instruction code execution.

The FAL (flash access layer) is a layer of EEPROM emulation system and encapsulates the low-level access to the physically flash in secure way. In case of Data Flash this layer is using the FDL. It provides a functional socket for Renesas EEPROM emulation software, but beside this it offers also direct access to the user at witch the access priority and access separation is fully controlled by the library.

Figure 1-1 Components of the EEPROM emulation system



To boost the flexibility and the real-time characteristics of the library it offers only fast atomic functionality to read, write and erase the Data Flash memory at smallest possible granularity. Beside the pure access commands some maintenance functionality to check the quality of the flash content is also provided by the library

1.1 Components of the EEPROM Emulation System

To achieve a high degree of encapsulation the EEPROM emulation system is divided into several layers with narrow functional interfaces.

1.1.1 Physical flash layer

The FDL is accessing the Data Flash as a physical media for storing data in the EEPROM emulation system. The Data Flash is a separate memory that can be accessed independent of the Code Flash memory. This allows background access to data stored in the Data Flash during program execution located in the code flash. The physical Data Flash is mapped by the FDL into a virtual pool called FDL-Pool below.

1.1.2 Flash access layer

The Data Flash access layer is represented by the flash access library provided by Renesas. In case of devices incorporating data-flash the FDL is representing this layer. It offers all atomic functionality to access the FDL pool. To isolate the data-flash access from the used flash-media this layer (the FDL) is transforming thy physical addresses into a virtual, linear address-room.

1.1.3 EEPROM access layer

The EEPROM layer allows read/write access to the Data Flash at abstract level. It is represented by Renesas EEL or alternatively any other, user specific implementation.

1.1.4 Application layer

The application layer is user’s application software that can use freely all visible (specified by the API definition) commandos of upper layers. The EEPROM layer and the flash access layer can be used asynchronously. The FDL manages the access rights to it in a proper way.

1.2 Naming Conventions

Certain terms, required for the description of the Data Flash Access and EEPROM emulation library 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 Architecture

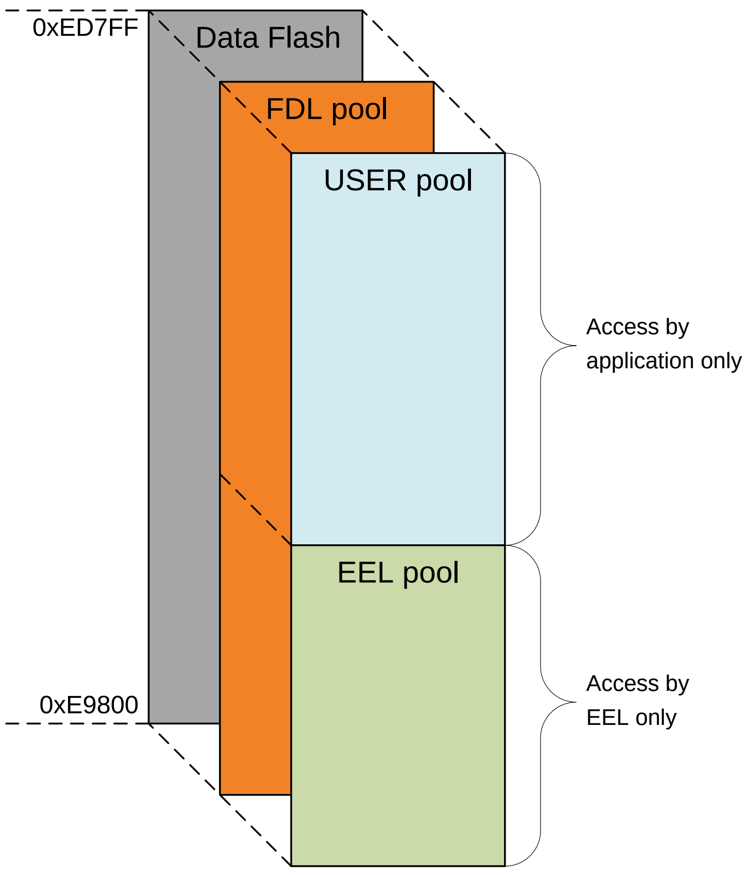
This chapter describes the overall architecture of the FDL library.

2.1 Data Flash fragmentation

The physical Data Flash location is fixed to a physical address assigned by the hardware (e.g. for 78K0R/Fx3: 0xE9800 – 0xED7FF). Just the logical fragmentation of the Data Flash can be configured within the given range.

Following figure shows the logical fragmentation of physical Data Flash.

Figure 2-1 Logical fragmentation of physical Data Flash



2.1.1 FDL pool

The FDL pool defines the maximum usage of physical Data Flash used by the FDL. In case of physical Data Flash size of 16KByte it is possible to define the following sizes for FDL pool configuration: 2KByte, 4KByte, 6KByte, 8KByte, 10KByte, 12KByte, 14KByte, 16KByte. This pool is divided into the EEL and USER pool which are described below.

2.1.2 EEL pool

EEL pool is a part of the FDL pool and is assigned exclusively to Renesas EEPROM Emulation Library (EEL) only. In case the EEL is not used the whole FDL pool will be reserved for USER pool.

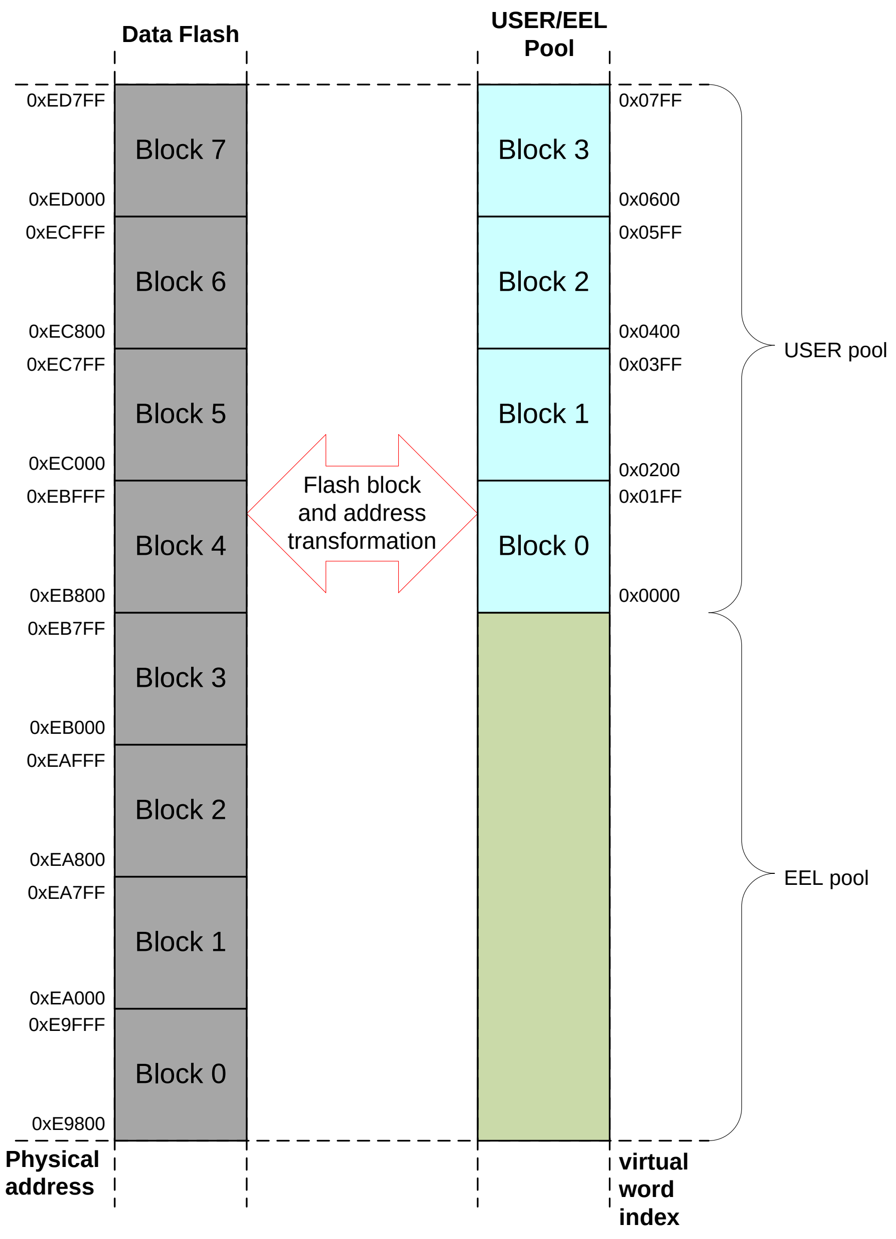
2.1.3 USER pool

The USER pool is a part of the FDL pool. It can be used exclusively by the application in a free way. In case of proprietary EEPROM emulation implementation (user specific) the completely FDL pool has to be configured as USER-pool.

2.2 Address virtualization

To facilitate the access to the USER pool the physical addresses were virtualized. The virtualized pool looks like a simple one-dimensional array of flash-words (4 bytes).

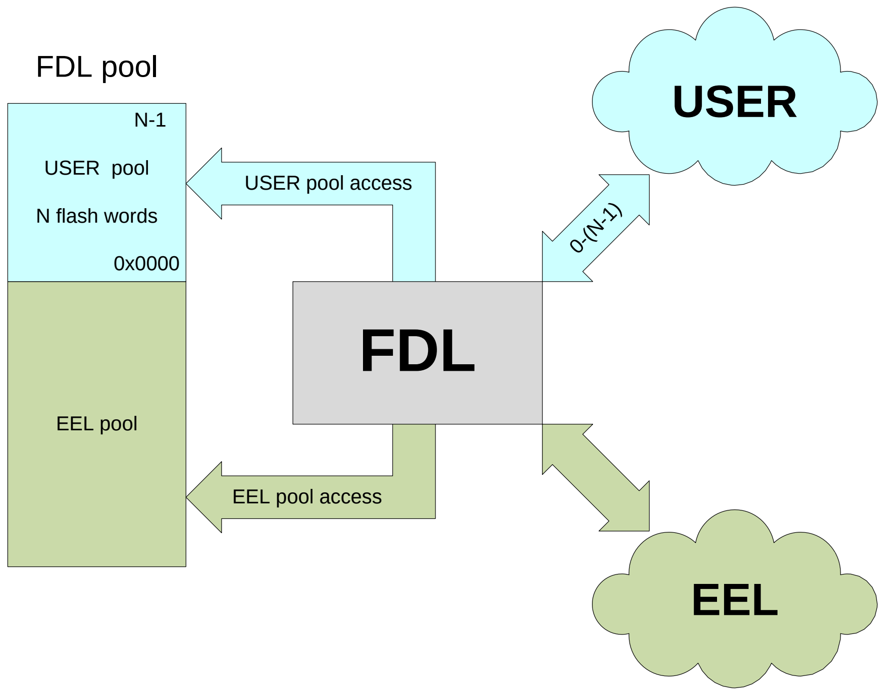
Figure 2-2 Relationship between physical and virtual pool addresses



2.3 Access right supervision

As mentioned before the complete FDL pool is divided into two parts shared between user and the EEL. The construction of the FDL does not allow user access to the EEL-pool and vice versa.

Figure 2-3 FDL pool access supervision



2.4 Request-Response architecture

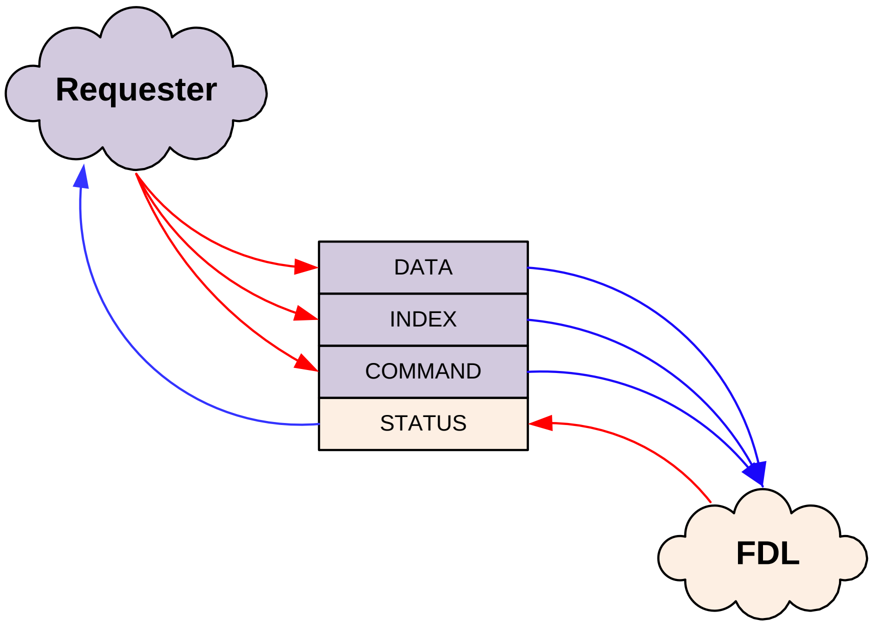
The communication between the requester (user) and the executor (here the FDL) is a common structured request variable. The requester can specify the request and pass it to the FDL. After acceptance the progress of the execution can be checked by polling the request status.

From execution-time point of view the commands of the FDL are divided into two groups:
- suspendable block-oriented command like block erase taking relatively long time for its execution
- not-suspedable word-oriented commands like write, read ... taking very short time for its execution

Depending on the real-time requirements the user can decide if independent, quasi-parallel execution of block and word commands is required or not. In such a case two separate request-variables have to be defined and managed by the application. Please refer to chapter “Operation” for details.

Following figure shows the access from requester and FDL point of view.

Figure 2-4 Request oriented communication between FDL and its requester



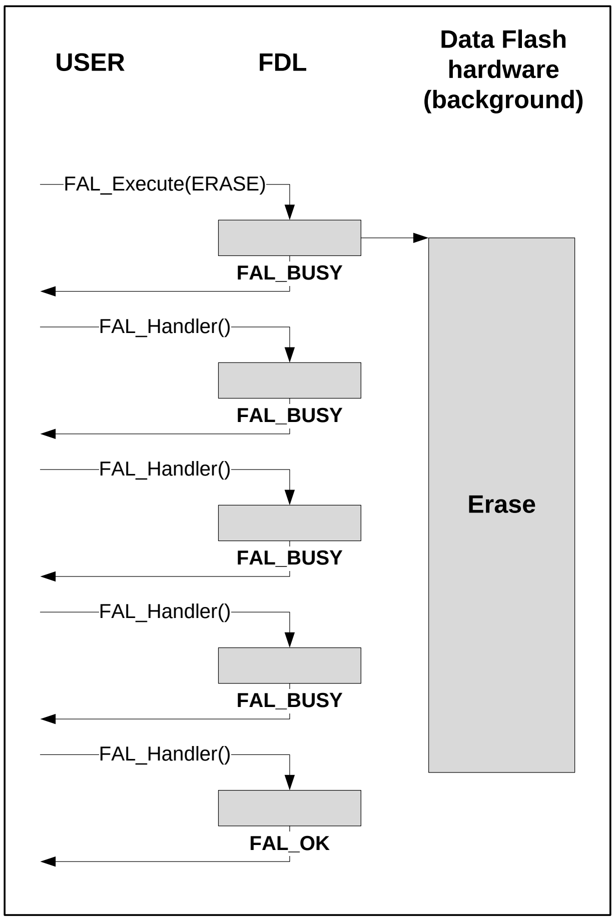
2.5 Background operation

Due to the fact that the Data Flash operates in the background it is possible to do something else in the meantime. For example the application could prepare next data for writing into the Data Flash or handle different ISRs. Background operation is a powerful feature especially in operation systems were each task could start FAL commands which will be executed in the background during task switching.

2.5.1 Background operation (Erase)

The erase command is from timing point of view the longest command. As shown in the figure below, the application has the possibility to execute other user code during the background operation.

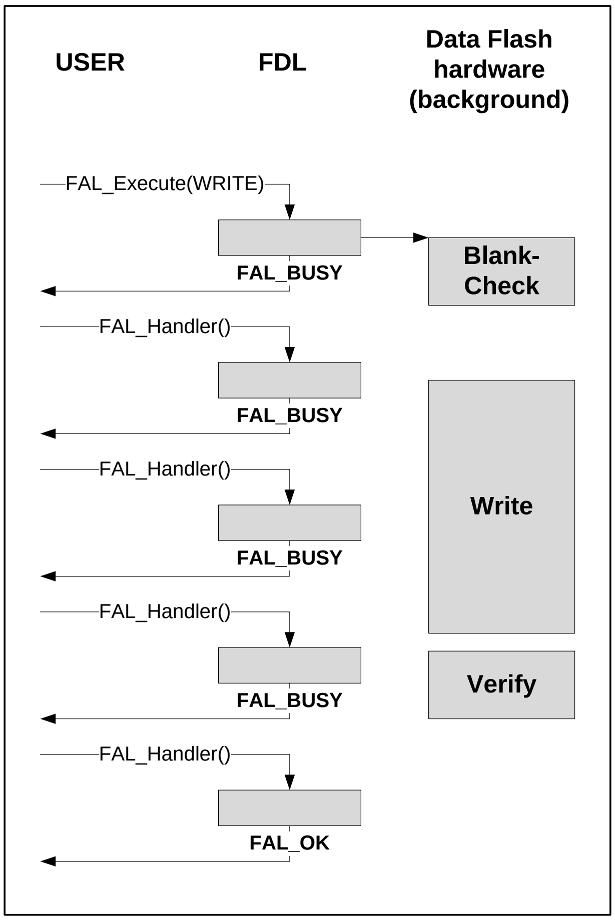
Figure 2-5 Background operation (Erase)



2.5.2 Background operation (write)

During the running write command blank-check/write/verify will be performed in background. As shown in the figure below, the application has the possibility to execute other user code during the background operation.

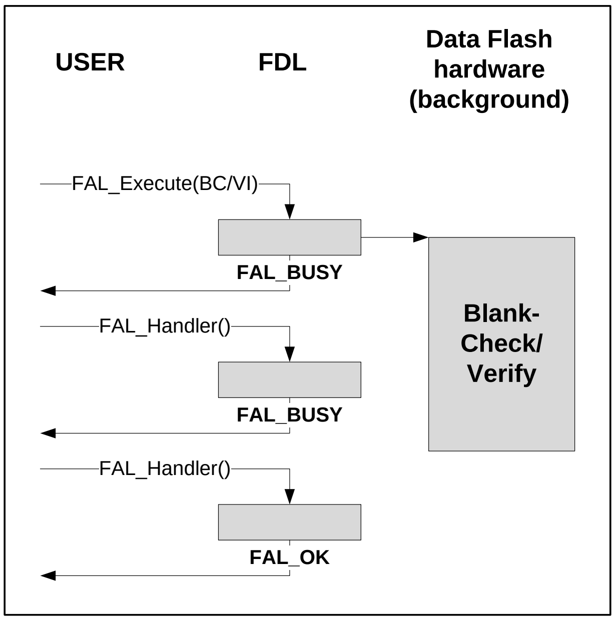
Figure 2-6 Background operation (write)



2.5.3 Background operation (blank-check/verify)

Same procedure as for erase the verify or blank-check will be performed in background.

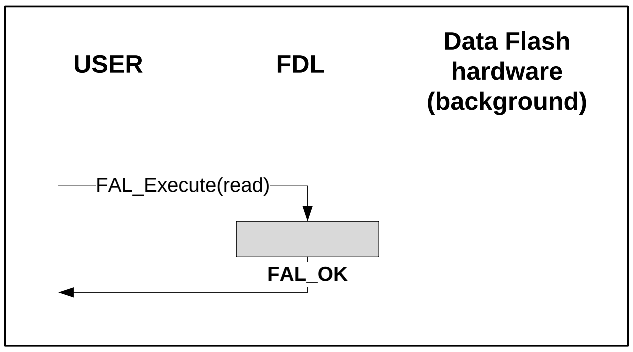
Figure 2-7 Background operation (blank-check/verify)



2.5.4 No background operation for read command

The read command doesn't use the background operation. It's directly finished after the request acceptance.

Figure 2-8 No background operation for read command



2.6 Suspension of block oriented commands (erase)

In cases of systems working under critical real-time conditions, immediately read/write access to the data is a must. In such cases, separate request variables must be defined for word command accesses and block command accesses. Both types of access are managed separately on FDL and requester side.

The suspension and resumption of the running block command (erase) is managed automatically according to the following rules

- word commands cannot be suspended
- each block command can be suspended by any word command
- User requested block command does always suspend running block commands of the EEL
- EEL requested block commands cannot suspend running user block command

In other words:

- word commands have always higher priority than block commands
- user access have always higher priority than EEL (running in background)

The following table shows dependencies between running and requested commands of EEL or user.

Table 1 Block command suspension rules

command acceptance		running command			
		WCMD (eel)	**WCMD (user)**	**BCMD (eel)**	**BCMD (user)**
requested command	**WCMD (eel)**	rejected	rejected	suspend/ resume***	suspend/ resume
	WCMD (user)	rejected	rejected	suspend/ resume	suspend/ resume***
	BCMD (eel)	rejected	rejected	rejected	rejected
	BCMD (user)	rejected	rejected	suspend/ resume	rejected

Agenda:
WCMD = word command
BCMD = block command,
rejected = requested command is rejected
suspend = running block command is suspended

*** when the address of the WCMD refers to the block addressed by BCMD the WCMD will be rejected too.

Chapter 3 User interface (API)

3.1 Run-time configuration

During runtime the configuration of the FDL can be changed dynamically. To be able to do it more than one descriptor constant has to be defined by the user in advance. Depends on the application mode different descriptors can be chosen for the FAL_Init(...) function.

```
/*    ................... */
/*    some code           */
/*    ................... */

/* load standard descriptor */
my_status=FAL_Init(&fal_descriptor_str);

/*    ................... */
/*    some code           */
/*    ................... */

FAL_Close();                /* close USER part of the FAL pool */
EEL_Close();                /* close EEL  part of the FAL pool */
                            /*  - but only if necessary, means */
                            /*    if EEL used in system        */

/* load alternative descriptor     */
my_status=FAL_Init(&fal_descr_2_str);

/*    ................... */
/*    some code           */
/*    ................... */
```

Note: Before changing FAL pool configuration by using of different FAL pool-descriptor the user has to close the FAL (USER part of the pool) in any case. In case that the EEL is active in the system, the EEL part of the FAL-pool has to be closed by using EEL_Close() too.

3.2 Data types

This chapter describes all data definitions used by the FDL.

3.2.1 Library specific simple type definitions

This type defines simple numerical type used by the library

```
typedef unsigned char                            fal_u08;
typedef unsigned int                             fal_u16;
typedef unsigned long int                        fal_u32;
```

3.2.2 Enumeration type “fal_command_t”

This type defines all codes of available commands

```
typedef   enum  {
                  FAL_CMD_UNDEFINED             = (0x00),
                  FAL_CMD_BLANKCHECK_WORD       = (0x00 | 0x01),
                  FAL_CMD_IVERIFY_WORD          = (0x00 | 0x02),
                  FAL_CMD_READ_WORD             = (0x00 | 0x03),
                  FAL_CMD_WRITE_WORD            = (0x00 | 0x04),
                  FAL_CMD_ERASE_BLOCK           = (0x00 | 0x05),
                } fal_command_t;
```

Code value description:

FAL_CMD_UNDEFINED	- default value
FAL_CMD_BLANKCHECK_WORD	- blank-check of 1 Data Flash word
FAL_CMD_IVERIFY_WORD	- verify of 1 Data Flash word
FAL_CMD_READ_WORD	- read 1 Data Flash word
FAL_CMD_WRITE_WORD	- write 1 Data Flash word
FAL_CMD_ERASE_BLOCK	- erases 1 Data Flash block

3.2.3 Enumeration type “ fal_status_t”

This enumeration type defines all possible status- and error-codes can be generated during data-flash access via the FDL. The FAL_OK and FAL_BUSY status are returned to the requester during normal operation. Other codes signalize problems.

```
typedef   enum  {
                  /* operation related status --------------*/
                  FAL_OK                        = (0x00),
                  FAL_BUSY                      = (0x00 | 0x01),

                  /* run-time error related status ---------*/
                  FAL_ERR_PROTECTION            = (0x10 | 0x00),
                  FAL_ERR_BLANKCHECK            = (0x10 | 0x01),
                  FAL_ERR_VERIFY                = (0x10 | 0x02),
                  FAL_ERR_WRITE                 = (0x10 | 0x03),
                  FAL_ERR_ERASE                 = (0x10 | 0x04),

                  /* configuration error related status ----*/
                  FAL_ERR_PARAMETER             = (0x20 | 0x00),
                  FAL_ERR_CONFIGURATION         = (0x20 | 0x01),
                  FAL_ERR_INITIALIZATION        = (0x20 | 0x02),
                  FAL_ERR_COMMAND               = (0x20 | 0x03),
                  FAL_ERR_REJECTED              = (0x20 | 0x04)
                } fal_status_t;
```

Status value	Description
FAL_OK	default value, ready, no error detected
FAL_BUSY	request is accepted and is being processed
FAL_ERR_PROTECTION	access outside permitted pool area
FAL_ERR_BLANKCHECK	specified flash-word is not blank
FAL_ERR_VERIFY	specified flash-word could not be verified
FAL_ERR_WRITE	write is failed
FAL_ERR_ERASE	block erase is failed
FAL_ERR_PARAMETER	not relevant for the FDL (defined for future improvements)
FAL_ERR_CONFIGURATION	Wrong values configured in descriptor
FAL_ERR_INITIALIZATION	FDL not initialized or not opened
FAL_ERR_COMMAND	wrong command code used
FAL_ERR_REJECTED	when FDL busy with another request

3.2.4 Structured type “fal_request_t”

This type is used for definition of request variables and used for information exchange between the application and the FDL. A request variable is passed to the FDL to initiate a command and can be used by the requester (EEL, application...) to check the status of its execution.

```
/* FAL request type (base type for any FAL access) */
typedef struct {
                  fal_u32             data_u32;
                  fal_u16             index_u16;
                  fal_command_t       command_enu;
                  fal_status_t        status_enu;
                } fal_request_t;
```

Struct member	Description
data_u32	32-bit buffer for data exchange during read/write access
index_u16	virtual word index within the targeted pool
command_enu	command code
status_enu	request status code (feedback)

3.2.5 Structured type “fal_descriptor_t”

This type defines the structure of the FDL descriptor. It contains all characteristics of the FDL. It is used in the fdl_descriptor.c file for definition of the ROM constant fal_descriptor_str.

Based on configuration data inside the fdl_descriptor.h the initialization data of descriptor constant is generated automatically in the fdl_descriptor.c.

```
/* FAL descriptor type */
typedef struct {
                fal_u32     fal_pool_first_addr_u32;
                fal_u32     eel_pool_first_addr_u32;
                fal_u32     user_pool_first_addr_u32;
                fal_u32     fal_pool_last_addr_u32;
                fal_u32     eel_pool_last_addr_u32;
                fal_u32     user_pool_last_addr_u32;
                fal_u16     fal_pool_first_block_u16;
                fal_u16     eel_pool_first_block_u16;
                fal_u16     user_pool_first_block_u16;
                fal_u16     fal_pool_last_block_u16;
                fal_u16     eel_pool_last_block_u16;
                fal_u16     user_pool_last_block_u16;
                fal_u16     fal_first_widx_u16;
                fal_u16     eel_first_widx_u16;
                fal_u16     user_first_widx_u16;
                fal_u16     fal_last_widx_u16;
                fal_u16     eel_last_widx_u16;
                fal_u16     user_last_widx_u16;
                fal_u16     fal_pool_wsize_u16;
                fal_u16     eel_pool_wsize_u16;
                fal_u16     user_pool_wsize_u16;
                fal_u16     block_size_u16;
                fal_u16     block_wsize_u16;
                fal_u08     fal_pool_size_u08;
                fal_u08     eel_pool_size_u08;
                fal_u08     user_pool_size_u08;
                fal_u08     fx_MHz_u08;
                } fal_descriptor_t;
```

Struct member	Description
fal_pool_first_addr_u32	first physical address of the FAL pool
eel_pool_first_addr_u32	first physical address of the EEL pool
user_pool_first_addr_u32	first physical address of the USER pool
fal_pool_last_addr_u32	last physical address of the FAL pool
eel_pool_last_addr_u32	last physical address of the EEL pool
user_pool_last_addr_u32	last physical address of the USER pool
fal_pool_first_block_u16	first virtual block of the FAL pool
eel_pool_first_block_u16	first virtual block of the EEL pool
user_pool_first_block_u16	first virtual block of the USER pool
fal_pool_last_block_u16	last virtual block of the FAL pool
eel_pool_last_block_u16	last virtual block of the EEL pool
user_pool_last_block_u16	last virtual block of the USER pool
fal_first_widx_u16	first virtual word-index inside the FAL pool
eel_first_widx_u16	first virtual word-index inside the EEL pool
user_first_widx_u16	first virtual word-index inside the USER pool
fal_last_widx_u16	last virtual word-index inside the FAL pool
eel_last_widx_u16	last virtual word-index inside the EEL pool
user_last_widx_u16	last virtual word-index inside the USER pool
fal_pool_wsize_u16	size of the FAL pool expressed in words
eel_pool_wsize_u16	size of the EEL pool expressed in words
user_pool_wsize_u16	size of the USER pool expressed in words
block_size_u16	size of one Data Flash block expressed in bytes
block_wsize_u16	size of one Data Flash block expressed in words
fal_pool_size_u08	size of the FAL pool expressed in blocks (Note 1)
eel_pool_size_u08	size of the EEL pool expressed in blocks (Note 2, 3)
user_pool_size_u08	size of the USER pool expressed in blocks (Note 2, 3)
fx_MHz_u08	not relevant for the FDL

Note 1: the maximal number of fal_pool_size_u08 is 0x08 (fixed to 78K0R/Fx3, 8x2KByte blocks)
Note 2: the sum of eel_pool_size_u08 and user_pool_size_u08 must not exceed fal_pool_size_u08.
Note 3: both descriptor configuration conditions will be checked by FAL_Init(...)

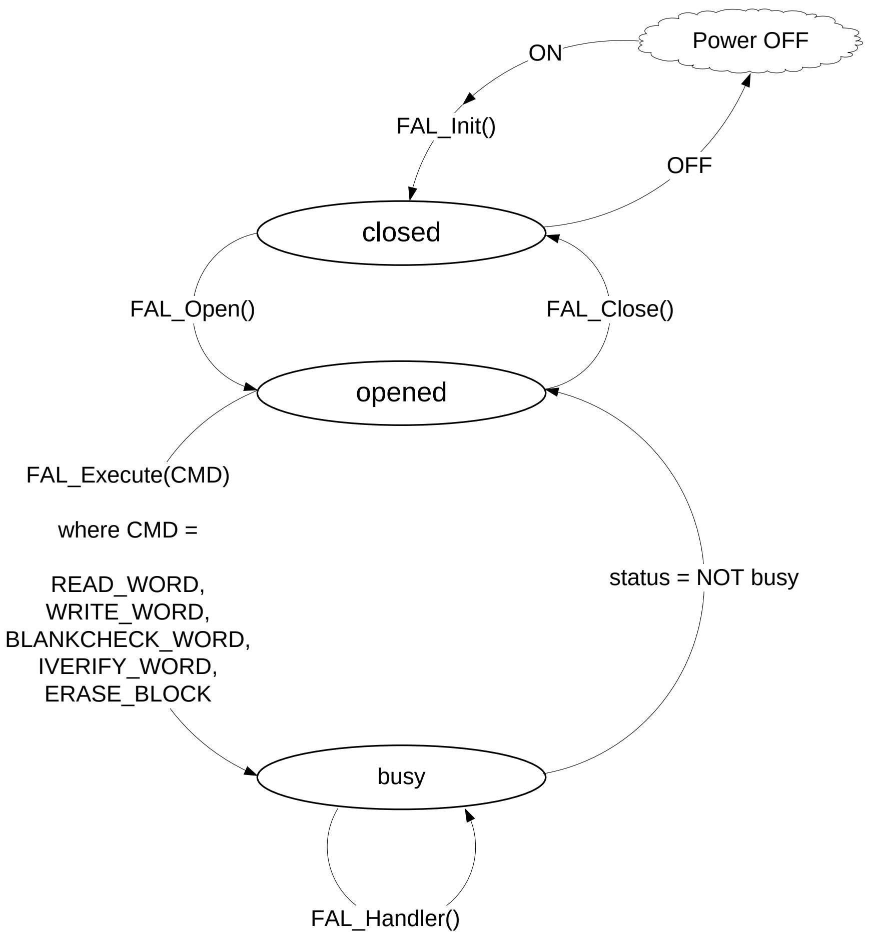
3.3 Functions

Due to the request oriented interface of the FDL the functional interface is very narrow. Beside the initialization function and some administrative function the whole flash access is concentrated to two functions only: FAL_Execute(...) and FAL_Handler().

3.3.1 Basic functional workflow

To be able to use the FDL (execute pool-related commands) in a proper way the requester has to follow a specific startup and shutdown procedure.

Figure 3-1 Basic workflow flow



3.3.2 Interface functions

The interface functions create the functional software interface of the library. They are prototyped in the header file fdl.h.

3.3.2.1 FAL_Init

Description

Initialization of all internal data.

Interface (REC version)

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

Interface (IAR version)

```
__far_func fal_status_t FAL_Init(const __far fal_descriptor_t
                                        __far* descriptor_pstr);
```

Pre-condition

None

Post-condition

Initialization is done.

Argument

Argument	Type	Description
descriptor_pstr	fal_descriptor	Pointer to the descriptor (describing the FDL configuration). The virtualization of the data-flash address-room is done based on that descriptor. The user can use different descriptors to switch between different FDL-pool configurations.

Return types/values

Argument	Type	Description
fal_status	fal_status_t	FAL_ERR_CONFIGURATION when descriptor data are not plausible. FAL_OK when descriptor correct and initialization successful.

Usage

```
fal_status_t my_status;

my_status = FAL_Init(&fal_descriptor_str);

if(my_status == FAL_OK)
{
  /* FDL can be used /
}
else
{
  / error handler */
}
```

3.3.2.2 FAL_Open

Description

This function must be used by the application to activate the data-flash.

Interface (REC version)

```
void __far FAL_Open(void);
```

Interface (IAR version)

```
__far_func void FAL_Open(void);
```

Pre-condition

FAL_Open() does not check any precondition, but FAL_Init(...) has to be executed successfully already.

Post-condition

Data flash clock is switched on.

Argument

Argument	Type	Description
None		

Return types/values

Argument	Type	Description
None		

Usage

```
FAL_Open();
```

3.3.2.3 FAL_Close

Description

This function deactivates the data flash.

Interface (REC version)

```
void  __far FAL_Close(void);
```

Interface (IAR version)

```
__far_func void   FAL_Open(void);
```

Pre-condition

None

Post-condition

Data flash clock is switched off. In case of FAL and EEL usage both FAL_Close and EEL_Close must be called for switching off the Data Flash.

Argument

Argument	Type	Description
None		

Return types/values

Argument	Type	Description
None		

Usage

```
FAL_Close();
```

3.3.2.4 FAL_Execute

Description

This is the main function of the FDL the application can use to initiate execution of any command. Please refer to the chapter “Operation” for detailed explanation of each command.

Interface (REC version)

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

Interface (IAR version)

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

Pre-condition

FAL_Init() executed successfully with status FAL_OK.

FAL_Open() executed already.

Post-condition

None

Argument

Argument	Type	Description
request_pstr	fal_request_t	This argument defines the command which should be executed by FDL. It is a request variable which is used for bi-directional information exchange before and during execution between FDL and the application.

Return types/values

Argument	Type	Description
None		

Usage

```
__near fal_request_t       my_fal_WCMD_request_str;

my_fal_WCMD_request.data_u32 = 0x12345678;
my_fal_WCMD_request.index_u16 = 0x0123;
my_fal_WCMD_request.command_enu = FAL_CMD_WRITE_WORD;

/* command initiation */
do {
     FAL_Execute(&my_fal_WCMD_request);
     FAL_Handler();   /* proceed background process */
} while (my_fal_WCMD_request.status_enu == FAL_ERR_REJECTED);

/* command execution */
do {
     FAL_Handler();
} while (my_fal_WCMD_request.status_enu == FAL_BUSY);
if(my_fal_WCMD_request.status_enu != FAL_OK) error_handler();
```

3.3.2.5 FAL_Handler

Description

This function is used by the application to proceed the execution of a command running in the background. In case of the FDL the functionality of the Handler is reduce to simple status polling of the sequencer. In case any background command was suspended in the past, the FAL_Handler takes care for the resume-process.

Interface (REC version)

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

Interface (IAR version)

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

Pre-condition

FAL_Init() executed successfully with status FAL_OK.

FAL_Open() executed already.

Post-condition

In case of finished command the status is written to the request structure.

Argument

Argument	Type	Description
None		

Return types/values

Argument	Type	Description
None		

Usage

```
/* infinite scheduler loop */
do {
       /* proceed potential command execution */
       FAL_Handler();

       /* 20ms time slize (potential FAL requester) */
       MyTask_A(20);

       /* 10ms time slize (potential FAL requester) */
       MyTask_B(10);

       /* 40ms time slize (potential FAL requester) */
       MyTask_C(40);

       /* 10ms time slize (potential FAL requester) */
       MyTask_D(10);
} while (true);
```

3.3.2.6 FAL_GetVersionString

Description

This function provides the internal version information of the used library.

Interface (REC version)

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

Interface (IAR version)

```
__far_func fal_u08 __far*   FAL_GetVersionString(void);
```

Pre-condition

None

Post-condition

None

Argument

Argument	Type	Description
None		

Return types/values

Argument	Type	Description
	fal_u08 __far*	Pointer to the first character of a zero terminated version string.

Usage (REC version)

```
__far const fal_u08 *my_version_string;

my_version_string = FAL_GetVersionString();
```

Usage (IAR version)

```
fal_u08 __far* my_version_string;

my_version_string = FAL_GetVersionString();
```

Chapter 4 Operation

4.1 Blank-check

The blank-check operation can be used to check if all bits within the addressed pool-word are still “erased”. The user can use blank-check command freely. The blank-check command is initiated by FAL_Execute() and must be continued by FAL_Handler() as long as command is not finished (request-status updated).

Table 2 Status of FAL_CMD_BLANKCHECK_WORD command

Status	Class	Background and Handling	
FAL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	Initialize and open FDL before using it
FAL_ERR_PROTECTION	heavy	meaning	request cannot be accepted
		reason	word index is outside the corresponding pool
		remedy	set correct word index and try again
FAL_ERR_REJECTED	normal	meaning	FDL driver cannot accept the request
		reason	FDL driver is busy with an other word command or block command (in case of same block).
		remedy	Call FAL_Handler as long as request isn’t accepted.
FAL_ERR_BLANKCHECK	normal	meaning	specified flash-word is not blank
		reason	any bit in the flash word addressed by word index isn’t erased
		remedy	nothing, free interpretation at requester side
FAL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FAL_Handler until status changes
FAL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	nothing

4.2 Internal verify

The internal verify operation can be used to check if all bits (0's and 1's) are electronically correct written. Inconsistent and weak data caused by asynchronous RESET can be detected by using the verify command. The user can uses verify freely to check the quality of user data. The verify command is initiated by FAL_Execute() and must be continued by FAL_Handler() as long as command is not finished (request-status updated).

Table 3 Status of FAL_CMD_IVERIFY_WORD command

Status	Class	Background and Handling	
FAL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	Initialize and open FDL before using it
FAL_ERR_PROTECTION	heavy	meaning	request cannot be accepted
		reason	word index is outside the corresponding pool
		remedy	set correct word index and try again
FAL_ERR_REJECTED	normal	meaning	FDL driver cannot accept the request
		reason	FDL driver is busy with an other word command or block command (in case of same block).
		remedy	Call FAL_Handler as long as request isn't accepted.
FAL_ERR_VERIFY	normal	meaning	specified flash-word in pool could not be verified
		reason	any bit in the addressed flash word isn't electrically correct
		remedy	nothing, free interpretation at requester side
FAL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FAL_Handler until status changes
FAL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	nothing

4.3 Read

The read operation can be used to read the content of the addressed pool-word. It is initiated and finished directly by FAL_Execute(). FAL_Handler() is not needed in that case.

Table 4 Status of FAL_CMD_READ_WORD command

Status	Class	Background and Handling	
FAL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	Initialize and open FDL before using it
FAL_ERR_PROTECTION	heavy	meaning	FDL driver cannot accept the request
		reason	FDL driver is busy with an other word command or block command (in case of same block).
		remedy	Call FAL_Handler as long as request isn't accepted.
FAL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	nothing

Caution:
During the execution of a read command the DMAs (via the SFR DMCALL.DWAITALL) and interrupts are disabled for a short period (see Section 6.2.4).
The reason for this originates from the following situation: In case a Data Flash Read access is performed exactly at the same time while any DMA transfer is triggered, there is a possibility for an internal bus conflict between CPU bus and Data Flash bus. Such kind of bus conflict can cause a wrong data to be read from the Data Flash. (See also Customer Notification R01TU0003ED0103.)

4.4 Write

The write operation writes 32-bit data into passed word index. To protect existing flash data against accidental overwrite 1-word blank-check is executed in advance. After that the write-command is initiated. In case of successfully finished writing the quality of data will be checked via internal verify.

Table 5 Status of FAL_CMD_WRITE_WORD command

Status	Class	Background and Handling	
FAL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	Initialize and open FDL before using it
FAL_ERR_PROTECTION	heavy	meaning	request cannot be accepted
		reason	word index is outside the corresponding pool
		remedy	set correct word index and try again
FAL_ERR_REJECTED	normal	meaning	FDL driver cannot accept the request
		reason	FDL driver is busy with an other word command or block command (in case of same block).
		remedy	Call FAL_Handler as long as request isn't accepted.
FAL_ERR_BLANKCHECK	normal	meaning	specified flash-word in pool is not blank, write was not performed, the content of flash-word remains untouched
		reason	overwriting of non-erased flash words is not allowed
		remedy	erase the block before writing again into this block
FAL_ERR_WRITE	normal	meaning	flash word addressed by word index couldn't be written correctly after performing the max. number of retries
		reason	flash problems
		remedy	erase the block and try to write again into this block
FAL_ERR_VERIFY	normal	meaning	after writing the data the flash word could not be verified
		reason	flash problems
		remedy	erase the block and try

			to write again into this block
FAL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FAL_Handler until status changes
FAL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	nothing

4.5 Erase

The erase operation can be used to erase one block of the related pool. After starting the erase-command the hardware is checking if the addressed block is already blank to avoid unnecessary erase cycles. After that the erase-command is initiated. The max. number of erase retries is 19.

Table 6 Status of FAL_CMD_ERASE_BLOCK command

Status	Class	Background and Handling	
FAL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	Initialize and open FDL before using it
FAL_ERR_PROTECTION	heavy	meaning	request cannot be accepted
		reason	block number outside the corresponding pool
		remedy	correct block number and try again
FAL_ERR_REJECTED	normal	meaning	FDL driver cannot accept the request
		reason	FDL driver is busy with an other word command or block command (in case of same block).
		remedy	Call FAL_Handler as long as request isn't accepted.
FAL_ERR_ERASE	fatal	meaning	specified flash block could not be erased
		reason	internal flash problems
		remedy	do not use this block anymore
FAL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FAL_Handler until status changes
FAL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	nothing

Chapter 5 FDL usage by user application

5.1 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 especially subchapter “Cautions” of chapter “Characteristics”.

5.2 Special considerations

5.2.1 Reset consistency

During the execution of FDL commands a reset could occur and the data could be damaged. In such cases it should be considered whether to uses two variables for same data and so on. In other words please consider such reset scenarios to avoid invalid data. The EEL provided by Renesas Electronics is designed to avoid read of invalid data cause by such reset scenarios. The following chapter describes the applications where the EEL should be used.

5.2.2 EEL+FDL or FDL only

Depending on the security level of the application, write frequency of variables and variables count it should be considered whether to uses the EEL+FDL or the FDL only.

5.2.2.1 FDL only

By using the FDL only the application has to take care about all reset scenarios and writing flow of different variables with different sizes.

Application scenarios

- Programming of initial or calibration data
- user specific EEPROM emulation

5.2.2.2 EEL+FDL

The duo of EEL and FDL allows the user to uses the EEL for high write frequency of different variables with different sizes in a secure way and additionally the USER pool is available for free usage.

Application scenarios

- Programming of initial or calibration data
- Large count of variables and high write frequency by using the EEL
- Secure data handling completely handled by EEL

5.3 File structure

5.3.1 Library for IAR Compiler

[root]	FDL library
fdl_info.txt	Library release notes
[root]\[lib]	
fdl.h	FDL interface definition
fdl_types.h	FDL types definition
fdl.r26	Pre-compiled library
[root]\[smp][C]	
fdl_descriptor.c	Descriptor calculation part
fdl_descriptor.h	Pool configuration part
fdl_sample_linker_file.xcl	Sample Linker file

5.3.2 Library for REC Compiler

[root]	
fdl_info.txt	Library release notes
[root]\[lib]	FDL library
fdl.h	FDL interface definition (Compiler)
fdl.inc	FDL interface definition (Assembler)
fdl_types.h	FDL types definition
fdl.lib	Pre-compiled library
[root]\[smp][C]	Sample folder for C-Compiler projects
fdl_descriptor.c	Descriptor calculation part
fdl_descriptor.h	Pool configuration part
fdl_sample_linker_file.dr	Sample Linker file
[root]\[smp][asm]	Sample folder for Assembler projects
fdl_descriptor.asm	Descriptor calculation part
fdl_descriptor.inc	Pool configuration part
fdl_sample_linker_file.dr	Sample Linker file

5.4 Configuration

5.4.1 Linker sections

Following segments are defined by the library and must be configured via the linker description file.

FAL_CODE	Segment for library code. Can be located anywhere in the code flash.
FAL_CNST	Segment for library constants like descriptor. Can be located anywhere in the code flash.
FAL_DATA	Segment for library data. Must be located inside the SADDR RAM

NOTE: FAL_CODE and FAL_CNST segments must be located anywhere in the Code Flash but inside the same 64 KByte page.

5.4.2 Descriptor configuration (partitioning of the data flash)

Before the FDL can be used the FDL pool and it's partitioning has to be configured first. The descriptor is defining the physical/virtual addresses and parameter of the pool which will be automatically calculated by using the FAL_POOL_SIZE and EEL_POOL_SIZE definition.

Because the physical starting address of the data flash is fixed by the hardware the user can only determine the total size of the pool expressed in blocks. Also the physical size of the pool is limited by the hardware and must not be defined by the user. Also the physical size of a flash block is a predefined constant determined by the used hardware.

The first configuration parameter is FAL_POOL_SIZE. The minimum value is 0 and means any access to the FDL-pool is closed. The maximum value in case of 78K0R/Fx3 is 8 (means 8 blocks = 16 Kbytes).

The other configuration parameter is EEL_POOL_SIZE, the size of the EEL-pool within the FDL-pool used exclusively for Renesas EEPROM emulation library only. The minimum size of the EEL-pool is 0. This means the complete FDL pool is occupied by the user for storing data. But also when a proprietary EEPROM emulation is implemented by the user the complete pool has to be reserved for it by specifying EEL_POOL_SIZE=0. The maximum size of the EEL-pool is FAL_POOL_SIZE.

Notes:

- The USER pool and EEL pool are complementary. This means: the USER pool is always the remaining none-EEL-pool (in other words USER_POOL_SIZE = FAL_POOL_SIZE – EEL_POOL_SIZE).

- The virtual address 0 of the user-pool corresponds with the successor of the last EEL-pool word.

5.4.3 Request structure

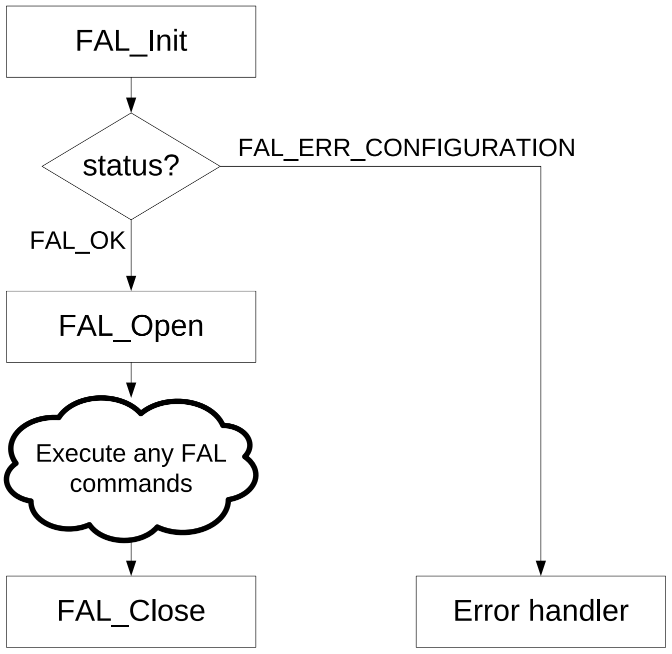
Depending on the user application architecture more than one request variable could be necessary. For example if an immediate write is necessary during running erase. In such a case two request variables (one for write and one for erase) are necessary. Please take care that each request variable is located on an even address.

5.5 General flow

5.5.1 General flow: Initialization

The following figure illustrates the initialization flow.

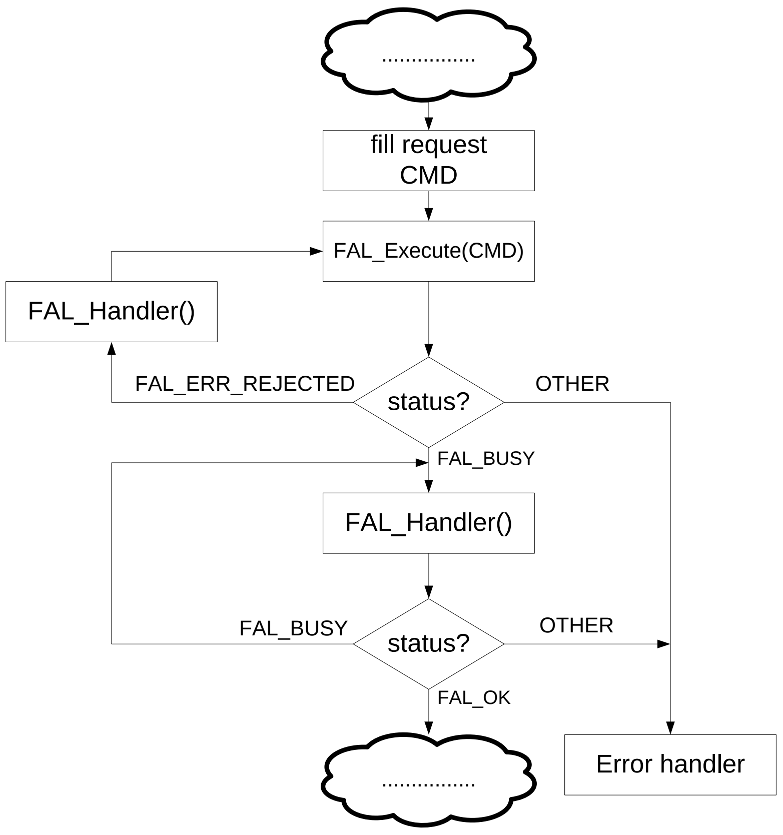
Figure 5-1 Initialization flow



5.5.2 General flow: commands except read

After initialization of the environment the application can uses the commands provided by the library. The following figure illustrates the general flow of command (except read command) execution.

Figure 5-2 FAL command execution (except read command)

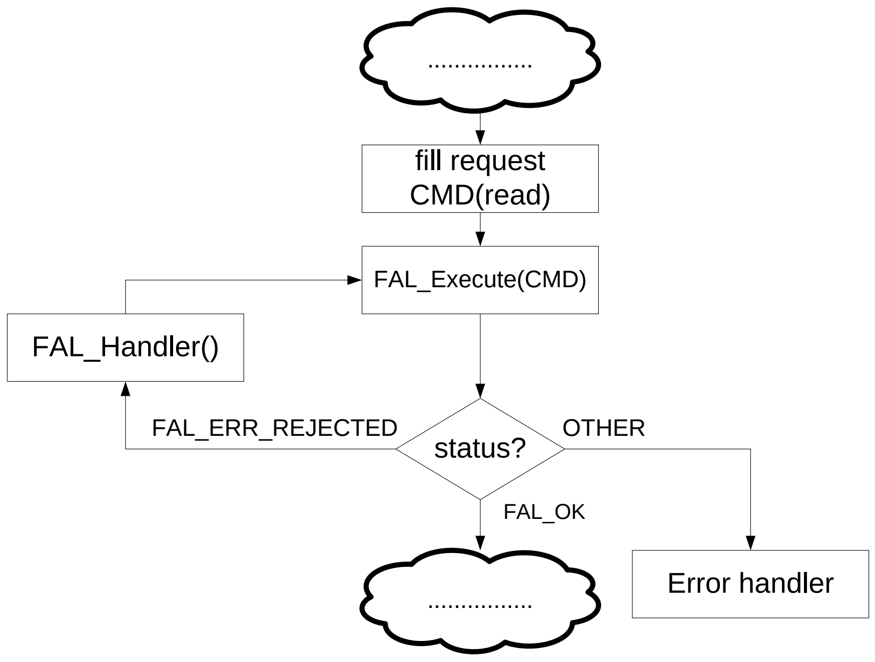


In case the requested command is rejected the application has to call the FAL_Handler() for finishing/suspend the background command and try to execute the command again.

5.5.3 General flow: read command

The difference between the read command and other commands (erase/write/verify/blank-check) is that the read command will be completed directly during FAL_Execute() function. That means no additionally FAL_Handler() calls are required.

Figure 5-3 FAL read command execution



In case the requested command is rejected the application has to call the FAL_Handler() for finishing/suspend the background command and try to execute the command again.

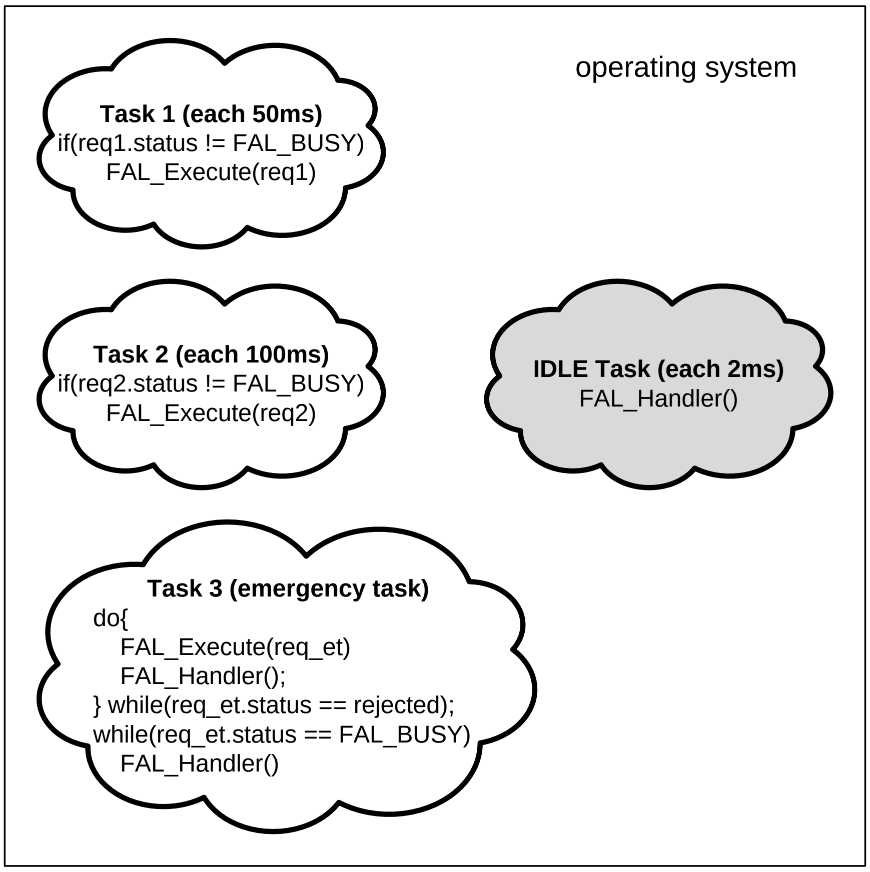
5.6 Example of FDL used in operating-systems

The possibility of background operation and request-response structure of the FDL allows the user to uses the FDL in an efficient way in operating systems.

Note: Please read the chapter “Characteristics->Cautions” carefully before using the FDL in such operating systems.

The following figure illustrates a sample operating system where the FDL is used for Data Flash access.

Figure 5-4 FDL used in an operating system



This sample operating system shows three different task types which are described below.

Task 1 and Task 2

This task type is a requesting task like Task 1 and 2. Such tasks just start any FDL command via the FAL_Execute function and assume that it will be finished in the background via the IDLE task.

IDLE task

The IDLE task will be used by the application for continuing any running FAL command. That means the FAL_Handler must be called inside of such a task.

Emergency task

The difference between this task type and the requesting type (Task 1 and

Task 2) is that this task performs any FAL commands completely without waiting in the background. Such task can be used in case of voltage drop where important data must be saved before the device is off.

5.7 Example: Simple application

The following sample shows how to use each command in a simple way.

```
extern __far const fal_descriptor_t  fal_descriptor_str;
fal_status_t                         my_fal_status_enu;
__near fal_request_t                 request;


/* initialization */
my_fal_status_enu = FAL_Init(
               (__far fal_descriptor_t*)&fal_descriptor_str );

if(my_fal_status_enu != FAL_OK) ErrorHandler();
FAL_Open();


/* erase block 0 */
request.index_u16   = 0x0000;
request.command_enu = FAL_CMD_ERASE_BLOCK;
FAL_Execute(&request);
while(request.status_enu == FAL_BUSY) FAL_Handler();
if(request.status_enu != FAL_OK)      ErrorHandler();

/* write patter 0x12345678 into the widx = 0 */
request.index_u16   = 0x0000;
request.data_u32    = 0x12345678;
request.command_enu = FAL_CMD_WRITE_WORD;
FAL_Execute(&request);
while(request.status_enu == FAL_BUSY) FAL_Handler();
if(request.status_enu != FAL_OK)      ErrorHandler();

/* read value of widx = 0 */
request.index_u16   = 0x0000;
request.command_enu = FAL_CMD_READ_WORD;
FAL_Execute(&request);
if(request.status_enu != FAL_OK) ErrorHandler();

/* check whether the written pattern is correct */
if(request.data_u32 != 0x12345678) ErrorHandler();

/* blank check widx = 0 */
request.index_u16   = 0x0000;
request.command_enu = FAL_CMD_BLANKCHECK_WORD;
FAL_Execute(&request);
while(request.status_enu == FAL_BUSY) FAL_Handler();
if(request.status_enu != FAL_ERR_BLANKCHECK) ErrorHandler();

/* verify widx = 0 */
request.index_u16   = 0x0000;
request.command_enu = FAL_CMD_IVERIFY_WORD;
FAL_Execute(&request);
while(request.status_enu == FAL_BUSY) FAL_Handler();
if(request.status_enu != FAL_OK) ErrorHandler();

FAL_Close();
```

5.8 Example: Read/Write during background erase

The FDL allows background erase operation, therefore during that time read- and write-access to data located in another block of the addressed pool is possible. To be able to use foreground read/write operation a separate request variable has to be declared for that purpose. Read and write commands do always suspend the erase process running in the background. Exception is when the word command tries to access the same block as the running erase in background. In such a case the FAL_Handler() has to be called until the running erase command is finished. Please refer to the detailed explanation of command suspension to chapter "Suspension of block oriented commands (erase)".

```
fal_request_t   my_BCMD_req, my_WCMD_req;
fal_u32         my_data_u32;

void erase_state_0(void)
{
  /* specify the BCMD parameter */
  my_BCMD_req.index_u16    = 4;
  my_BCMD_req.command_enu = FAL_CMD_ERASE_BLOCK;

  FAL_Execute(&my_BCMD_req);

  /* if erase-request accepted goto next state 1 */
  /* if erase-request rejected remain in state 0 */
  /* if erase-request error occurs goto error-state */

  if(my_BCMD_req.status_enu == FAL_BUSY;)
    next_state = erase_state_1;
  else
  {
    if (my_BCMD_req.status_enu != FAL_ERR_REJECTED)
       next_state = erase_state_err;
  }
}

/* block erase is running in background here */
void erase_state_1(void)
{
  /* if read during erase needed, read immediately */
  if(emergency_read==TRUE)
  {

    do {
      my_WCMD_req.index_u16    = 234;
      my_WCMD_req.command_enu = FAL_CMD_READ_WORD;
      FAL_Execute(&my_WCMD_req);

      FAL_Handler();  /* enforce eventually blocking command */

    } while((my_WCMD_req.status_enu==FAL_ERR_REJECTED));

    /* read-request accepted -> read the data directly */
    if (my_WCMD_req.status_enu==FAL_OK)
      my_data_u32 = my_WCMD_req.data_u32;
    else
    {
      /* in case of error, goto error-state */
      next_state = erase_state_err;
    }

  } /* ########### NEXT PAGE -----> ############### */
```

```
    /* if write during erase needed, read immediately */
    if(emergency_write==TRUE)
    {
      do {
          my_data_u32 = 0x12345678;
          my_WCMD_req.data_u32 = my_data_u32;
          my_WCMD_req.index_u16   = 234;
          my_WCMD_req.command_enu = FAL_CMD_WRITE_WORD;
          FAL_Execute(&my_WCMD_req);

          FAL_Handler();/* enforce eventually blocking command */

      } while((my_WCMD_req.status_enu==FAL_ERR_REJECTED));

      /* enforce execution of the write-request */
      do {
          FAL_Handler();
      } while((my_WCMD_req.status_enu==FAL_BUSY));

      /* if error during write -> goto error-state */
      if (my_WCMD_req.status_enu!=FAL_OK)
        next_state = erase_state_err;
    }

    /* proceed the BCMD execution */
    FAL_Handler();

    /* erase-request finished -> goto state 2 */
    if(my_BCMD_req.status_enu==FAL_OK))
      next_state = erase_state_2;
    else
    {
      /* in case of error, goto error-state */
      next_state = erase_state_err;
    }
  }
```

Chapter 6 Characteristics

6.1 Resource consumption

Resource consumption		
	REC Compiler	**IAR Compiler**
Max. code size (code flash)	1715 bytes	1749 bytes
Constants (code flash)	62 bytes	62 bytes
Internal data (SADDR RAM)	2 bytes	2 bytes
Max. stack (RAM)	40 bytes	40 bytes

All values are based on FDL version V1.10.

6.2 Timings

The following timings have been measured on the uPD78F1845 device and FDL version V1.10.

6.2.1 Maximum function execution times

Function	Maximum function execution time
FAL_Init	1476/fclk + 14µs
FAL_Open	28/fclk + 89µs
FAL_Close	270/fclk + 13µs
FAL_Execute	1172/fclk + 14µs
FAL_Handler	864/fclk + 3µs
FAL_GetVersionString	14/fclk

6.2.2 Maximum command execution times

	Maximum command execution time				
Command	**2 MHz**	**4 MHz**	**8 MHz**	**16 MHz**	**24 MHz**
Read (1 word)	230µs	116µs	58µs	29µs	20µs
Blank check (1 word)	677µs	339µs	171µs	105µs	70µs
Write (1 word)	2372µs	1595µs	1210µs	1067µs	987µs
Verify (1 word)	684µs	342µs	173µs	105µs	82µs
Erase (1 block)	285473µs	285194µs	285020µs	284932µs	284915µs

6.2.3 Typical command execution times

Command	Typical command execution times				
	2 MHz	4 MHz	8 MHz	16 MHz	24 MHz
Read (1 word)	191µs	96µs	48µs	24µs	16µs
Blank check (1 word)	564µs	282µs	142µs	87µs	58µs
Write (1 word)	1295µs	705µs	383µs	250µs	196µs
Verify (1 word)	570µs	285µs	144µs	87µs	68µs
Erase (1 block)	12695µs	12408µs	12262µs	12190µs	12176µs

6.2.4 Interrupt and DMA disable period

The following table shows the interrupt and DMA disable period for each FAL function/command.

Function/command	Max. interrupt disable period (cycles)	Max. DMA disable period (cycles)
FAL_Init	27	0
FAL_Open	0	0
FAL_Close	27	0
Blank-check command (1 word)	27	0
Verify command (1 word)	27	0
Read command (1 word)	30	20
Write command (1 word)	27	0
Erase command (1 block)	27	0

6.3 Cautions

Following cautions must be considered before developing of an application.

- Library code and constants must be located completely in the same 64k flash page.
- Initialization by FAL_Init must be performed before execution FAL_Handler/FAL_Execute functions.
- Do not read data flash directly (means without FAL) during command execution of FAL
- Each request variable must be located from an even address
- All functions are not re-entrant. That means don’t call FAL functions inside the ISRs while any FAL function is already running.
- Task switches, context changes and synchronization between FDL functions

  All FDL functions depend on FDL global 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:
  - Task 1: 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
- After execution of FAL_Close or FAL_Init function all requested/running commands will be aborted and cannot be resumed. Please take care that all running commands are finished before calling this functions.
- It is not possible to modify the Data Flash parallel to modification of the Code Flash
- Suspension of word commands like read, write, verify, and blank-check is not possible
- During the execution of any FAL function, the interrupts may be disabled for a short period of time (see also Section 6.2.4).
- During the execution of the read command the DMA operation will be disabled for short period (see also Section 6.2.4).

Revision history

This is a first release of the user’s manual.

Chapter	Page	Description
all		Rev. 1.01: Initial document
 4.3 6.1 6.2 6.3	 32 46 46-47 48	Rev. 1.02: Caution added Resource consumption updated Timing measurements updated Cautions regarding interrupt and DMA disable times added

Data Flash Access Library