User Manual

32

EEPROM Emulation Library

Type T01, European Release

RENESAS 32-Bit MCU
RH Family / RH850 Series

Installer:
RENESAS_EEL_RH850_T01Vx.xx

Renesas Electronics
www.renesas.com

09.06.2015

Notice

Regional information

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

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

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

Visit

http://www.renesas.com

to get in contact with your regional representatives and distributors.

Preface

Readers	This manual is intended for users who want to understand the functions of the concerned libraries.	
Purpose	This manual presents the software manual for the concerned libraries.	
Note	Additional remark or tip	
Caution	Item deserving extra attention	
Numeric notation	Binary:	xxxx or xxxB
	Decimal:	xxxx
	Hexadecimal	xxxxH or 0x xxxx
Numeric prefix	Representing powers of 2 (address space, memory capacity):	
	K (kilo)	210 = 1024
	M (mega):	220 = 1024^2 = 1,048,576
	G (giga):	230 = 1024^3 = 1,073,741,824
Register	X, x = don’t care	
Diagrams	Block diagrams do not necessarily show the exact software flow but the functional structure. Timing diagrams are for functional explanation purposes only, without any relevance to the real hardware implementation.	

How to Use This Document

(1) Purpose and Target Readers

This manual is designed to provide the user with an understanding of the hardware functions and electrical characteristics of the MCU. It is intended for users designing application systems incorporating the MCU. A basic knowledge of electric circuits, logical circuits, and MCUs is necessary in order to use this manual. The manual comprises an overview of the product; descriptions of the CPU, system control functions, peripheral functions, and electrical characteristics; and usage notes.

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) Related documents

Document number	Description
R01US0079	FDL User Documentation

(3) List of Abbreviations and Acronyms

Abbreviation	Full form
API	Application Programming Interface
Flash Area	Area of Flash consists of several coherent Flash Blocks
Code Flash	Embedded Flash where the application code or constant data is stored.
CR	Complementary Read
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 behaviour. To gain a similar behaviour some side parameters have to be taken in account.
FDL	Data Flash Library (Data Flash access layer)
Flash	Electrically erasable and programmable nonvolatile memory. The difference to ROM is, that this type of memory can be re-programmed several times.

Abbreviation	Full form
(Physical) 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.
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” - non-volatile memory. The content of that memory cannot be changed.
Serial programming	The on board programming mode is used to program the device with an external programmer tool.
Virtual Flash blocks	The EEL merges together small physical Flash blocks to bigger virtual Flash blocks which are then managed in the ring buffer

Table of Contents

Chapter 1 Introduction

This user manual describes the internal structure, the functionality and the application programming interface (API) of the Renesas RH850 EEPROM Emulation Library (EEL) Type 01, designed for RH850 devices with Data Flash based on the RV40 flash technology.

While RH850 devices are equipped with Data Flash, a direct usage of this non-volatile memory can be more complex than the usage of classical electrically erasable programmable read-only memory (EEPROM), as erases of flash memory can only be performed block-wise, i.e. on rather large continuous address ranges. Furthermore, the number of program-erase cycles for each flash block is finite and demands for effective data management and wear-leveling techniques.

The EEPROM Emulation Library described in this document addresses these challenges by providing a simple-to-use software interface encapsulating the detailed flash management with a framework based on emulated data sets. The developer can access (read and write) these data sets independently in an EEPROM-like manner. This way, the developer can concentrate on the actual functionality of the application rather than spending time on tedious details of Data Flash access sequences.

The Renesas RH850 EEPROM Emulation Library Type 01 (from here on referred to as EEL) is prepared for the Green Hills, IAR and Renesas compiler environments. The distributed versions depend on customer requests. The library is distributed using an installer tool allowing for selection of the appropriate environment. The library is delivered together with device dependent application programs showing the implementation of the libraries and the usage of the library functions.

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

http://www.renesas.eu/updates?oc=EEPROM_EMULATION_RH850.

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

The EEL requires the corresponding RH850 Data Flash Access Library (FDL) Type 01 for operation. It can be obtained from the same URL as the EEL. Please ensure to always use the correct release of the FDL which is specified inside the EEL release in order to avoid incompatibilities between the two libraries.

If support on the library or library usage is required, please contact the Flash support:

application_support.flash-eu@lm.renesas.com

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

1.1 Flash Infrastructure

The flash technology which is utilized in RH850 devices is called RV40. Besides the Code Flash, devices of the RH850 microcontroller family are also equipped with a separate flash area - the Data Flash. This flash area is meant to be used exclusively for data. It cannot be used for instruction execution (code fetching).

1.1.1 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 fetch instruction code from the Code Flash (to execute program) while data are read or written into Data Flash. This allows implementation of EEPROM emulation concepts running quasi in parallel to the application software without significant impact on its execution timing.

If not mentioned otherwise in the device users manuals, RH850 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.

1.1.2 Data Flash Granularity

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

1.1.3 Complementary-Read Data Flash

The Data Flash of RH850 devices is based on a complementary read (CR) mechanism in order to achieve a high endurance (number of erase cycles). Each data bit is stored by means of two Flash cells, which are programmed to opposite voltage levels allowing for reproducing the content by comparison.

Thereby, erased cells provide a very small differential level rather than a clean voltage difference indicating a value. Hence, values read from data flash memory that has been erased but not yet been programmed again are essentially undefined. However, read values have a tendency to match formerly written data. Please consider the following when working with the RH850 Data Flash:

- The EEL handles the CR behavior of erased cells in the library concept. Application wise, no special treatment is necessary when using the EEL.
- When using the FDL directly to access the Data Flash, blank checking needs to be used to confirm that an area is in the non-programmed state (see FDL user manual).
- When inspecting the Data Flash during debugging, the debugger needs to provide additional information on the Flash cell status (erased/written) in order to allow for correct interpretation of the Flash content.

1.1.4 Data Flash Error Correction Code Treatment

The Data Flash content or RH850 devices is protected by means of an error correction code. Reading from an area of the Data Flash memory that has been erased or contains corrupted data (e.g. caused by single event upsets) can lead to the detection of an ECC error and generation of the corresponding exception.

In many situations, the EEL will handle these exceptions and invoke countermeasures. However, there are well-defined situations where the ECC exception is still triggered during EEL library operation. Please refer to Section 5.4.7, “ECC Errors” for detailed information.

1.2 Feature Overview

The EEL for RH850 devices offers easy-to-use EEPROM-like access to user-defined data sets stored in the non-volatile Data Flash memory. Thereby, the EEL offers a rich function set to designers e.g. covering the following features:

- Reset resistance
- Block rotation (balanced data flash usage)
- Applicability in operating systems
- Standby functionality for energy savings
- Immediate and incremental write features

- Early read/write access during start-up
- Normal and limited data set operation mode
- Individually configurable data size
- Configurable usage of parts or the complete Data Flash

Chapter 2 Architecture

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

2.1 Layered Software Architecture

The EEPROM emulation system is built up from several hierarchical functional blocks. This user manual concentrates on the functionality and usage of the EEL. However, a short description of all involved functional blocks and their relationship is important for the general understanding of the concepts and usage of the EEL.

As depicted in Figure 1, the software architecture of the EEPROM emulation system is built up of several layers:

- **Physical flash layer:** 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 from the code flash.
- **Flash access layer:** The Data Flash access layer is represented by the Data Flash Access Library (FDL) provided by Renesas. It offers an easy-to-use API to access and manage the Data Flash by encapsulating and abstracting tedious timing and flash access sequence details.
- **EEPROM layer:** The EEPROM layer allows read/write access to the Data Flash on an abstract level. It is represented by a Renesas EEL (as described in this document) or alternatively any other, user specific implementation.
- **Application layer:** The application layer covers the user's application software which has access to the Data Flash by using functions and commands of the lower layers.

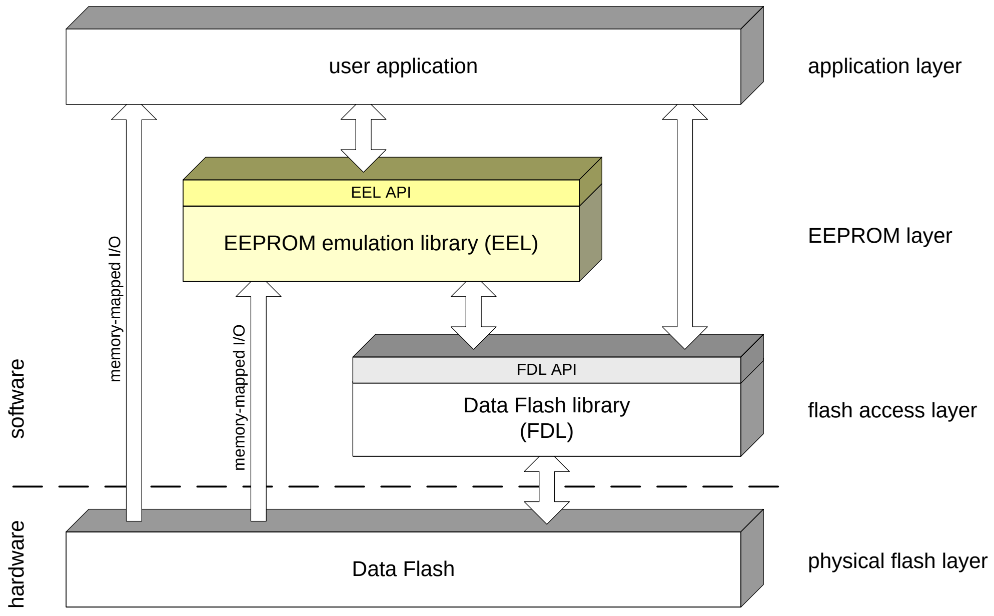


Figure 1: Layers of the EEPROM emulation system

The EEL accesses the Data Flash via the Data Flash Library (FDL). For performance reasons however, there are also direct read accesses to the Data Flash by means of memory-mapped input/output (I/O). Such a direct read access is also possible from the user application (please refer to your device user manual for details).

Please note however, that the Data Flash does not support multiple parallel accesses. It is the duty of the user to ensure an exclusive usage of one of the layers. That means that the user may not access the Data Flash via FDL or memory-mapped I/O while the EEL is in operation. Therefore, the EEL features a standby/wakeup functionality in order to interrupt EEL processing (see also Section 3.9, “Suspend / Resume”).

Depending on the data to be stored, each library has its advantages and disadvantages. While the FDL enables an efficient way to store data which is changed very seldom (e.g. configuration parameters or constant identification numbers) and offers the designer full freedom how to manage the data, the EEL provides a comfortable way to handle frequently changing data sets at the cost of a mean resource overhead.

2.2 Data Flash Pool Structure

The decision for using FDL or EEL as Data Flash access mechanism is usually driven by characteristics of the data, in particular how often the data needs to be updated, reset robustness and the application overhead to access and manage the data. As a result it is often desirable to use both libraries within the same application. Consequently, the Data Flash is separated into individual pools as depicted in Figure 2:

- **FDL pool:** 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 runtime during FDL initialization, while usually the complete Data Flash is selected.
- **EEL pool:** The FDL pool provides the space for the EEL pool which is allocated the FDL pool. The EEL Pool provides the Flash space for the EEL to store the emulation data and management information.
- **User pool:** All FDL pool space not allocated by the EEL pool is freely usable by the user application, so is called the user pool.

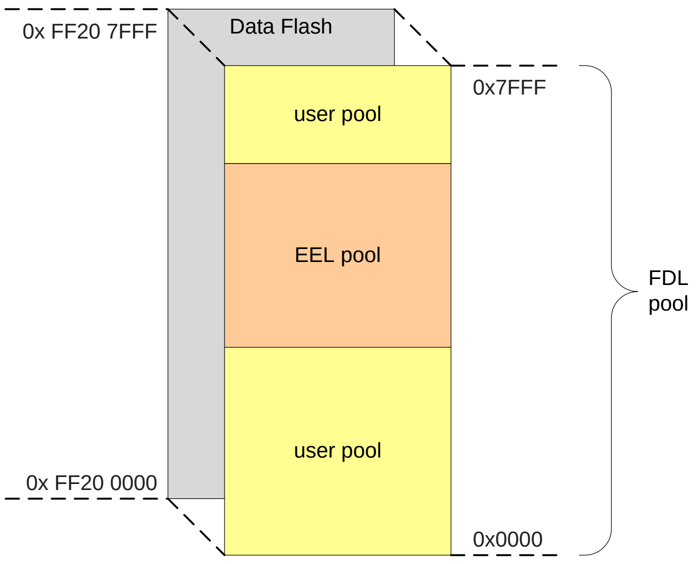


Figure 2: Logical fragmentation of Data Flash in EEL and user pool (example for 32 kB Data Flash)

The separation in EEL and user pool is necessary as the EEL stores additional administrative data in the blocks of the EEL pool for managing the blocks and the instances of variables. In order to avoid FDL handling errors which might corrupt the EEL data structures, the API structure and sanity checks of the FDL ensure that only the user pool is accessed directly from the user application, while the EEL pool is only accessed by the EEL.

The distribution of the FDL pool to EEL and user pool can be configured at library start-up in the descriptor of the FDL (see the RH850 FDL T01 user manual for details of the FDL configuration options). Please note that the assignment of Data Flash memory to user and EEL pool can only be done on a flash-block basis.

2.2.1 User Pool

The user pool allocates the Data Flash blocks that are directly managed by the user application via the FDL. It can be used to simply store constants or to even build up an own user EEPROM emulation. Please note however, that it is the designers duty to take care of reset and failure scenarios himself when using the FDL, e.g. by a proper failure mode and effects analysis.

2.2.1.1 Address Virtualization

In order to simplify the flash content handling, the physical addresses used by the flash hardware were transformed into a linear 16-bit index addressing (8-bit units) inside the user pool (see Figure 2). By this measure, the FDL pool can be treated as a simple array. To address the array elements (read/write access), virtual addresses starting at 0x0000 can be used.

2.2.2 EEL Pool

The EEL pool allocates the Data Flash blocks that are used by the EEPROM emulation to store user content and administrative data. The handling of the flash blocks is completely encapsulated in the EEL and abstracted through the EEL API. The direct access to this pool via the FDL is prohibited.

While the data stored in the user pool needs to be managed completely by the user application, the usage of the EEL pool is greatly simplified by data set (DS) virtualization. The user defines multiple DSs with individual sizes at compile time. These DSs can be read and written via dedicated commands of the EEL API during runtime. Pool handling and reset-safe variable update processes help the developer to concentrate on the actual functionality of the application rather than detailed flash-access sequences.

2.2.2.1 Block Virtualization

The EEL concept relies on Flash blocks of a reasonable size to store block management data and data sets. The physical blocks provided by the RV40 Flash technology (64Bytes erase granularity) are too small to support EEPROM emulation effectively. Therefore, multiple physical Flash blocks are merged to one virtual block. The virtual blocks of the EEL pool are then managed by the EEL as a ring buffer.

The size of the virtual blocks is runtime configurable at initialization of the library. Selecting a proper size for the virtual block is important and depends on the data structure used in the application (see also Section 5.4.1, “EEL Pool Configuration”).

The transition from physical to virtual blocks is done within the EEL near to the FDL interface. As the FDL API handles physical blocks, an EEL internal low level routine converts the virtual blocks to physical blocks before calling the FDL.

The complete EEL operates on virtual blocks. Hence, all pool sizes and alignments need to be configured based on the virtual block size intervals.

2.3 EEL Management

While the simple usage of the EEL via its API hides many complex management issues from the developer, it is still mandatory to understand the management mechanisms of the library in order to use it efficiently. Therefore, the EEL management is introduced in the following by first specifying the structures of the EEL pool, blocks and data sets. Afterwards, the transitions, sequences and processes within the EEL are highlighted briefly.

2.3.1 EEL Pool Structure

As introduced earlier, the EEL organizes the EEL pool in virtual blocks. The virtual blocks are used as a kind of ring buffer as depicted in Figure 3.

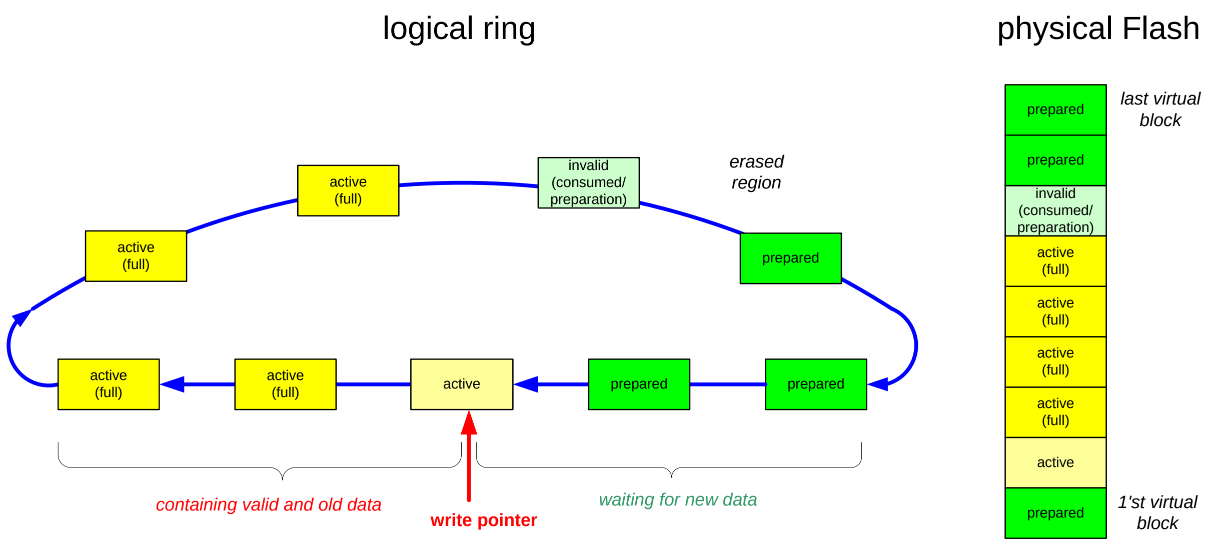


Figure 3: Virtual block ring buffer

Each block has a life-cycle status which indicates the current usage of the block. The following life-cycle statuses have to be distinguished:

- **Prepared:** The block is prepared to store data sets. This means that the block is essentially erased. Only a few cells are written to manage the block status.
- **Active:** An active block stores valid values of data sets.
- **Active (full):** Whenever it is not possible to append new data-set values to the contents of an active block, it is considered as full. However as long as it stores valid data-set values it still needs to be treated as active, i.e. “active (full)”.
- **Invalid:** Typically only one block in the ring buffer (Except special power fail conditions) is invalid. This block passes different conditions:
  - **Invalid(consumed):** Blocks contains only outdated data-set values and is marked invalid
  - **Invalid (preparation):** Consumed blocks are being erased and then marked prepared. The erasure of a virtual block is a comparably time-consuming operation.

Figure 3 considers a write pointer staying fix, while the ring buffer rotates clockwise. Every block reaching the write pointer gets activated. This block is called the *active zone head*. When a block reaches the end of the active zone it is called the *active zone tail*.

Each virtual block will pass a complete life cycle on every ring buffer loop as detailed in Figure 4.

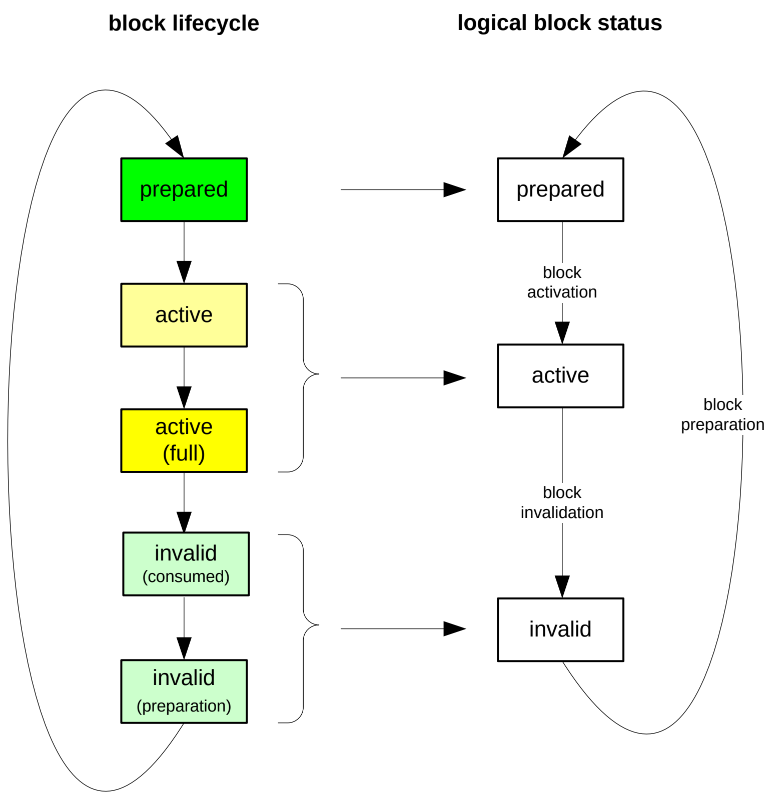


Figure 4: EEL block life cycle

The library itself does not need to distinguish all five phases of the block life cycle. For instance, the active state and the active (full) state are not explicitly distinguished by the library and mapped to a logical active state.

Also the invalid block (s) will be treated by the library in different processing steps.
If a certain data set is updated (i.e. written) seldom, it can happen that the active zone tail does not move meaning that it remains in the same physical block. In order to keep the rotation of the logical block ring alive, it is necessary to copy valid data sets from the active zone tail to the active zone head. After all data is copied, the active (full) block in the tail can be transferred to the invalid (consumed) state. This complete process is referred to as *refresh*.
Afterwards, the block is erased and then marked prepared. While the block status during this complete proceeding is considered as invalid (preparation), the processing is called *prepare.*

2.3.2 EEL Block Structure

The detailed block structure used by the EEL is depicted in Figure 5. In general, an EEL block is divided into three utilized areas: the block header, the reference zone and the data zone.

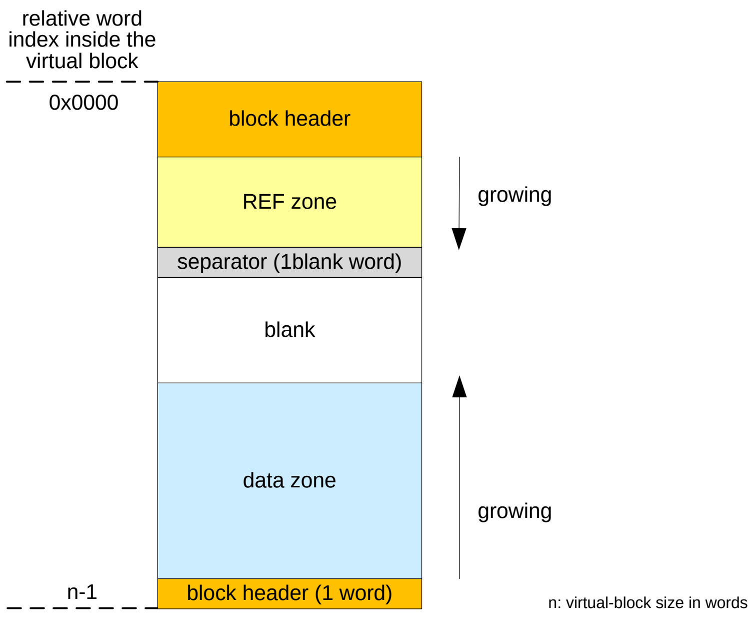


Figure 5: EEL block structure

The individual purpose of each area is given in the following:

- **Block header:** The block header contains all block status information needed for the block management within the EEL-pool. It has a fixed size of 7 words (28 bytes). Please note that one word of the block header is stored at the bottom of the virtual block (details in Section 0).
- **Reference zone:** The reference zone contains reference entries which are required for the management of EEL data sets. It grows in direction of larger indexes whenever a variable is written.
- **Data zone:** The data zone contains the pure data values of the defined EEL data sets. It grows in direction of smaller indexes whenever a variable is written.

Between reference and data area, there is an erased area of not-written flash cells. With each data set update (i.e. the DS is written), this area is reduced successively. However, at least one word of space always remains between reference and data zone for management and separation of these zones. This is indicated by the separator in Figure 5.

The EEL block header is detailed in the following, while the structure of variable instances stored in the reference and data area are described in Section 2.3.3.

2.3.2.1 EEL Block Header

The block header is a small area at the top and bottom of each flash block belonging to the EEL pool. It contains all information necessary for block management during EEL operation. It is composed of seven words, one of which is placed at the end of the block.

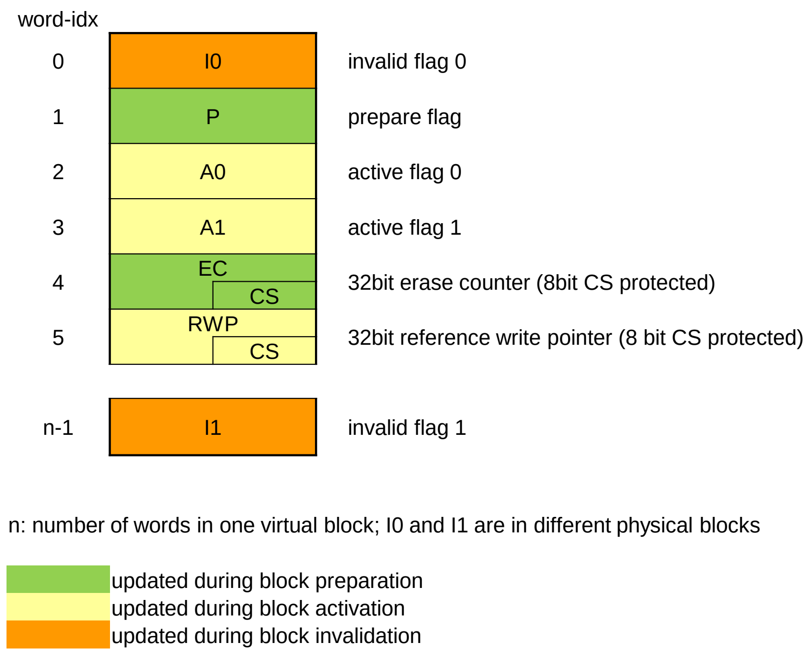

word-idx
0
I0
invalid flag 0
1
P
prepare flag
2
A0
active flag 0
3
A1
active flag 1
4
EC
CS
32bit erase counter (8bit CS protected)
5
RWP
CS
32bit reference write pointer (8 bit CS protected)
n-1
I1
invalid flag 1
n: number of words in one virtual block; I0 and I1 are in different physical blocks
updated during block preparation
updated during block activation
updated during block invalidation

Figure 6: Structure of each EEL block header

Inside the header area, a set of status words is used to code the block status in a reset-resistant manner. Each status word has a size of 4 byte. In the following, the basic purpose of each word is explained concentrating on the case of normal operation. Details about how reset resistance is ensured are omitted here for the sake of clarity.

P (prepare flag):

The prepare flag is written with the pattern 0x55555555 in order to indicate that a block is prepared. The prepare flag is updated during block preparation.

A0, A1 (active flags):

If the active flags are set to 0x55555555 (in addition to the prepare flag), the block is treated as active. These flags are written during block activation.

I0, I1 (invalid flags):

By setting the invalid flags to 0x55555555, a block is marked as invalid. Invalid flags have a higher priority than prepare and active flags and hence can be used to overwrite the block status.

The reason for splitting up invalid flags across the virtual block originates from the technology used for the Data Flash: It does not allow for overwriting already written cells with arbitrary values to mark patterns invalid. In order to never end in an instable block status, the invalidation flags are placed into two different physical Flash blocks within a virtual block.

EC (erase counter):

The erase counter is written during the preparation of a block according to the following rule:

EC = {previous block EC} + 1   , if the block is the first virtual block of the EEL pool
EC = {previous block EC}   , otherwise.

Thereby, on each ring buffer turn around the erase counter in each block is increased by 1 as exemplarly depicted in Figure 7. The EC is checksum protected in order to be robust against accidental overwriting due to application failures.

Please note that the erase counter does not necessarily match the real number of Flash block erase cycles, but only the erase cycles since the EEPROM emulation has been set up last time. The erase counter is affected by complete Data Flash erase or manual Flash modification (e.g. programmer or debugger).

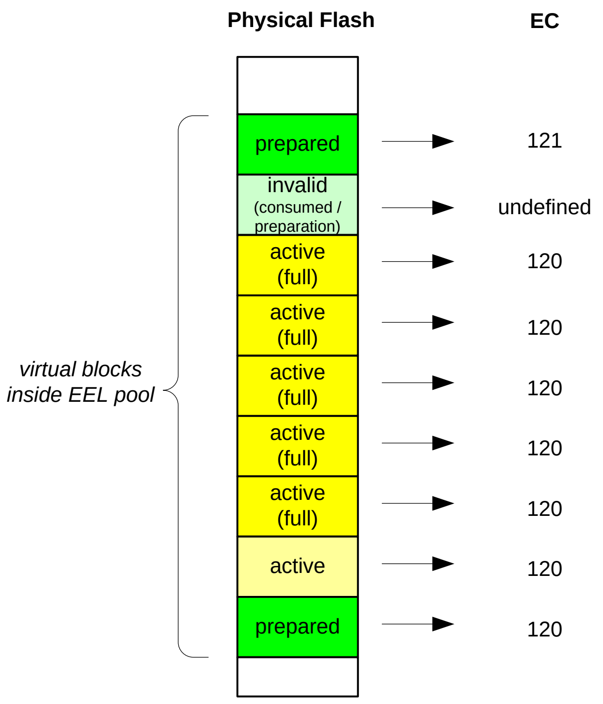


Figure 7: Exemplary erase counters across EEL pool

RWP (reference write pointer):

The reference write pointer is written in during the activation of a block. It points to the previous block separator between reference and data zone (see Figure 8). This enables a fast analysis of the reference zones of all active blocks during library start-up. The RWP is checksum protected in order to be robust against accidental overwriting due to application failures.

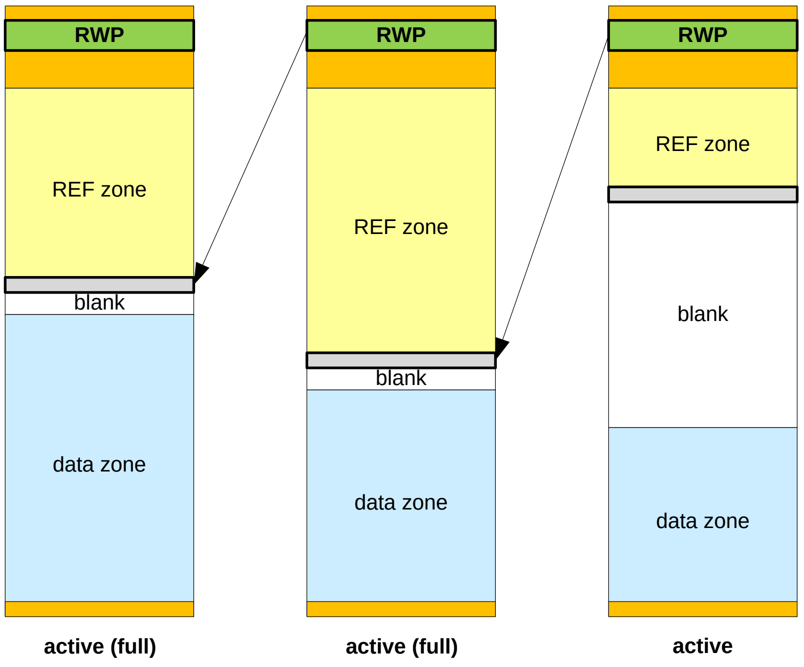


Figure 8: Usage of reference write pointers within the active zone of an EEL pool

2.3.3 EEL Data-Sets

Differing from a real EEPROM, where user data is referenced by addresses, the user data in the EEPROM emulation is referenced by identifiers (IDs). An ID is unique number referring to a data set with a dedicated length. By means of the so-called EEL descriptor, the developer can configure multiple data sets to be used within the EEL. (The exact specification of the format of the EEL descriptor can be found in Section 4.2.2, “EEL Runtime Configuration Parameters”.)

2.3.3.1 Data Set Instances within the EEL Block

Differing from an EEPROM, the data is stored "somewhere" in the Flash memory but not on a fix address. Whenever a DS content is updated, i.e. the DS is written, a new *instance* of the DS is written to the active block. This means that there can be multiple instances of a variable at a time. However, only the newest instance is considered and referred to when reading the DS.

Each DS instance consists of two mayor parts: a reference entry in the reference zone and a data entry in the data zone as shown in Figure 9. Thereby, the reference of a DS contains the ID and a pointer to the corresponding data.

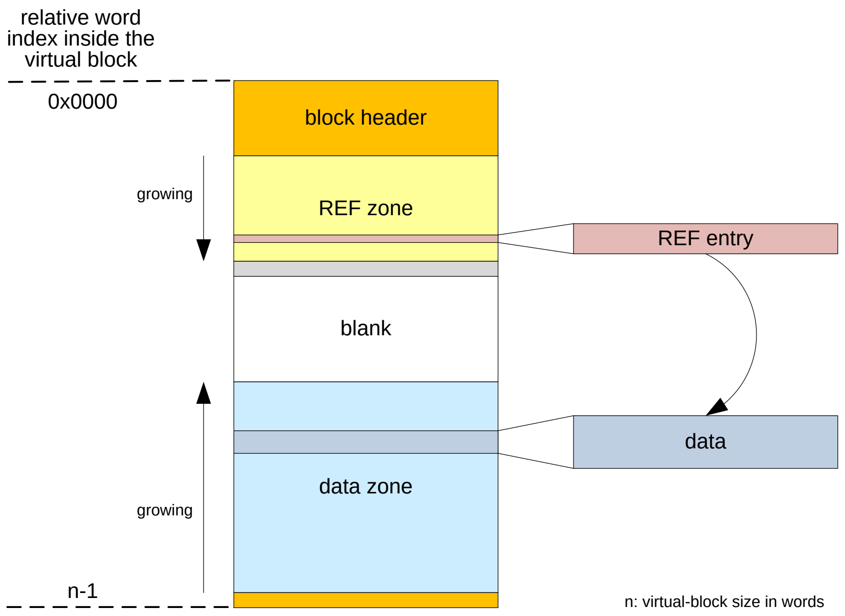

relative word index inside the virtual block
0x0000
block header
growing
REF zone
REF entry
blank
data
data zone
growing
n-1
n: virtual-block size in words

Figure 9: Data-set instance: REF entry points to user data

New reference entries are appended to the reference zone while the corresponding data is prepended to the data zone. As a result the reference zone is growing towards larger word indices while the data zone is growing towards smaller word indices.

Neither the reference entry nor the data of a DS instance is split across EEL blocks. This has two consequences:

- The size of a DS instance may not exceed the size of one virtual block (block minus header and separator). As a consequence, the virtual block size needs to be configured in considering the DS sizes (see Section 5.4.1, “EEL Pool Configuration”).
- A block has to be considered as full, whenever a DS to be written does not fit into the active block. In this case, the new DS instance has to be created within the next prepared block, making it new active block. Figure 10 and Figure 11 illustrate a DS write in case fitting into the current active block and not fit fitting in the active block respectively.

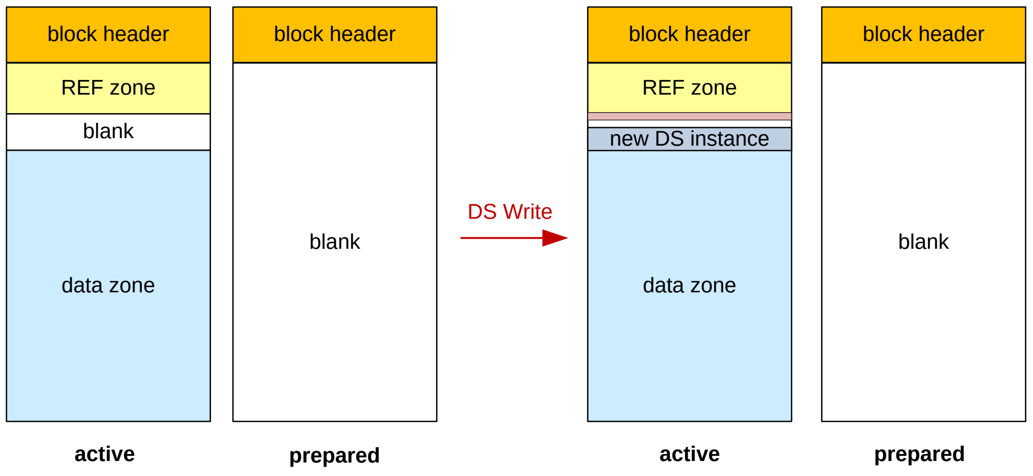


Figure 10: New DS fits into the active block

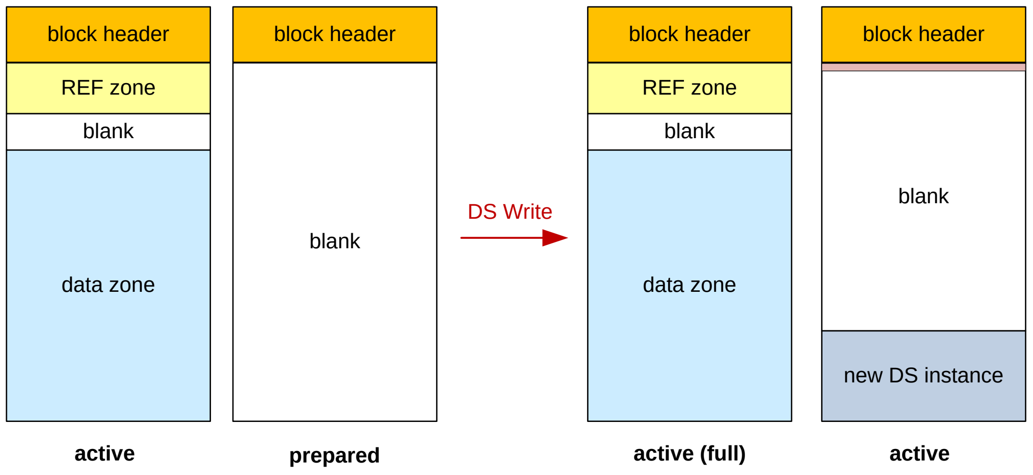


Figure 11: New DS does not fit into the active block

2.3.3.2 Data Set Structure

The data sets are stored according to the scheme shown in Figure 12. As already described in the previous sections, one data set instance is assembled from a reference entry and the corresponding user data.

Every reference entry is built up of four words (SOR, DRP, EOR1 and EOR2), each of which has a dedicated meaning as described in the following:

SOR (start of reference):

This flag is written first in a DS write sequence with the pattern 0x55555555 and therefore indicates that the write process has started.

DRP (data reference pointer):

The data reference pointer is written right after the SOR and contains two 16 bit values:

- 16-bit lower half word: ID,
- 16-bit upper half word: widx, a pointer to the data.

The DRP refers to the word index within the data flash and hence can address up to 256kB Data Flash. It points to the first element of the data array in the data zone. The actual data is written after the DRP has been initialized.

EOR0, EOR1 (end of reference):

The end of reference (first EOR0, then EOR1) is filled with the pattern 0x55555555 after the user data has been written to the data zone. It indicates the successful completion of a write sequence.

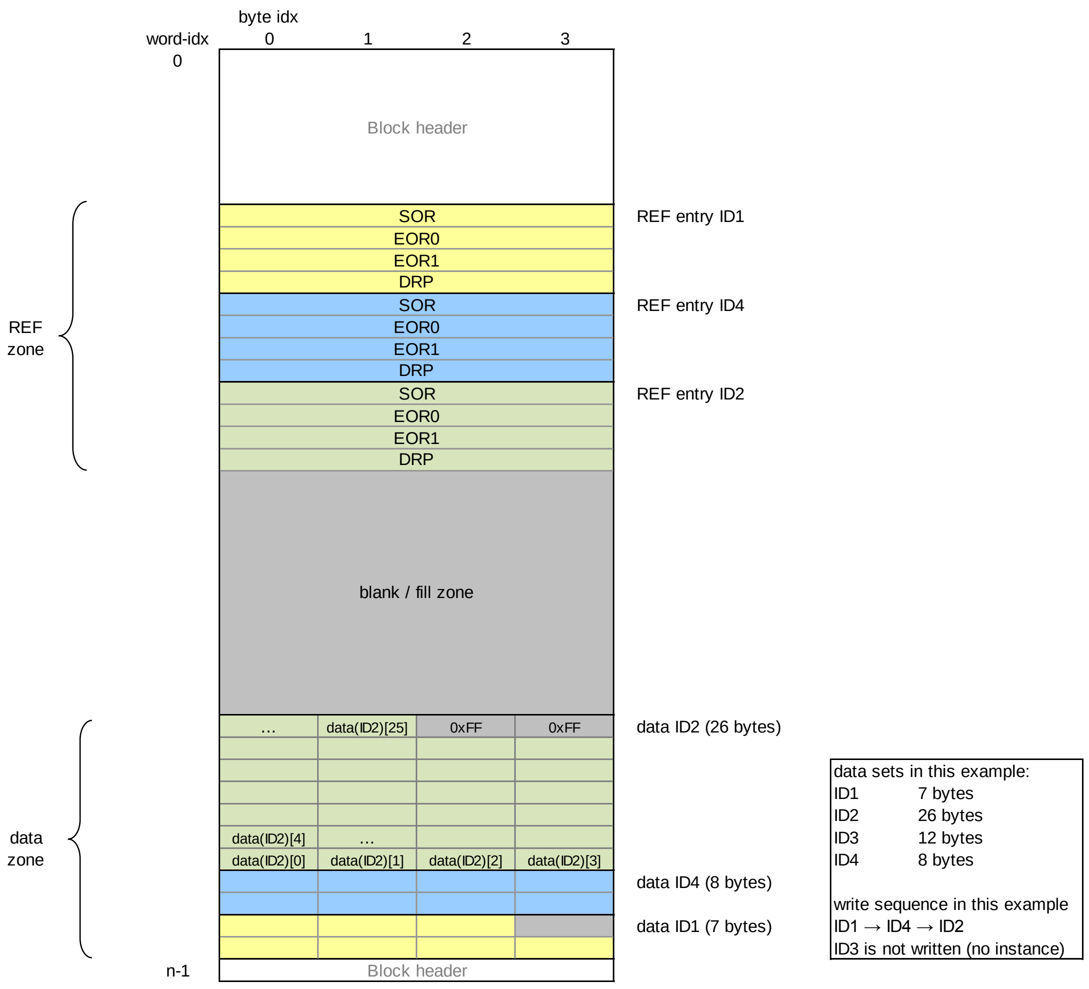


Figure 12: Reference entry and data structure details

The user data is stored sequentially from larger to smaller indices. As Flash accesses can only be performed on word granularity, the individual data sizes are expanded to a number of bytes dividable by 4. Unused bytes are set to 0xFF. The data of an DS instance is not protected by additional checksums or ECC. If the application requires additional data protection it is the user's duty to include the checksums in the raw data.

Please note that the data set instances are specifically designed to ensure that interrupted data set updates (writes) can be identified. This makes the library resistant against reset scenarios. The library detects incomplete write sequences at start-up and will in this case stick to the pervious value of the data set.

2.3.3.3 Invalid Data Sets

Besides the regular case that a DS has been written with arbitrary data, it may also be in an invalid, i.e. not contain any data. There are different ways this state can be reflected in the active blocks depending on the cause of invalidation.

In any case, the library will return an error (`EEL_ERR_NO_INSTANCE`) when trying to read an invalid data set.

Missing data set instance

In case that there is no instance of a particular data set available in the complete active zone of the EEL pool, the data set is invalid. This can happen for instance when a data set has not been written so far.

Invalidated data set by reference entry

The EEL provides a feature to actively invalidate already written data sets (see also Section 4.5.6, "R_EEL_CMD_INVALIDATE — DS invalidation"). This invalidation is realized by a special kind of reference entry. The widx of DRP is set to zero and no data is written in the data zone.

Please note that such an invalid instance will not be copied during maintenance processes running in the background of the EEL. This means that an invalidated data set will be eventually transferred to a missing data set instance as described above.

Chapter 3 Functional Specification

This chapter introduces the operation of the EEL. Thereby, the focus is put on the concepts and flows required for a proper usage of the library. The exact specification of the API can be found in Chapter 4.

3.1 Functions and Commands

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

3.1.1 Functions

The following functions are provided to control the EEL:

Initialization

- **R_EEL_Init:**
  This function is used to initialize the internal data structures of the EEL and to prepare it for the actual start-up.

Start-up/Shutdown

- **R_EEL_Startup:**
  R_EEL_Startup resets and starts the EEL state machine. It also triggers the EEL pool analysis, which is required in order to access the EEL data sets.
- **R_EEL_ShutDown:**
  This function initiates a controlled deactivation of the EEL state machine.

Suspend/Resume

- **R_EEL_SuspendRequest:**
  This function requests suspension of all active EEL operations and processes. Thereby, the library is put into a passive state.
- **R_EEL_ResumeRequest:**
  This function requests resuming the EEL operations after suspend.

Operation

- **R_EEL_Execute:**
  By means of the R_EEL_Execute function, the user can issue commands to access and manage the EEL data sets. It is one of the main functions for utilizing the EEL. However, please note that issued commands are not completed directly but rather require to be processed with calls of R_EEL_Handler.
- **R_EEL_Handler:**
  R_EEL_Handler needs to be called regularly to drive pending commands and to observe their progress. Please note that R_EEL_Handler is also used to drive some operations triggered by a function (e.g. for start-up, shutdown, suspend and resume).

Administrative

- **R_EEL_GetDriverStatus:**
  This function opens a way to check the internal status of the EEL driver. On the one hand this enables to monitor the progress of pending operations. On the other hand R_EEL_GetDriverStatus can be used to analyze causes of command errors.

- **R_EEL_GetSpace:**
  This function returns the current free space in the EEL pool (prepared space for new data).
- **R_EEL_GetVersionString:**
  This function returns the pointer to the library version string.
- **R_EEL_GetEraseCounter:**
  This function reads the current erase counter of the ring buffer/EEL pool.

3.1.2 Commands

Commands are used to manage the EEL pool and to access the EEL dataset. The number and size if the EEL data sets can be defined by the designer at compile time.

Commands are initiated via R_EEL_Execute and processed stepwise by consecutive calls of R_EEL_Handler. This way, the execution of each EEL command is separated into several steps processed by an EEL-internal state machine. The following commands are offered by the EEL:

Pool-oriented Commands

- **Format:**
  Format the EEL pool for initial usage with the EEL. All Flash blocks of the EEL pool are erased and prepared for later writing of data sets.
- **Clean-up:**
  This command can be used to defragment the Flash ring buffer (EEL pool). All DSs are refreshed and obsolete DS instances are deleted. Following that, the pool will afterwards contain only one DS instance for each ID and the pool will contain as much prepared Flash space as possible to receive new data.

Data set-oriented Commands

- **DS read:**
  Read a DS identified by a given ID and copy the read data to a read buffer provided by the user application. The Read operation has the highest priority of all standard operations and can interrupt all write/invalidate operations.
- **DS write:**
  Write a DS identified by a given ID. The data is provided by a user application buffer.
- **Incremental DS write:**
  The operation is based on the normal write but checks in advance whether the given data has changed since last DS writing. Only then, a normal write operation is executed.
- **Immediate DS write:**
  The immediate write has the same functionality as write. However, the command is executed with a higher priority. Thereby, the Immediate write can interrupt any normal priority write/invalidate operations.
- **Incremental immediate DS write:**
  This command has the same functionality as an incremental DS write, but it is executed with a higher priority. It can interrupt any normal priority write/invalidate operation.
- **DS invalidation:**
  This command sets a DS identified by a given ID to invalid. A read operation on an invalidated DS will return a read error instead of reading data.
- **Immediate DS invalidation:**
  This command has the same functionality as DS invalidation, but the execution priority is higher.

3.1.3 Request-Response oriented Dialog

The EEL utilizes a request-response architecture to initiate the commands. This means a request variable has to be prepared by the application as a kind of "request form sheet" (see Figure 13) and pass it by reference to the EEL driver using its R_EEL_Execute function. The EEL interprets the content of the request variable, checks its plausibility and initiates the execution. The feedback is reflected immediately to the requester via the status member of the same request variable. The completion of an accepted request/command is done by calling R_EEL_Handler periodically as long the request remains "busy".

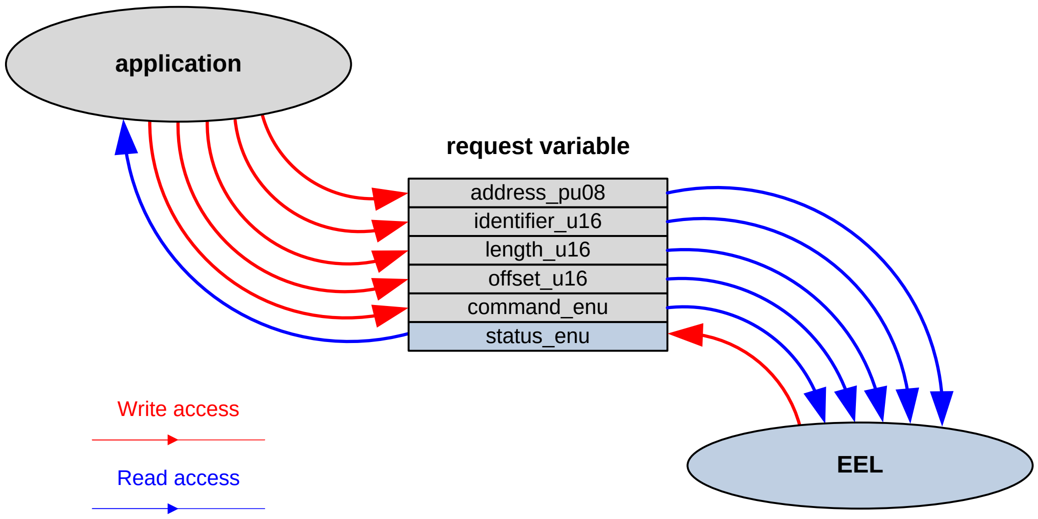


Figure 13: Schematic usage of the request variable

The biggest advantage of the request-response architecture is the constant and narrow parameter interface. It allows steady parameter passing and is therefore independent from the used compiler and its memory models. Furthermore it allows for easy EEL integration into operation systems as well as building EEL operations stacks and buffers.

The details on the request variable structure and its members are given later in Section 4.3.2.6, “r_eel_request_t”. Please also note that not all structure members are required for all commands. The individual command descriptions in Section 4.5, “Commands” provide the corresponding detailed information.

Note:
The request variable can be accessed (read or written) by the library at any time when R_EEL_Handler is executed. This also means that it is imperative to ensure the existence of the data structure as long as the command is being processed.

3.1.4 Handler-oriented Command Execution

In order to satisfy the operation in concurrent or distributed systems, the command execution is divided into two steps:

1. Initiation of the command execution using R_EEL_Execute.
2. Processing of the requested command state by state using R_EEL_Handler.

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

Please note that R_EEL_Execute only initiates the command execution and returns immediately with the request-status "busy" after execution of the first internal state (or an error in case the request cannot be accepted). The further command execution is performed in R_EEL_Handler, where the internal sequences of the command are executed state by state. Together with the background-operation feature of the FDL, this enables to design complex applications utilizing the computational resources of the processor efficiently, i.e. to perform other tasks on the CPU while flash operations are running in parallel.

Note:
Each state has a strictly limited execution time. Based on that, the library function controlling the state machine—the R_EEL_Handler—will immediately return to the user application.

3.2 Flash Interrupt Support

The EEL is prepared to support the Flash interrupt. This means, that the EEL triggers the Flash interrupt when the handler function shall be called in order to process a next EEL process state. By that, the handler function can be executed in the Flash interrupt context or in an interrupt triggered task which means as few as possible handler function calls (no polling) by achieving the best EEL performance.

Basically, each Flash operation end triggers the Flash interrupt. However, quite some EEL state machine internal states (User operations as well as background processes) do not issue a Flash operation. In order to support the Flash interrupt in a sufficient way, these states must issue the Flash interrupt by SW. This can be achieved by the EEL configuration (See Section 4.1,”Pre-compile Configuration”).

Note:
Even when the EEL is idle, the handler function shall be called regularly in order to execute idle time supervision tasks like bit error check (See Section 3.4, "Background Operations"). As for that the handler shall not be called with high frequency (e.g. 10ms~100ms task), the user application need to trigger the interrupt by software.

3.3 EEL User Command Priority

The EEL provides the following user operations which are invoked by appropriate commands: *Format*, *Clean-up*, *DS read*, *DS write*, *Incremental DS write*, *Immediate DS write*, *Incremental immediate DS write*, *DS invalidation* and *Immediate DS invalidation*. These commands have partially been mentioned before and are described in the API description.

The Read and Write operations are considered to be prioritized according to the following scheme:

- **Priority 1:**
  DS Read
  can interrupt
  DS Write, *Incremental DS Write*, *Invalidation*, *Immediate DS Write*, *Immediate Incremental DS Write* and *Immediate DS Invalidation*.
- **Priority 2:**
  Immediate DS Write, *Immediate Incremental DS Write* and *Immediate DS Invalidation*
  can interrupt
  DS Write, *Incremental DS Write* and *Invalidation*.
- **Priority 3:**
  DS Write, *Incremental DS Write* and *DS Invalidation* cannot interrupt any other user operation.

The following rules apply to these operations:

- All of the above operations can interrupt ongoing background operations (see Section 3.4, “Background Operations”).
- A command invoking an operation when an operation of the same priority is ongoing will be rejected.
- When an operation of a higher priority is invoked, a possibly ongoing operation of a lower priority will be suspended.
- When invoking an operation of a lower priority, a possibly ongoing operation of a higher priority is will be finished first then the lower priority operation is executed.

Furthermore, special conditions apply for the other commands:

- **Format command:**
  The format command requires that the system executes no user or background operations. If this is not the case, the command will be rejected. When started, all other operations are blocked.
- **Clean-up command:**
  The Clean-up command requires that the system executes no user or background operations. If this is not the case, the command will be rejected. After being started other operations can be executed, the Clean-up operation will be suspended and later on resumed automatically.

3.4 Background Operations

The EEL operations are based on independent processes for the different user commands, such as Read, Write, Immediate Write. Furthermore, background processes are to be executed in order to manage the EEL pool and to manage the start-up flow.

The following background operations and their processes are available:

- **Refresh:**
  This process manages copying any potential DS instances from the EEL pool active tail to the head before invalidating a virtual block. The copy process itself is done by the *Write (Refresh)* process. After invalidation, the block can be prepared again.
- **Write (Refresh):**
  This process is triggered by the refresh process to copy one DS instance from the active zone tail to the active zone head.
- **Prepare:**
  The prepare process erases invalid Flash blocks and marks them prepared. Thereby new space for the active pool is provided.
- **Supervision:**
  This process manages the complete start-up processing (See Section 3.7, “Start-up Processing”). When the library is started up and in normal operation, it controls the following background processes:
  - The supervision process triggers the refresh process and sequentially the prepare process to provide enough prepared pool space to store new DS instances. In order to decide when it is necessary to start a refresh, the so-called *refresh threshold* can be set at compile time (see Section 4.2.2, “EEL Runtime Configuration Parameters”). The refresh threshold specifies the number of blocks which the library background operation should always try to keep prepared.
  - When no further pool handling is required, the process checks the EEL pool for bit errors (c.f. Section 1.1.4). To do so, the complete pool address range (only active and prepared virtual blocks) is checked word by word using the FDL bit error check function. On detection of a bit error, further refresh and prepare operations are triggered and by that, sequentially all blocks are refreshed. This is continued until the bit error is gone.
    In this case any ECC error and exception is handled completely by the EEL. There is no need for the user to intervene or react.

On invocation of a user command, the background processes are interrupted at the next possible state in order to execute the user processes. After user process termination, the background processes are continued.

3.5 Data-Set Search and Read

The library uses internal tables to store the DS size information and latest DS location.

While the DS size is stored together with the ID statically in ROM (the so-called ID-L table), the pointers to the latest DS instances are evaluated on library start-up and stored in RAM (the so-called IDX table) as depicted in Figure 14.

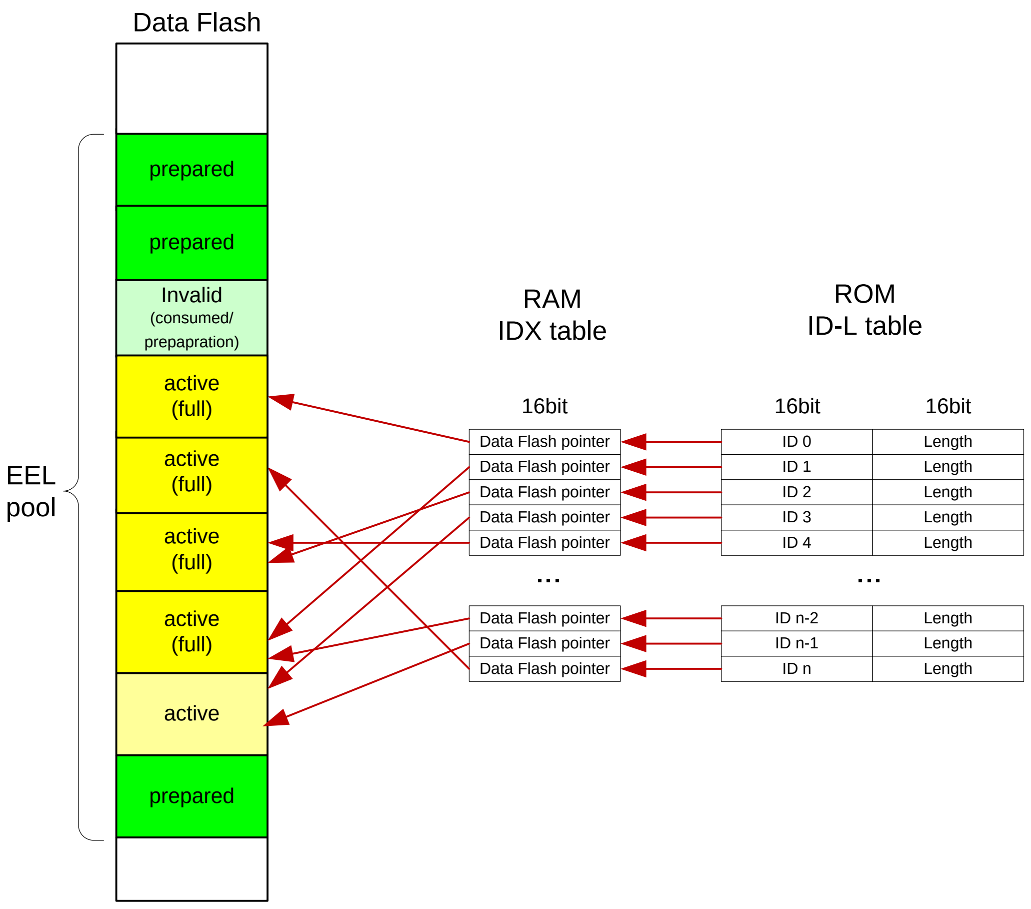


Figure 14: Library ID tables

The ID-L table (ROM table) contains one entry for each ID available in the system, together with its DS length information. This table is configured at compile time.

IDX table (RAM table) contains for each ID available in the system the pointer to the latest data instance. On EEL start-up the IDX table is filled and continuously updated on each DS Write access.

Searching DS instances via these ID tables is fast. However, the RAM table needs to be built up during start-up before this mechanism can be used. Therefore, the EEL supports two different mechanisms for data read:

- **ROM table search**
  Whenever a DS with a dedicated ID shall be read, the requested ID is searched in the ROM table. The index of the ROM table entry with the fitting ID is then used to get the data pointer (to the Data Flash) from the RAM table.
  This ROM table search is fast, but the RAM table must be initialized on start-up which requires some time.
  The ROM table is used for the read process as well as for the refresh process

- **REF zone search**
  In order to be able to read data without initialized RAM table, the library provides another read (data search) mechanism. The library can parse the REF zone of the blocks and read the entries sequentially until an entry with the requested ID is found. It needs to be considered, that the REF zone parsing requires some time and creates 100% CPU load.
  The REF zone search is used in the library start-up phase, when the ID-L table is not yet initializes and also in other special library operation modes (see Section 3.8, “Limited Operation Mode”).

3.6 Driver Status

The library internal status can be monitored by the user application by means of the function `R_EEL_GetDriverStatus`. The library status is separated into three orthogonal classes: the operational status, the access status and the background operation status. For better usability, the three status components are collected in a data structure (see Section 4.3.2.9, “r_eel_driver_status_t”). Each status class is individually detailed in the following.

Operational Status

The operational status describes the general status of the EEL state machine. Table 1 shows the different operational statuses the EEL can assume during execution.

Table 1: Operational statuses of the EEL

Status	Description
R_EEL_OPERATION_PASSIVE	The state machine can handle neither internal nor user initiated processes. This state is set • after EEL initialization and before EEL start-up • after EEL shutdown is finished • after fatal EEL operations errors
R_EEL_OPERATION_STARTUP	This status is set as long as the start-up processing is ongoing. This indicates that the EEL is not completely up and running. As long as this operational status is returned, EEL functionality is inhibited or limited. Please see emulation access status below.
R_EEL_ OPERATION _BUSY	This status is set, if either a background process, e.g. refresh or prepare is active or a user process read or write is being processed. As Flash operations may be processed, the device should not be switched off in this status in order to avoid repair operations to be executed on EEL star-up.
R_EEL_ OPERATION _IDLE	No process active except supervision doing margin checks. No refresh or prepare necessary and no user process read, write, format active.
R_EEL_OPERATION_SUSPENDED	When the suspend request is issued to the EEL by the `R_EEL_SuspendRequest` function, the state machine will try to enter the suspend mode. As this cannot be done immediately, the application need to call the handler function frequently until the suspend status is set.

The status `R_EEL_OPERATION_IDLE` also signals to the user application that no EEL operation and with that also no Flash modification is ongoing. This is an indication that a power save mode can be entered.

Note:
The user application needs to ensure that no power save mode is entered that may result in losing any Flash programming hardware contents (e.g. deep stop).

Access Status

During start-up, the full functionality of the EEPROM emulation is not given. It is increased step by step depending on the proceeding of the start-up flow. The available functionality is defined by the access status as presented in Table 2.

It is important, that not only start-up processing affects the access level, but also EEL failures may result in loss of functionality. Depending on the failure, either Write is prohibited or no access is possible.

Table 2: Access statuses of the EEL

Status	Description
R_EEL_ACCESS_LOCKED	During Start-up: The state machine is in an early start-up phase and so, does not accept any user operation. During normal operation: Due to a failure no more data access is possible.
R_EEL_ACCESS_READ_WRITE	Set during Start-up only. The state machine proceeded further in the start-up phase and so, accepts DS read and write operations. • The read operations require REF table search as the RAM table is not yet available. So, the read requires longer execution time at 100% CPU load • The DS write capability is limited to the available passive blocks (prepared and invalid) as due to the missing RAM table no refresh operation is possible
R_EEL_ACCESS_READ_ONLY	During normal operation only: A user DS write operation resulted in a Flash write error, either caused by hardware or software problem. In order to preserve the remaining Flash contents the library forbids any further Flash modification operations. Read operations are still possible, however a certain risk is given, that the read data may be wrong if the write operation caused damage to the read data.
R_EEL_ACCESS_UNLOCK	The state machine is up and running. All user and background operations should be possible, if no error occurred. The RAM table is built up, so read operations are executed fast from now on.

Background Operation Status

Beside user operations, also the background operations may return errors. As only process errors are considered (no errors on `R_EEL_Execute` resulting in not starting an operation), the error range is limited to the ones presented in below.

Table 3: Background operation statuses of the EEL

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	No background operation problem.
		reason	-
		remedy	-
R_EEL_ERR_FIX_DONE	warning	meaning	During start-up processing a fix in the EEL pool blocks had to be done.
		reason	Block handling operations (activation, invalidation, preparation) were interrupted, e.g. by power fail. When the EEL start-up processing detects this, the block will be invalidated and the warning is returned.
		remedy	No remedy necessary. The status is a warning only. **Note:** After reading the warning once by the function `R_EEL_GetDriverStatus`, the warning is reset.

R_EEL_ERR_FLASH_ERROR	error	meaning	Some Flash contents could not be erased or written.
		reason	A background Flash erase or write operation ended with an error. The Flash range has no defined status, allowing using it later on.
		remedy	Stop the emulation and investigate in the root cause.
R_EEL_ERR_POOL_ INCONSISTENT	error	meaning	The EEL pool structure is not consistent and the EEL cannot work with it.
		reason	Start-up processing does several consistency checks on the EEL pool. If one of the checks fails, the EEL sets this error.
		remedy	Stop the emulation and investigate in the root cause.
R_EEL_ERR_INTERNAL	error	meaning	A library internal problem occurred, that cannot be related to a concrete root cause. The library will be locked.
		reason	Some library internal checks that should never fail, failed. These can be checks on hardware or software values.
		remedy	Stop emulation and investigate the root cause. In some cases, e.g. in case of external influence on the EEL or FDL variables or the Flash hardware (e.g. caused by wild running application pointers or PC), re-initialization of the FDL and EEL may help. A reasonable proceeding might be: During development: Stop the emulation and investigate in the root cause In the field: 2~3 times try to re-initialize the library (or reset the device). If the problem still exists, stop the emulation and investigate in the root cause.

3.7 Start-up Processing

The start-up processing is controlled by the internal state machine of the EEL. After library initialization and start-up invocation by means of R_EEL_Startup, several start-up process steps are executed until the system is in normal operation. Along with the start-up progress the access rights to the data and the library features are unlocked and the full performance of the EEL is reached. The actual steps during start-up are illustrated in Figure 15.

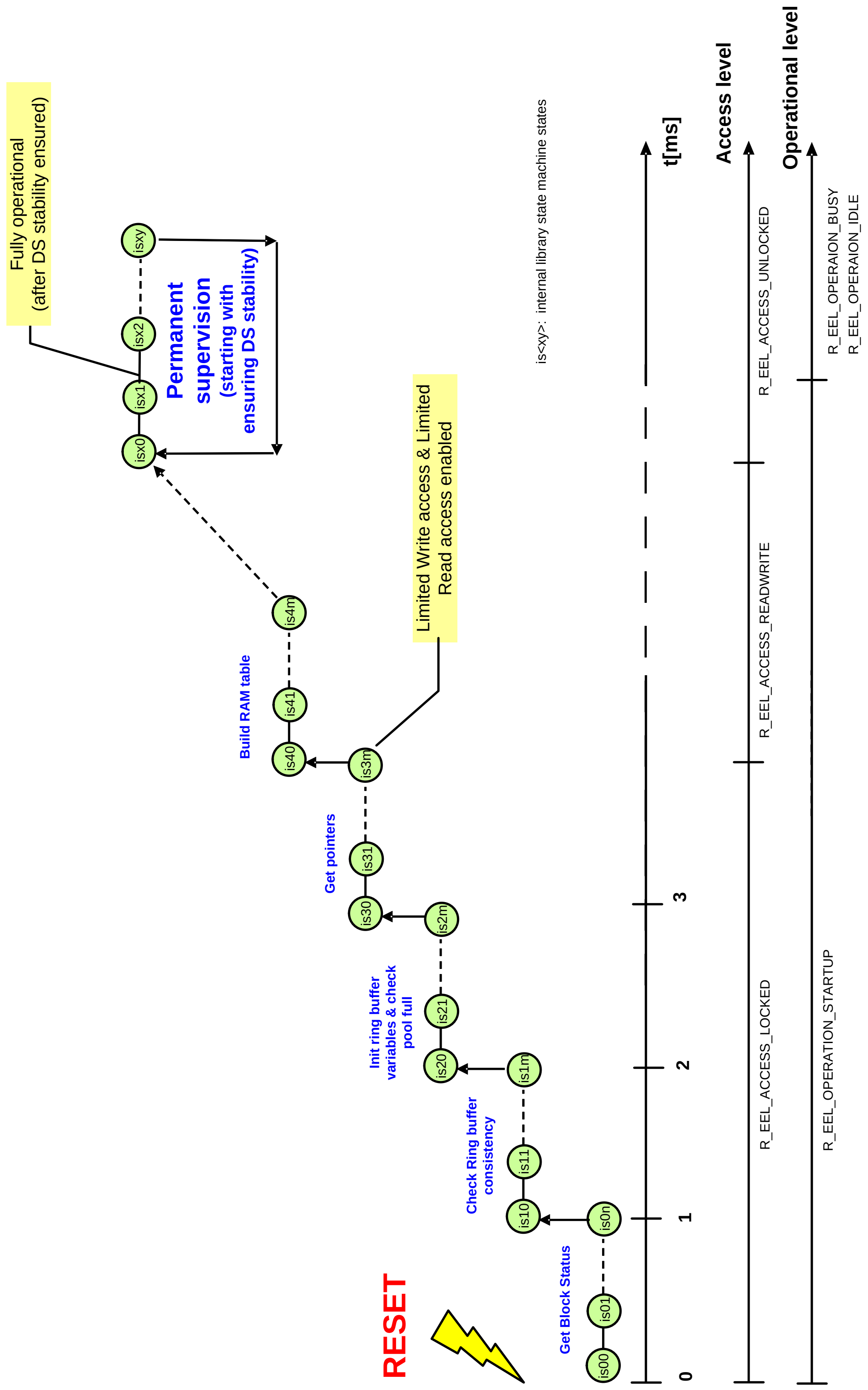


Figure 15: Start-up processing steps

The start-up progress can be checked by the user application with the function `R_EEL_GetDriverStatus` which returns the access status and the operational status (see Section 3.6, “Driver Status”). Please check Table 4 for the status values depending on the progress.

Table 4: Start-up processing steps

Start-up progress	Access status	Operational status	Comment
EEL initialized	R_EEL_ACCESS_ LOCKED	R_EEL_OPERATION_ PASSIVE	All library operations are prevented.
EEL start-up started	R_EEL_ACCESS_ LOCKED	R_EEL_OPERATION_ STARTUP	All library operations are prevented.
EEL start-up ongoing - basic start-up finished	R_EEL_ACCESS_ READ_ WRITE	R_EEL_OPERATION_ STARTUP	DS read is possible with limited performance (REF zone search). DS Write is possible until the prepared blocks are full.
EEL start-up ongoing - RAM table filled	R_EEL_ACCESS_ UNLOCKED	R_EEL_OPERATION_ STARTUP	DS Read is possible with full performance (ROM table search). DS Write is possible and supervision processing is active to manage the ring buffer.
EEL start-up end	R_EEL_ACCESS_ UNLOCKED	R_EEL_OPERATION_ BUSY or _IDLE (depending on refresh/ prepare operations are to be done)	DS Read and DS Write as before. Stability of the latest DS instances is ensured.

In case of a fatal error during any start-up step, the library switches to R_EEL_ACCESS_LOCKED and R_EEL_OPERATION_PASSIVE and the function R_EEL_GetDriverStatus will additionally return an appropriate error.

Note:
The last start-up processing step (ensuring the stability of the latest DS instances) checks if the valid DS instances have been completely written. Therefore, it checks if the last step of a DS write was executed (EOR1 is written). If not, redundant information (valid EOR0) ensures that the DS data is valid. On detection of such cases, the DS is refreshed (copied to active zone head).

3.8 Limited Operation Mode

The following description assumes a usage scenario with a device containing a boot loader and an application.

The boot loader as well as the application needs to access EEPROM emulation data with read as well as write. While the application requires frequent data write, the boot loader will only store a very limited amount of data, e.g. to store the application update process status.

The ROM ID-L table containing all IDs available in the emulation belongs to the application. On application update it needs to be removed together with the application.

So, the boot loader cannot trust on the availability of the application table, but requires a separate one. However, as the ID-L table may change with the application update, the boot loader can only have a limited table with always available IDs (with stable DS length), owned by the boot loader.

In order to support the scenario, the library additionally provides the so-called “limited mode”.

The major issue in this scenario is that the EEL requires an ID-L table containing all available IDs for the full library functionality. This affects the refresh process which will have a limited performance and require more CPU load without complete ID-L table. Furthermore, the library will ensure the DS stability (data retention) during the start-up for the case that a power fail interrupted writing the EOR markers. It will do this by refreshing the data when EOR1 is not available. This mechanism will not be executed in limited mode. Although, the possibility of data loss caused by power fail due to the missing feature is very low, the EEL should be re-started in normal mode soon.

The mode configuration is done by the initialization function `R_EEL_Init`. In order to change the mode, EEL_Init need to be called again.

- **R_EEL_OPERATION_MODE_NORMAL:**
  Full (normal) operation of the library, requires the complete ID-L table in ROM.
- **R_EEL_OPERATION_MODE_LIMITED:**
  Operation with limited ID-L-table in ROM, containing only the IDs required by the boot loader. The DS read and write work on the ID-L table.

Figure 16 presents a flow chart illustrating the application update idea. Although the boot loader may also always work in limited mode, the sample explains how the EEL can be switched between normal operation mode (application ID-L table available) and limited mode (only boot loader ID-L table).

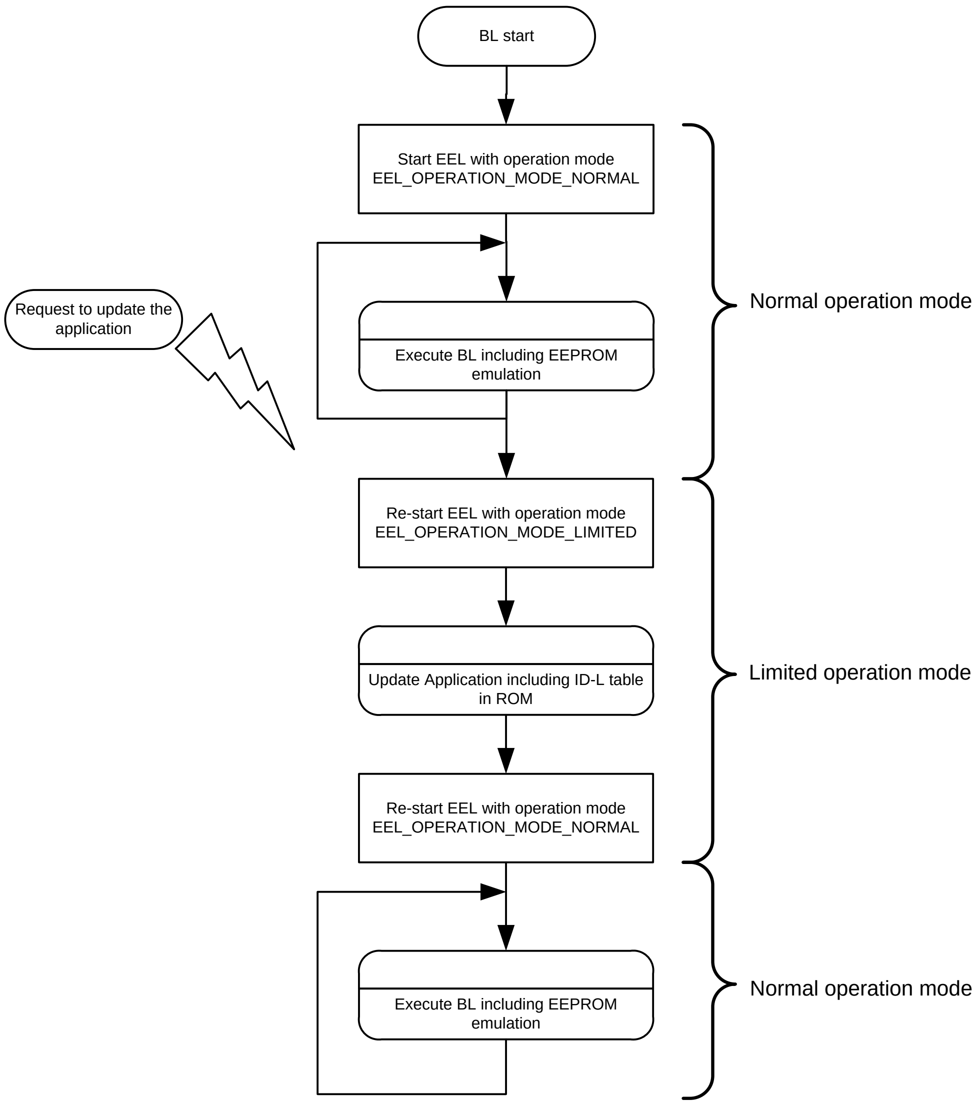

BL start
Start EEL with operation mode EEL_OPERATION_MODE_NORMAL
Request to update the application
Execute BL including EEPROM emulation
Normal operation mode
Re-start EEL with operation mode EEL_OPERATION_MODE_LIMITED
Update Application including ID-L table in ROM
Limited operation mode
Re-start EEL with operation mode EEL_OPERATION_MODE_NORMAL
Execute BL including EEPROM emulation
Normal operation mode

Figure 16: Switching between normal and limited operation mode

3.9 Suspend / Resume

The library provides the functionality to suspend and resume its 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.

Please note that a proper synchronization between EEL, FDL and Flash access via memory-mapped I/O is the user’s duty (c.f. Figure 1 on page 14). This means that the user has to ensure that only one way of flash access is used at the same time. In general there are two ways to treat this synchronization from an EEL point of view as described in the following.

EEL Suspend / Resume

This EEL suspend/resume mechanism allows to suspend the EEL operation on top level. However, this functionality will not suspend underlying low level operations but just the ensure that no further processes are started (such as prepare, refresh and user commands). Ongoing Flash (FDL) operations will be finished before the suspend status is entered. This implies a longer suspend latency time. Please see sections 4.4.4, “R_EEL_SuspendRequest” and 4.4.5, “R_EEL_ResumeRequest” for a detailed description of the corresponding API functions `R_EEL_SuspendRequest` and `R_EEL_ResumeRequest`. Please note that both of these functions only trigger suspension/resume of the EEL. Consecutive calls of `R_EEL_Handler` are required in a sufficient amount in order to drive the EEL state machine into and back from the suspended state as exemplified in the following sequence:

Call `R_EEL_SuspendRequest` ⇨ cyclic call of `R_EEL_Handler` until the library is suspended ⇨ perform FDL commands and ensure that they are completed by a sufficient amount of `R_FDL_Handler` calls ⇨ call `R_EEL_ResumeRequest` ⇨ cyclic call of `R_EEL_Handler` until the library is resumed ⇨ continue EEL operation.

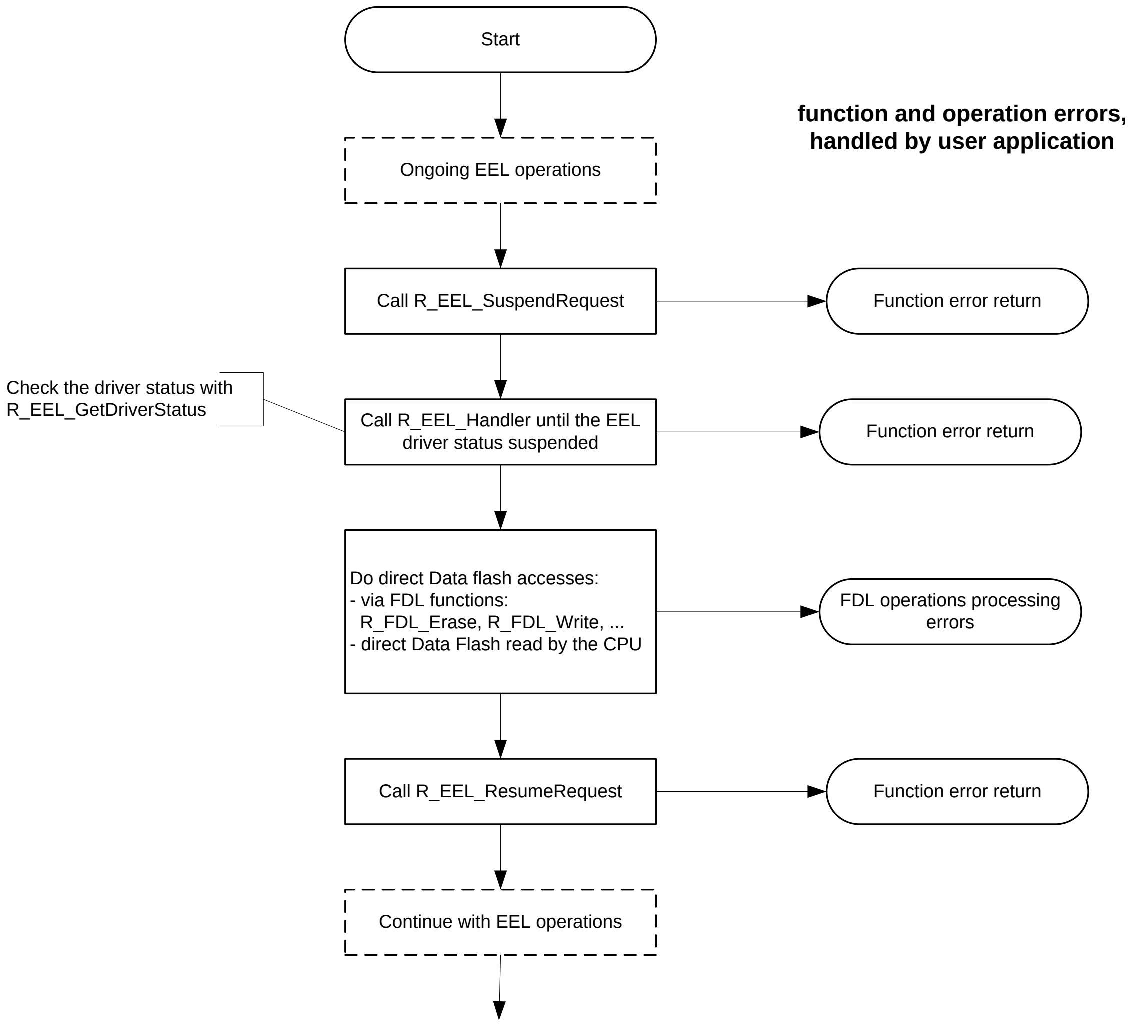

Start
function and operation errors, handled by user application
Ongoing EEL operations
Call R_EEL_SuspendRequest
Function error return
Check the driver status with R_EEL_GetDriverStatus
Call R_EEL_Handler until the EEL driver status suspended
Function error return
Do direct Data flash accesses:
- via FDL functions:
R_FDL_Erase, R_FDL_Write, ...
- direct Data Flash read by the CPU
FDL operations processing errors
Call R_EEL_ResumeRequest
Function error return
Continue with EEL operations

Figure 17: Suspend-resume flow

While this suspend/resume mechanism operates on a high level of abstraction, it comes with the drawback that a quick switch between the modes is not guaranteed as pending Flash operations are finished first.

FDL Standby / Wakeup

If faster mechanisms are required, the low layer FDL provides a standby/wakeup mechanism to suspend ongoing Flash erase and write accesses (see FDL documentation). The user application will have to take care for the synchronization between FDL and EEL by a proper sequence, e.g.:

Cyclic `R_EEL_Handler` calls ⇨ cyclic call of `R_FDL_StandBy` until the FDL is in standby ⇨ read Data Flash contents via memory-mapped I/O ⇨ call of `R_FDL_WakeUp` ⇨ continue EEL operation.

This mechanism allows reading Data Flash with very low latency. Writing Data Flash by the user application requires EEL suspending—and with that a higher latency.

Chapter 4 Application Programming Interface (API)

This chapter provides the formal description of the application programming interface of the EEPROM Emulation Library Type T01 for RH850 devices. It is strongly advised to read and understand the previous chapters presenting the concepts and structures of the library before continuing with the API details.

4.1 Pre-compile Configuration

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

The configuration contains the following element:

- **R_EEL_FLINT_SET_SW:**
  Each Flash operation end triggers the Flash interrupt. However, quite some EEL state machine internal states do not issue a Flash operation. If the handler function shall be executed in the Flash interrupt context, these states must issue the Flash interrupt by SW.
  The `R_EEL_FLINT_SET_SW` macro can be used to request the Flash interrupt by SW. If set, it is called within the handler function at the end of each process state, when no Flash operation was started.
  Please note that in case it is used this define is device family specific.

Sample implementation in eel_cfg.h for RH850 F1L devices:

```
#define R_EEL_FLINT_SET_SW ((*(uint16_t*)0xffff60feuL) = \
                                                (*(uint16_t*)0xffff60feuL) | 0x1000)
```

4.2 Runtime Configuration

The overall EEPROM emulation runtime configuration is defined by an EEL-specific part (EEL runtime configuration) and by the FDL runtime 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 runtime configuration.

Both configurations of FDL and EEL are stored in descriptor structures which are declared in r_fdl_types.h / r_eel_types.h, but defined in the user application and passed to the libraries as reference by means of the functions `R_FDL_Init` and `R_EEL_Init`.

The files fdl_descriptor.c and eel_descriptor.c show an example of the descriptor structure definition and filling, while fdl_descriptor.h and eel_descriptor.h show an example of the definitions required to fill in the structure. For more details on the file structure, please refer to Section 5.1.

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

4.2.1 FDL Runtime Configuration Parameters

The following settings shall be configured by the user inside the FDL descriptor. In the sample application, they are set as defines in r_fdl_descriptor.h, but could (even if not recommended) also be configured at run-time by other means:

- **CPU_FREQUENCY_MHZ:**
  This defines the internal CPU frequency in MHz unit, rounded up to the nearest integer, e.g. for 24.3 MHz set `CPU_FREQUENCY_MHZ` to 25. Please check the Device Manual for limit values.
  Note:
  The define requires the CPU frequency, not the crystal frequency. The CPU frequency must be set correctly. If not, malfunction may occur such as unstable Flash data without data retention, programming failure or operation blocking.

- **FDL_POOL_SIZE:**
  It defines the number of blocks to be accessed by the FDL for user access and EEL access. Usually it is set to the total number of blocks physically available on the device. For example, if the device is equipped with 32 KB of Data Flash and the block size is 64 bytes, then FDL_POOL_SIZE can be any value up to 512.

Value range:	Min:	`EEL_POOL_SIZE`
	Max:	Physical number of Data Flash blocks

- **EEL_POOL_START:**
  It defines the starting block of the EEL-Pool. If FDL is used without EEL on top, the value should be set to 0.

Value range:	Min:	0
	Max:	`FDL_POOL_SIZE` - `EEL_POOL_SIZE`

- **EEL_POOL_SIZE:**
  It defines the number of blocks used for the EEL-Pool. If FDL is used without EEL on top, the value should be set to 0.

Value range:	Min:	4 * `EEL_VIRTUALBLOCKSIZE`	(see below for virtual block size)
	Max:	`FDL_POOL_SIZE` - `EEL_POOL_START`	

For setting these parameters it is sufficient to adapt the defines specified in fdl_descriptor.h. The defined values are utilized in fdl_descriptor.c in order to initialize the actual FDL descriptor variable. It is not necessary to modify fdl_descriptor.c. Further details on FDL runtime configuration can be found in the RH850 FDL T01 user manual.

Please recall that the EEL operates on virtual blocks rather than physical blocks (see Section 2.2.2.1). However, the aforementioned parameters of the FDL descriptor relate to physical blocks (Data Flash block size for RH850 is 64 byte). Therefore, when using the EEL it is strongly recommended to relate the specification of the defines `EEL_POOL_START` and `EEL_POOL_SIZE` to an additional define for the virtual block size. Thereby, a proper alignment of the EEL virtual blocks in the FDL pool can be ensured. For proper operation the EEL requires at least 4 virtual blocks.

Example 1:
FDL descriptor setup for a device with 32kB Data Flash and a virtual block size of 61 physical blocks. The EEL uses the virtual blocks from 2 to 5 for operation, while virtual blocks 0, 1, 6 and 7 can be addressed directly by the FDL.

```
#define CPU_FREQUENCY_MHZ        (80)
#define FDL_POOL_SIZE            (512)
/* FDL pool will use 32KB, from wich EEL pool occupies area:
START:        2 * 61 * 64 = 7808 till
END:          4 * 61 * 64 + 7808 = 23424 */
#define EEL_VIRTUALBLOCKSIZE     (61u)
#define EEL_POOL_START           (2u * EEL_VIRTUALBLOCKSIZE)
#define EEL_POOL_SIZE            (4u * EEL_VIRTUALBLOCKSIZE)
```

Example 2:
FDL descriptor setup for a device with 32kB Data Flash and a virtual block size of 32 physical blocks. The EEL uses the complete Data Flash for EEPROM emulation.

```
#define CPU_FREQUENCY_MHZ        (80)
#define FDL_POOL_SIZE            (512)
/* FDL pool will use 32KB, from wich EEL pool occupies area:
START:         0 * 32 * 64 = 0 till
END:          16 * 32 * 64 - 1 = 32767 */
#define EEL_VIRTUALBLOCKSIZE     (32u)
#define EEL_POOL_START           ( 0u * EEL_VIRTUALBLOCKSIZE)
#define EEL_POOL_SIZE            (16u * EEL_VIRTUALBLOCKSIZE)
```

4.2.2 EEL Runtime Configuration Parameters

The following settings should be configured by the user inside the EEL descriptor:

- **EEL_CONFIG_VBLK_SIZE:**
  This define describes how many physical Flash blocks are merged together to one virtual block (see Section 2.2.2.1), which are used by the EEL to manage the ring buffer. The following relation between physical blocks, virtual blocks and EEL pool size must match:
  - The EEL pool size is defined in the FDL in terms of physical blocks; virtual blocks are unknown to the FDL (see Section 4.2.1).
  - The number of virtual blocks is defined by: `EEL_POOL_SIZE / EEL_CONFIG_VBLK_SIZE`
  - `EEL_POOL_SIZE` must be a multiple of `EEL_CONFIG_VBLK_SIZE`.

Value Range:	Min:	16	(a virtual block needs to be at least 1kB)
	Max:	`EEL_POOL_SIZE` / 4	

  Recommendation: Selecting a proper virtual block size is a non-trivial task and requires some evaluation of the behaviour of the actual application. As a rule of thumb, virtual blocks should be at least of 2kB size for a smooth operation of the library. More hints on how to select a proper virtual-block size can be found in Section 5.4.1, “EEL Pool Configuration”.

- **EEL_CONFIG_VBLK_CNT_REFRESH_THRESHOLD:**
  The EEL requires prepared blocks (passive pool) to be able to accept new DS Write requests by the user application. The background supervision process will provide these blocks during runtime. As the supervision process has a lower priority than the user DS Write operation, it will be displaced by this. A fast sequence of DS Write operations might lead to using up the prepared space because the supervision will not get the time to prepare new space in between. So, a threshold is defined which determines the number of blocks that the supervision process shall provide in order to overcome situations where DS Write operations use up the free space faster than the supervision process can provide new one.
  Increasing the threshold allows longer sequences of DS Write operations until the prepared space is used up. Reducing the threshold improves the Flash usage as written data sets stay longer in the ring buffer and need less refresh copy operations. When the threshold is set too low and the ring buffer gets full, the library will return a pool full error and block further write operations until the supervision had enough time to prepare at least one additional Flash block.
  The `EEL_CONFIG_VBLK_CNT_REFRESH_THRESHOLD` parameter is specified in terms of number of virtual blocks.

Value range:	Min:	2	(required for proper EEL operation)
	Max:	`EEL_POOL_SIZE` - 2	

  Example: On a threshold of 6 the EEL will always try to have 6 prepared virtual blocks as passive pool in the ring buffer. This means that the user application could write 5 virtual blocks of data in sequence (one block must remain prepared for pool full situation handling).
  Recommendation: ~1/3 of the total available Flash blocks might be a reasonable starting point to evaluate the balance between long uninterrupted DS Write sequences (big threshold) and reducing the data copy effort on Refresh (low threshold). The service function `R_EEL_GetSpace` provides a tool to trace the available free space in the ring buffer during runtime enabling threshold optimization.

- **EEL_CONFIG_ERASE_SUSPEND_THRESHOLD:**
  When the EEL background operation executes the prepare process, the Data Flash block is erased. Any user read or write command will suspend the Flash erase. After command completion, the erase will be resumed again. Based on the Flash implementation, this erase suspend/resume flow is restricted. The erase operation might not finish, if it is interrupted continuously. The user application must be realized in a way that the erase operation once gets the time to complete, which means that the user application must provide a time frame as long as the worst case Flash block erase time in which the erase operation is not suspended. As long as the erase is not finished, the EEL cannot continue to provide new free passive pool space for further write operations. In order to signal too often erase suspends to the user application, a threshold can be configured by means of `EEL_CONFIG_ERASE_SUSPEND_THRESHOLD`. A user operation resulting in exceeding the threshold will return a warning “erase suspend overflow”. This is no hard error resulting in EEL reaction but just a signal to the user application to provide enough time to the EEL to finish the background operation.

Value range: Min: 1 (on every erase suspend the warning is returned)
Max: 0xFFFF

- **EEL_CONFIG_IDL_TABLE:**
  This define is used to specify the ID-L table (see also Section 3.5, "Data-Set Search and Read"). The table needs to be given as an array of the structure r_eel_ds_cfg_t (see Section 4.3.2.1, "r_eel_ds_cfg_t"). For each DS, an ID needs to be specified as positive integer number which is used to refer to the DS. Additionally, for each specified ID, the corresponding size of the DS needs to be given in terms of number of bytes (adjusted library internal to word boundary).

  { { ID1, size 1 }, { ID2, size 2 }, { ID3, size 3 }, ... }

Value range:	ID min:	1
	ID max:	0xFFFE
	Size min:	1
	Size max:	Virtual block size - block header size - REF entry size - separator size = `EEL_CONFIG_VBLK_SIZE` - 28 - 16 - 4 = `EEL_CONFIG_VBLK_SIZE` - 48

For setting these parameters it is sufficient to adapt the defines specified in eel_descriptor.h. The defined values are utilized in eel_descriptor.c in order to initialize the actual EEL descriptor variable. It is not necessary to modify eel_descriptor.c.

Example:
Data Flash size is 32kB, separated into virtual blocks of 2kB.
The EEL uses the complete Data Flash for the EEL pool. The refresh threshold is set to ~1/3 of 16 virtual blocks (i.e. 5). The erase shall be suspendable up to 10 times until the erase suspend warning is issued.

```
#define EEL_CONFIG_VBLK_SIZE                    (32)
#define EEL_CONFIG_VBLK_CNT_REFRESH_THRESHOLD   ( 5)
#define EEL_CONFIG_ERASE_SUSPEND_THRESHOLD      (10)

#define EEL_CONFIG_IDL_TABLE        {                       \
                                        { 0x1111, 0x0005 },    \
                                        { 0x2222, 0x0006 },    \
                                        { 0x3333, 0x0007 },    \
                                        { 0x4444, 0x0008 },    \
                                        { 0x5555, 0x0009 },    \
                                        { 0x6666, 0x000a },    \
                                        { 0x7777, 0x000b },    \
                                        { 0x8888, 0x000c },    \
                                        { 0x9999, 0x000d },    \
                                        { 0xaaaa, 0x0015 }     \
                                    }
```

4.3 Data Types

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

The EEL data types are defined in r_typedefs.h and r_eel_types.h.

4.3.1 Header file r_typedefs.h

4.3.1.1 Library-specific Simple-Type Definitions

Type definition:

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

Description: These simple types are used throughout the complete library API for passing of integer parameters.

4.3.2 Header file e_eel_types.h

4.3.2.1 r_eel_ds_cfg_t

Type definition:

```
typedef struct R_EEL_DS_CFG_T {
    uint16_t             ID_u16;
    uint16_t             len_u16;
} r_eel_ds_cfg_t;
```

Description: The structure defines the ID-L table elements. The user application needs to set up an array of this structure to define all supported DSs (refer to Section 4.2.2, "EEL Runtime Configuration Parameters" for details).

Member / Value:

Member / Value	Description
ID_u16	dataset ID
len_u16	length of the DS in byte

4.3.2.2 r_eel_descriptor_t

Type definition:

```
typedef struct R_EEL_DESCRIPTOR_T {
    uint16_t          vBlkRefreshThreshold_u16;
    const uint16_t    *IDLTab_pastr;
    uint16_t          *IDXTab_pau16;
    uint16_t          IDLTabIdxCnt_u16;
    uint16_t          eraseSuspendThreshold_u16;
 } r_eel_descriptor_t;
```

Description: The EEL descriptor is used to specify certain general behavior of the EEL as well as the actual datasets. In case that the eel_descriptor.c is used, it is not necessary to utilize this type directly. It is sufficient to modify eel_descriptor.h according to the application requirements. Please refer to Section 4.2.2, "EEL Runtime Configuration Parameters" for details.

Member / Value:

Member / Value	Description
vBlkRefreshThreshold_u16	Refresh threshold
*IDLTab_pastr	pointer to the IDL table
*IDXTab_pau16	pointer to the IDX table
IDLTabIdxCnt_u16	number of IDL/IDX table entries
eraseSuspendThreshold_u16	erase suspend threshold

4.3.2.3 r_eel_operation_mode_t

Type definition:

```
typedef enum R_EEL_OPERATION_MODE_T {
    R_EEL_OPERATION_MODE_NORMAL,
    R_EEL_OPERATION_MODE_LIMITED
} r_eel_operation_mode_t;
```

Description: This type is specifies the supported operation modes of the library. Please refer to 3.8, "Limited Operation Mode" for a more detailed specification of the modes.

Member / Value:

Member / Value	Description
R_EEL_OPERATION_MODE_ NORMAL	Normal operation mode
R_EEL_OPERATION_MODE_ LIMITED	Mode with limited Refresh performance with incomplete ID-L table

4.3.2.4 r_eel_status_t

Type definition:

```
#define R_EEL_WRN      0x10
#define R_EEL_ERR      0x20

typedef enum R_EEL_STATUS_T {
    R_EEL_OK                           = 0x00u,
    R_EEL_BUSY                         = 0x01u,
    R_EEL_ERR_ERASESUSPEND_OVERFLOW    = R_EEL_WRN+0x00u,
    R_EEL_ERR_FIX_DONE                 = R_EEL_WRN+0x03u,
    R_EEL_ERR_CONFIGURATION            = R_EEL_ERR+0x00u,
    R_EEL_ERR_PARAMETER                = R_EEL_ERR+0x01u,
    R_EEL_ERR_REJECTED                 = R_EEL_ERR+0x02u,
    R_EEL_ERR_ACCESS_LOCKED            = R_EEL_ERR+0x03u,
    R_EEL_ERR_NO_INSTANCE              = R_EEL_ERR+0x04u,
    R_EEL_ERR_POOL_FULL                = R_EEL_ERR+0x05u,
    R_EEL_ERR_FLASH_ERROR              = R_EEL_ERR+0x06u,
    R_EEL_ERR_INTERNAL                 = R_EEL_ERR+0x07u,
    R_EEL_ERR_POOL_EXHAUSTED          = R_EEL_ERR+0x08u,
    R_EEL_ERR_POOL_INCONSISTENT       = R_EEL_ERR+0x09u,
    R_EEL_ERR_COMMAND                 = R_EEL_ERR+0x0au
} r_eel_status_t;
```

Description: Function and operation error codes. Please refer to the Sections 4.4 "Functions" and 4.5 "Commands" for a detailed explanation of the errors root cause, and remedy.

`R_EEL_WRN` and `R_EEL_ERR` are defined to distinguish between warnings and errors.

Member / Value:

Member / Value	Description
R_EEL_OK	Operation ended successfully
R_EEL_BUSY	EEL is busy (operation on-going)
R_EEL_ERR_ERASESUSPEND_ OVERFLOW	Warning: An on-going Flash block erase has been suspended too often
R_EEL_ERR_FIX_DONE	Warning: a fix has been done during start-up
R_EEL_ERR_CONFIGURATION	Error: Wrong library configuration
R_EEL_ERR_PARAMETER	Error: Parameter error on operation invocation
R_EEL_ERR_REJECTED	Error: Operation rejected due to busy EEL
R_EEL_ERR_ACCESS_LOCKED	Error: operation not possible, Read/Write access locked
R_EEL_ERR_NO_INSTANCE	Error: No DS instance in the EEL pool
R_EEL_ERR_POOL_FULL	Error: Pool is full, Write operation blocked
R_EEL_ERR_FLASH_ERROR	Error: Flash write/erase error
R_EEL_ERR_INTERNAL	Error: Internal undefined error
R_EEL_ERR_POOL _INCONSISTENT	Error: Pool is inconsistent
R_EEL_ERR_COMMAND	Error: Unknown command

4.3.2.5 r_eel_command_t

Type definition:

```
typedef enum R_EEL_COMMAND_T {
    R_EEL_CMD_READ,
    R_EEL_CMD_WRITE,
    R_EEL_CMD_WRITE_INC,
    R_EEL_CMD_INVALIDATE,
    R_EEL_CMD_WRITE_IMM,
    R_EEL_CMD_WRITE_INC_IMM,
    R_EEL_CMD_INVALIDATE_IMM,
    R_EEL_CMD_FORMAT,
    R_EEL_CMD_CLEANUP
} r_eel_command_t;
```

Description: The enumeration represents the EEL commands to invoke the different EEL operations. For a detailed description of each command , please refer to Section 4.5 "Commands".

Member / Value:

Member / Value	Description
R_EEL_CMD_READ	Read command
R_EEL_CMD_WRITE	Write command
R_EEL_CMD_WRITE_INC	Incremental write command
R_EEL_CMD_INVALIDATE	Invalidation command
R_EEL_CMD_WRITE_IMM	Immediate write command
R_EEL_CMD_WRITE_INC_IMM	Immediate incremental write command
R_EEL_CMD_INVALIDATE_IMM	Immediate invalidate command
R_EEL_CMD_FORMAT	Format command (format the EEL pool)
R_EEL_CMD_CLEANUP	Clean-up command (clean-up, defragment the EEL pool)

4.3.2.6 r_eel_request_t

Type definition:

```
typedef volatile struct R_EEL_REQUEST_T {
    uint8_t *        address_pu08;
    uint16_t         identifier_u16;
    uint16_t         length_u16;
    uint16_t         offset_u16;
    r_eel_command_t  command_enu;
    r_eel_status_t   status_enu;
} r_eel_request_t;
```

Description: Request structure, used as parameter for the function `R_EEL_Execute` in order to invoke an EEL operation. Please refer to Section 3.1.3, "Request-Response oriented Dialog" for the structure usage overview and Section 4.5, "Commands" for the structure usage.

Member / Value:

Member / Value	Description
address_pu08	Data address (read destination, write source)
identifier_u16	Dataset ID
length_u16	Number of words to read
offset_u16	Data offset within the DS
command_enu	Command to execute
status_enu	Status of the command to execute

4.3.2.7 r_eel_access_status_t

Type definition:

```
typedef enum R_EEL_ACCESS_STATUS_T {
    R_EEL_ACCESS_LOCKED,
    R_EEL_ACCESS_READ_ONLY,
    R_EEL_ACCESS_READ_WRITE,
    R_EEL_ACCESS_UNLOCKED,
    R_EEL_ACCESS_UNDEFINED
} r_eel_access_status_t;
```

Description: Enumeration describing the access status of the EEL. Please refer to Section 3.6 "Driver Status" for the access status details.

Member / Value:

Member / Value	Description
R_EEL_ACCESS_LOCKED	Read and write access are disabled
R_EEL_ACCESS_READ_ONLY	Only read access to DSs
R_EEL_ACCESS_READ_WRITE	Read and write access, but limited performance
R_EEL_ACCESS_UNLOCKED	Full access
R_EEL_ACCESS_UNDEFINED	No valid status, this is used library internal only

4.3.2.8 r_eel_operation_status_t

Type definition:

```
typedef enum R_EEL_OPERATION_STATUS_T {
    R_EEL_OPERATION_PASSIVE,
    R_EEL_OPERATION_IDLE,
    R_EEL_OPERATION_BUSY,
    R_EEL_OPERATION_STARTUP,
    R_EEL_OPERATION_SUSPENDED,
    R_EEL_OPERATION_UNDEFINED
} r_eel_operation_status_t;
```

Description: Enumeration describing the operation status of the EEL. Please refer to Section 3.6 “Driver Status” for the access status details.

Member / Value:

Member / Value	Description
R_EEL_OPERATION_PASSIVE	EEL passive, all operations and commands locked
R_EEL_OPERATION_IDLE	EEL up and running, no internal operation ongoing
R_EEL_OPERATION_BUSY	EEL up and running, either user or background operations are served
R_EEL_OPERATION_STARTUP	EEL is in start-up processing
R_EEL_OPERATION_SUSPENDED	EEL is suspended by a user command
R_EEL_OPERATION_UNDEFINED	No valid status, this is used library internal only

4.3.2.9 r_eel_driver_status_t

Type definition:

```
typedef struct R_EEL_DRIVER_STATUS_T {
    r_eel_operation_status_t operationStatus_enu;
    r_eel_access_status_t    accessStatus_enu;
    r_eel_status_t           backgroundStatus_enu;
} r_eel_driver_status_t;
```

Description: Structure to contain the full driver status information. Please refer to Section 3.6 “Driver Status” to read the status information.

Member / Value:

Member / Value	Description
operationStatus_enu	EEL background operational status
accessStatus_enu	EEL commands access status
backgroundStatus_enu	EEL background operations error status

4.4 Functions

Due to the request-oriented interface of the EEL, the functional interface is very narrow. Beside the initialization/start-up and some administrative functions, the access to the EEL pool concentrates on two functions only: `R_EEL_Execute` and `R_EEL_Handler`.

All EEL interface functions are prototyped in the header file r_eel.h.

4.4.1 R_EEL_Init

Outline: Initialize the EEL variables; prepare the EEL for starting up.

Interface: C Interface

```
r_eel_status_t R_EEL_Init( const r_eel_descriptor_t* descriptor_pstr,
                           r_eel_operation_mode_t    opMode_enu);
```

Parameters

Argument	Type	Access	Description
descriptor_pstr	const r_eel_descriptor_t*	R	EEL run-time configuration structure. See Section 4.2.2 “EEL Runtime Configuration Parameters” for details of the library configuration.

Argument	Type	Access	Description
opMode_enu	r_eel_operation_mode_t	R	EEL operation mode configuration. See 3.8 “Limited Operation Mode”.

Return value

Type	Description
r_eel_status_t	Function execution result. The table below explains the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	function finished successfully
		reason	-
		remedy	-
R_EEL_ERR_CONFIGURATION	error	meaning	Initialization of the library failed. Further EEL functionality is blocked
		reason	The configuration checks found a problem in the descriptor configuration, passed to the function. Checks done: • EEL pool size (defined in FDL descriptor) must be a multiple as the virtual block size `vBlkSize_u16` • `vBlkRefreshThreshold_u16` must be >= 2 • remaining active pool must be >= 2 bocks • Max DS size must be as described in 4.2.2, "EEL Runtime Configuration Parameters"
		remedy	Fix the wrong configuration
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Initialization of the library failed. Further EEL functionality is blocked
		reason	The underlying FDL is busy with any Flash operation or is suspended
		remedy	Retry the initialization, when the FDL is idle

Pre-conditions: EEL initialization is not possible when Flash operations are active in order to avoid wrong synchronization between FDL and EEL. So:

- `R_FDL_Init` needs to be called before `R_EEL_Init` may be called.
- FDL must be idle

Post-conditions: Library internal variables are initialized, but the emulation processing is not yet started.

Description: The function is executed before any execution of other EEL functions. It does descriptor variable configuration checks and then initializes the basic internal variables according to the parameters passed to the library.

Example: eel_rtConfiguration is configured globally in eel_descriptor.c.

```
ret = EEL_Init (eel_rtConfiguration, EEL_OPMODE_FULL);

if (EEL_OK != ret)
{
    /* Error treatment */
}
```

4.4.2 R_EEL_Startup

Outline: Resets and starts the EEL state machine. The library start-up sequence is initiated.

Interface: C Interface

```
r_eel_status_t R_EEL_Startup (void);
```

No Parameters

Return value

Type	Description
r_eel_status_t	Function execution result. The table below explains the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Function finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Starting the state machine failed. State machine status is unchanged.
		reason	See pre-conditions.
		remedy	Fix the flow of function handling.

Pre-conditions: The library must be initialized. Call `R_EEL_Init` before execution.
The library may not already be active (function `R_EEL_Startup` already called).
In case of re-initialization, the function `R_EEL_ShutDown` must be called before `R_EEL_Init` and `R_EEL_Startup`

Post-conditions: None

Description: This function starts the EEL state machine and initiates execution of the start-up process.
Please note that this function does not execute but only initiates the start-up sequence of the library. By consecutive `R_EEL_Handler` calls, the library passes the start-up status and enters the operational status (see Section 3.7, “Start-up Processing”).

Example: Option: Wait after `R_EEL_Startup` until the library is completely up and running

```
r_eel_driver_status_t driverStatus_str;

ret = R_EEL_Init();

if (R_EEL_OK == ret)
{
        R_EEL_Startup();
}
else
{
        /*error treatment */
}

do
{
    R_EEL_Handler();
    R_EEL_GetDriverStatus (&driverStatus_str);
}
/* Wait until the system is completely up and running (or error) */
while (R_EEL_OPERATION_STARTUP == driverStatus_str.operationStatus_enu);

/* Error check */
if (R_EEL_OK != driverStatus_str.errorStatus_enu)
{
        /* Error handler */
         . . .
}
```

Example: Option: Wait after `R_EEL_Startup` until the library at least partially unlocked

```
r_eel_driver_status_t driverStatus_str;

ret = R_EEL_Init();

if (R_EEL_OK == ret)
{
        R_EEL_Startup();
}
else
{
        /*error treatment */
}

do
{
    R_EEL_Handler();
    R_EEL_GetDriverStatus (&driverStatus_str);
}
/* Wait until early read/write is possible (or error) */
while (    (R_EEL_OPERATION_STARTUP == driverStatus_str.operationStatus_enu)
        && (R_EEL_ACCESS_LOCKED == driverStatus_str.accessStatus_enu));

/* Error check */
if (R_EEL_OK != driverStatus_str.errorStatus_enu)
{
        /* Error handler */
         . . .
}
```

4.4.3 R_EEL_ShutDown

Outline: This function initiates deactivation of the EEL state machine.

Interface: C Interface

```
eel_status_t EEL_ShutDown (void);
```

No Parameters

Return value

Type	Description
r_eel_status_t	Function execution result. The table below explains the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Function finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Shut down request failed. State machine status is unchanged.
		reason	See pre-conditions.
		remedy	Fix the flow of function handling.

Pre-conditions: The library must be active. So, `R_EEL_Startup` must have been called before.
The library may not be suspended.

Post-conditions: Library access status is locked. No further user commands are accepted.

Description: This function initiates the deactivation of the EEL state machine.

After this function the `R_EEL_Handler` need to be consecutively executed in order to complete running processes and to set the state machine status to passive.

The individual effects on running processes is as follows:

- Start-up:
  The process is stopped after a sub-process execution.
- Refresh
  An ongoing DS write is finished, then the refresh is stopped.
- Prepare
  The prepare is finished in order not to waste a Flash erase cycle.
- User DS write
  An on-going DS write is finished.
- User DS read
  An on-going DS read is finished.

Example:

```
r_eel_driver_status_t driverStatus_str;

/* ... */

ret = R_EEL_ShutDown();
if (R_EEL_OK != ret)
{
        /* Error treatment */
}

/* Wait until operation end */
do
{
    R_EEL_Handler();
    R_EEL_GetDriverStatus (&driverStatus_str);
}
while (R_EEL_OPERATION_PASSIVE != driverStatus_str.operationStatus_enu );

/* Error check */
if (R_EEL_OK != driverStatus_str.errorStatus_enu)
{
       /* Error handler */
        . . .
}
```

4.4.4 R_EEL_SuspendRequest

Outline: This function requests suspension of the EEL operations and puts the EEL in a passive state.

Interface: C Interface

```
r_eel_status_t  R_EEL_SuspendRequest (void);
```

Arguments: No parameters

Return value

Type	Description
r_eel_status_t	Function execution result. The table below explains the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Function finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Suspend request failed. State machine status is unchanged.
		reason	See pre-conditions.
		remedy	Fix the flow of function handling.

Pre-conditions: The library must be active. Call `R_EEL_Init` before execution.

Post-conditions: Library access status is locked. So, no further user commands are accepted.

Description:

This function requests suspension of the EEL operations and puts the EEL in a passive state.

After this function the `R_EEL_Handler` need to be executed consecutively in order to complete running processes and to set the state machine status to passive. Then the driver status is set to `R_EEL_OPERATION_SUSPENDED`.

The individual effects on running processes is as follows:

- Start-up
  The process is stopped after a sub-process execution.
- Refresh
  An on-going DS Write is finished, then the Refresh is stopped.
- Prepare
  The Prepare is finished in order not to waste a Flash erase cycle.
- User DS write
  An on-going DS Write is finished.
- User DS read
  An on-going DS read is finished.

Please refer to Section →3.9 for a more detailed description of the suspend/resume mechanism.

Example:

```
r_eel_driver_status_t driverStatus_str;

/* ... */

ret = R_EEL_SuspendRequest();
if (R_EEL_OK != ret)
{
      /* Error treatment */
}

/* Wait until operation end */
do
{
    R_EEL_Handler();
    R_EEL_GetDriverStatus (&driverStatus_str);
}
while (R_EEL_OPERATION_SUSPENDED != driverStatus_str.operationStatus_enu);

/* Do other Flash operations by directly using the FDL API, read Data Flash user
pool contents (direct read by CPU) or bring the device in power safe mode */
   ...


ret = R_EEL_ResumeRequest();
if (R_EEL_OK != ret)
{
      /* Error treatment */
}


/ Continue with EEL operations */
```

4.4.5 R_EEL_ResumeRequest

Outline: This function requests resuming the EEL operations after suspend.

Interface: C Interface

```
ret =  R_EEL_ResumeRequest (void);
```

Arguments: No Parameters

Return value

Type	Description
r_eel_status_t	Function execution result. The table below explains the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Function finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Starting the state machine failed. State machine status is unchanged.
		reason	See pre-conditions.
		remedy	Fix the flow of function handling.

Pre-conditions: The library must be suspended. This implies that not only `R_EEL_SuspendRequest` has been executed before, but also that there have been enough R_EEL_Handler calls in order to drive the library into the suspended state.

Post-conditions: None

Description: This function requests resuming the EEL operations after suspend. The resume handling is done by the R_EEL_Handler function, i.e. it is necessary to call the handler function until the operation status changed to a status differing from `R_EEL_OPERATION_SUSPENDED`.

Example: See 4.4.4, “R_EEL_SuspendRequest”

4.4.6 R_EEL_Execute

Outline: This function initiates an EEL user command.

Interface: C Interface

```
void R_EEL_Execute (r_eel_request_t * request_pstr);
```

Parameters

Argument	Type	Access	Description
request_pstr	r_eel_request_t*	R/W	The request structure defines the command to be executed. The concrete usage of the request structure is command specific and therefore described in Section 4.5, “Commands” individually for each command. Furthermore, the request structure contains the status variable for the operation result storage (`status_enu`). Setting of the variable after function end is described in the post conditions.

No return value

Pre-conditions: The library has to be initialized and started up successfully before any command can be accepted via R_EEL_Execute.

Post-conditions: The function updates the request structure status variable according to the function interpretation of the command and its starting (execution) conditions. Please refer to Section 4.5, “Commands” for the possible result values of the individual commands.

Description:

This function initiates an EEL user command. The command type and parameters are passed to the EEL by the request structure.

The underlying request-response concept is introduced in Section 3.1.3.

A detailed description of all available commands can be found in Section 4.5. Please note that the meaning of the different request structure elements depends on the command. Before execution, the request structure is checked for plausibility. However, the error codes (returned as a member of the request structure) are command-specific as well. Please have a look at the individual command description for details.

This function only starts a process according to the command to be executed. The processes must be controlled and stepped forward by the state machine handler function `R_EEL_Handler`.

One advantage of the request-response oriented command execution is that multiple independent request structures can be used. Thereby, different commands can be prepared and monitored in parallel, although a real parallel execution is not possible. The priorities in which concurring commands are executed are described in Section 3.3.

Note 1:
The request structure might be read by the EEL state machine any time before the command execution is not yet completed. Therefore, it is imperative to ensure a sufficient lifetime of the request variable and not to modify its members as long as a command is being processed.

Note 2:
The user application can either react directly on the errors returned by the `R_EEL_Execute` function or call `R_EEL_Handler` as long as the status is busy and react on errors then. The errors set on `EEL_Execute` are not reset and the handler execution does not do additional operations in case of an error already set.

Example:

```
r_eel_request_t myRequest_str;
uint8_t         buffer[0x100]={ 0 };

/* Start the write operation */
myRequest_str.address_pu08    = (uint8_t*)(&buffer);   /* Set receive buffer */
myRequest_str.identifier_u16 = 10u;
myRequest_str.command_enu     = R_EEL_CMD_WRITE;

R_EEL_Execute (&myRequest_str);

/* Wait until operation end */
while (R_EEL_BUSY == myRequest_str.status_enu)
{
       R_EEL_Handler();
}

/* Error check */
if (EEL_OK != myRequest_str.status_enu)
{
       /* Error handler */
        . . .
}
```

4.4.7 R_EEL_Handler

Outline: This function handles the EEL state machine.

Interface: C Interface

```
void R_EEL_Handler (void);
```

No Parameters

No Return value

Pre-conditions: The library has to be initialized by means of `R_EEL_Init` before `R_EEL_Handler` may be called.

Post-conditions: The handler function executes user commands as well as background processes. By that, it modifies the Data Flash contents as well as the user operation status as well as background operation (driver) status.

Description: This function handles the complete EEL state machine including user and background operations management.

On the one hand, the function `R_EEL_Handler` needs to be called regularly in order to drive pending commands and observe their progress. Thereby, the command execution is performed state by state. When a command execution is finished, the request status variable (structural element `status_enu` of `r_eel_request_t`) is updated and contains the status/error code of the corresponding command execution. Please have a look at Section 3.1.4 for a more detailed description of the principles of the handler-oriented command execution.

On the other hand, background processes are also executed by means of the handler. Hence it is mandatory to call the `R_EEL_Handler` regularly in case that there are no user commands pending.

Example: See Section 4.4.6, “R_EEL_Execute”.

4.4.8 R_EEL_GetEraseCounter

Outline: This function reads the current erase counter of the ring buffer / EEL pool.

Interface: C Interface

```
r_eel_status_t R_EEL_GetEraseCounter (uint32 *counter_pu32);
```

Parameters

Argument	Type	Access	Description
counter_pu32	uint32 *	W	Pointer to the destination buffer to store the current erase counter.

Return value

Type	Description
r_eel_status_t	Function execution result. See table below regarding the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Function finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Erase counter could not be read.
		reason	See pre-conditions.
		remedy	Fix the flow of function handling.

Pre-conditions: The library must be unlocked which means

- The library has be initialized and started up completely (i.e. access status != R_EEL_ACCESS_LOCKED).
- Shut down processing (R_EEL_ShutDown) or suspend processing (R_EEL_Suspend) may not have been initiated beforehand.

Post-conditions: None

Description: This function reads the erase counter of the currently active virtual block. Due to the ring buffer handling of the EEL pool, the erase counter of all other blocks only differs from the active block in the range of -1 to +1.

Note:
The erase counter is counting the ring buffer loops. As long as the ring buffer is normally handled by the library, the erase counter is counted up. Of cause, the erase counter is as reliable as all EEPROM emulation data. It is handled by the library and any mistreatment outside the library (e.g. manual erase of the Flash) may destroy the erase counter.

Example:

```
eel_u32         eraseCounter;
r_eel_status_t ret;

ret = R_EEL_GetEraseCounter (&EraseCounter);

if (R_EEL_OK != ret)
{
       /* Error treatment */
}
```

4.4.9 R_EEL_GetDriverStatus

Outline: This function returns the state machine status into the driver status structure.

Interface: C Interface

```
r_eel_status_t R_EEL_GetStatus (r_eel_driver_status_t
                                   *driverStatus_str);
```

Parameters

Argument	Type	Access	Description
driverStatus_str	uint32 *	W	Pointer to the buffer to store the driver status structure. See Section 3.6 “Driver Status”.

Return value

Type	Description
r_eel_status_t	Function execution result. See table below regarding the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Function finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Driver status could not be read.
		reason	See pre-conditions.
		remedy	Fix the flow of function handling.

Pre-conditions: The library needs to be initialized by means of `R_EEL_Init` before the function `R_EEL_GetDriverStatus` may be called.

Post-conditions: None

Description: The `R_EEL_GetDriverStatus` function opens a way to check the internal status of the EEL driver before placing a request. It returns the EEL state machine status into the driver status structure. A detailed description of the different statuses and how to interpret them can be found in Section 3.6, “Driver Status”.

Example:

```
r_eel_driver_status_t      driverStatus_str;
r_eel_status_t             ret;

ret = R_EEL_GetDriverStatus (&driverStatus_str);

if (R_EEL_OK != ret)
{
        /* Error treatment */
}
```

4.4.10 R_EEL_GetSpace

Outline: This function returns the current free space in the EEL pool (prepared space for new data).

Note:

During transient conditions the space calculation is not accurate as currently running write operations may not be considered. The function is meant to be used only when EEL operation status is `R_EEL_OPERATION_IDLE`.

Interface: C Interface

```
r_eel_status_t R_EEL_GetSpace (uint32_t *space_pu32);
```

Parameters

Argument	Type	Access	Description
space_pu32	uint32 *	W	Pointer to the buffer to store the space value.

Return value

Type	Description
r_eel_status_t	Function execution result. See table below regarding the possible results.

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Function finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	The space information could not be read.
		reason	See pre-conditions.
		remedy	Fix the flow of function handling.

Pre-conditions: The library must be unlocked which means:

- The library has be initialized and started up completely (i.e. access status != `R_EEL_ACCESS_LOCKED`).
- Shut down processing (`R_EEL_ShutDown`) or suspend processing (`R_EEL_Suspend`) may not have been initiated beforehand.

Post-conditions: None

Description:

This function returns the current free space in the EEL ring buffer (prepared space for new data) into the user buffer `space_pu32`.

As the library always need to reserve one virtual block for refreshing data sets (copy from the ring buffer tail to the front), the space of this block is not considered for space computation.

Furthermore it is important to note that each dataset instance requires also space for the management data in the REF zone. Please refer to Section 2.3.3 “EEL Data-Sets” for a detailed description of the space requirement of a dataset instance.

Please also note that it cannot be guaranteed that a dataset instance can fit into the current active block and the next prepared block might to be used. Hence, whenever a dataset is written, the remaining free space might be reduced by more than the space required for the dataset instance. Also background processes can change the free space in the EEL pool. This makes it necessary to recheck the free space by means of the R_EEL_GetSpace function before issuing a command or calling the R_EEL_Handler. Otherwise the information is outdated.

Calculation base:
Free space = (no. of prepared blocks – 1) * (block size – block header – 1 word) + emaining space in the active block

Example:

```
uint32_t            space;
r_eel_status_t      ret;

ret = R_EEL_GetSpace (&space);

if (R_EEL_OK != ret)
{
       /* Error treatment */
}
```

4.4.11 R_EEL_GetVersionString

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

Interface: C Interface

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

No Parameters

Return value

Type	Description
(const uint8_t*)	Pointer to the version string, stored in EEL linker section. String format: “DH850T01xxxxxYZabcD” • Replace xxxxx by the used version. If no information is coded, the build is a generic library valid for different compilers. • “Y” coded the used memory/register model. If no information is coded, the build is a generic library valid for different models. • P: 22 register model • Q: 26 register model • R: 32 register model • Replace “Z “by “E” for engineering version or “V” for final version. • Additionally “abc” must be substitute by the library version numbers according to version Va.bc. • “D” is an optional character to identify different engineering versions.

Pre-conditions: None

Post-conditions: None

Description: This function returns the pointer to the library version string. The version string is a zero terminated string identifying the library. The version string is stored in the library code section.

Example:

```
/* Read library version */
uint8_t  *version_pu08;

version_pu08 = R_EEL_GetVersionString();
```

4.5 Commands

The following Sections describe the EEL commands that can be executed by the library. A command is initiated by the library function `R_EEL_Execute` utilizing a request data structure of type `r_eel_request_t` which contains the individual command details. An initiated command is controlled and driven forward by the library function `R_EEL_Handler`. The command status can be read from the request structure element `status_enu`. Figure 18 shows the schematic usage of the request structure as already introduced in Section 3.1.3.

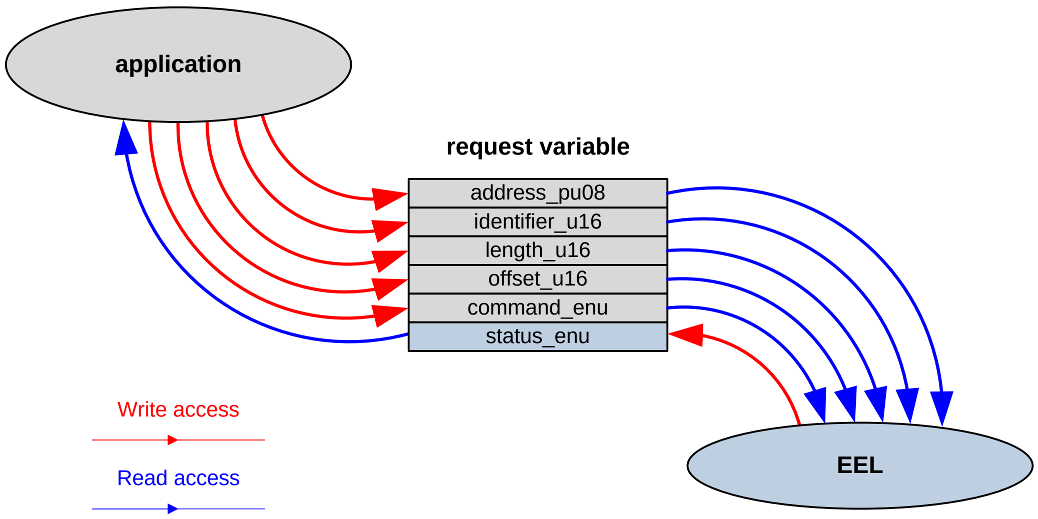


Figure 18: Schematic usage of the request variable (reprise of Figure 13)

Please note that the library may access the request structure during any `R_EEL_Execute` or `R_EEL_Handler` call. Therefore, it is on the one hand mandatory to ensure a proper lifetime of the request variable. On the other hand, it is imperative not to change the elements of the request variable as long as the command execution is ongoing.

Not all request structure elements are required for each command. While the elements `command_enu` and `status_enu` are mandatory for all commands, the usage of the other four elements is command specific and listed in Table 5.

Table 5: Command codes and usage of request variable members

Command identifier	address_pu08	identifier_u16	length_u16 offset_u16	Comment
R_EEL_CMD_READ	used	used	used	DS read
R_EEL_CMD_WRITE	used	used	unused	DS write
R_EEL_CMD_WRITE_INC	used	used	unused	incremental DS write
R_EEL_CMD_WRITE_IMM	used	used	unused	immediate DS write
R_EEL_CMD_WRITE_INC_IMM	used	used	unused	incremental immediate DS write
R_EEL_CMD_INVALIDATE	unused	used	unused	DS invalidation
R_EEL_CMD_INVALIDATE_IMM	unused	used	unused	immediate DS invalidation
R_EEL_CMD_FORMAT	unused	unused	unused	EEL pool format
R_EEL_CMD_CLEANUP	unused	unused	unused	EEL pool Clean-up

In general, all EEL commands can be handled in the same way as illustrated in Figure 19 (a corresponding code example can be found in Section 5.5, “Sample Application”):

1. The application fills up the private request variable `my_request` (command definition).
2. The application tries to initiate the command execution by `R_EEL_Execute(&my_request)`.
3. In case of a command rejection, the application has to call the `R_EEL_Handler` first to proceed the processing of the already pending request and can retry by continuing with step 2.

4. The application has to call `R_EEL_Handler` to proceed the EEL command execution as long the request is being processed (i.e. `my_request.status_enu == EEL_BUSY`).
5. After finishing the command (i.e. `my_request.status_enu != EEL_BUSY`) the application has to analyze the status to detect potential errors.

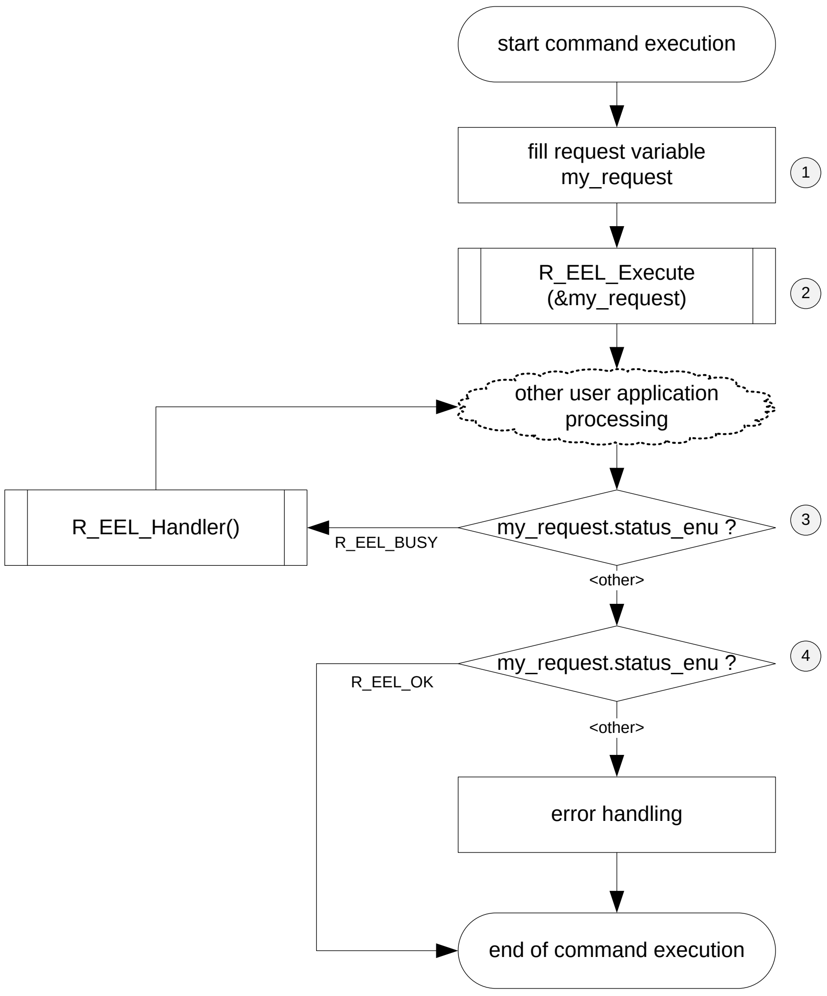


Figure 19: Generic command execution flow

Not all commands can/should be executed from every library state. For details, please refer to Section 3.7, "Start-up Processing".

For exemplary code showing how to use EEL commands, please refer to Section 5.5, "Sample Application".

4.5.1 R_EEL_CMD_READ — DS Read

This command reads a DS identified by a given ID and copies the read data to a read buffer provided by the user application. It is possible to read the complete DS or only fractions, defined by an offset and number of bytes.

The read command has the highest priority of all standard operations and can interrupt all write/invalidate operations.

Note:
Differing from other commands, the read command actions are completely done within one handler call. This will also result in longer handler function execution time, depending on the amount of Bytes to read from the Flash and depending on the Data flash access speed.

Table 6: Configuration of the request structure for DS Read

Request structure element	Data	Comment
command_enu	R_EEL_CMD_READ	Read command
address_pu08	{pointer to uint8_t buffer}	Pointer to the read data destination buffer.
identifier_u16	{uint16 number}	Defines the ID of the DS to read
offset_u16	{uint16 number}	Offset of the read data from the DS base address. Number range: 0 <= number < DS size
length_u16	{uint16 number}	Number of bytes to read Number range: 0 < number < (DS size – offset_u16)
status_enu	-	This is an output member. It contains the status of the command during and after the execution. Possible values are described in the next table.

Table 7: Possible statuses of DS-read command returned to status_enu

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Command finished successfully.
		reason	-
		remedy	-
R_EEL_BUSY <*1>	normal	meaning	Command started successfully.
		reason	-
		remedy	-
R_EEL_ERR_ERASESUSPEND_ OVERFLOW	warning	meaning	Warning: Background Flash block erase interrupted too often.
		reason	Each DS read or write command will interrupt a possibly on-going background Flash erase operation (part of the prepare process). If the threshold (acceptable number of interruptions) defined by `eraseSuspendThreshold_u16` (see Section 4.2.2, "EEL Runtime Configuration Parameters") is exceeded, this warning is returned.
		remedy	The return value is just a warning without impact on the operations. Reduce the load to the EEL caused by Read/Write commands to give potential background Flash erase the time to finish.

Status	Class	Background and handling	
R_EEL_ERR_PARAMETER <*1>	error	meaning	Execution of the requested command is rejected.
		reason	There is/are wrong parameter(s) in the request structure. Parameter checks done by the library: • `identifier_u16` invalid (configured ID not available in the ID-L table) • `offset_u16` + `length_u16` > DS size configured in the ID-L table
		remedy	Fix the parameters.
R_EEL_ERR_REJECTED <*1>	error	meaning	Execution of the requested command is rejected.
		reason	A read operation is ongoing and not yet finished. Invoking a 2nd read command is not possible.
		remedy	Request the command again after the end of the 1st read operation.
R_EEL_ERR_ACCESS_LOCKED <*2>	error	meaning	Execution of the requested command is rejected.
		reason	The library internal status does not allow execution of the command. Either it’s in Start-up processing and the access is not yet unlocked or the library that was previously unlocked has been locked by suspend or shut-down processing (User requests) or a library internal problem.
		remedy	If the error is caused by a library internal problem, use the function `R_EEL_GetDriveStatus` to read the library status. Then, react according to the observed error (see Section 3.6, "Driver Status"). The other described reasons are based on wrong library handling by the user application. In that case, stop the application, investigate in the root cause and fix the application.
R_EEL_ERR_NO_INSTANCE <*1>	normal or error	meaning	The read command could not find any valid DS with the requested ID.
		reason	Either the EEL pool contains no DS with that ID at all (ID was not yet written) or the last DS instance with that ID is an invalidation instance (data was explicitly invalidated on application request).
		remedy	Write a new DS instance with the concerning ID.
R_EEL_ERR_INTERNAL <*2>	error	meaning	A library internal problem occurred, that cannot be related to a concrete root cause. The library will be locked.
		reason	Some library internal checks that should never fail, failed. These can be checks on hardware or software values.

Status	Class	Background and handling	
		remedy	Stop emulation and investigate the root cause. In some cases, e.g. in case of external influence on the EEL or FDL variables or the Flash hardware (e.g. caused by wild running application pointers or PC), re-initialization of the FDL and EEL may help. A reasonable proceeding might be: During development: Stop the emulation and investigate in the root cause In the field: 2~3 times try to re-initialize the library (Or reset the device). If the problem still exists, stop the emulation and investigate in the root cause.

The command status can be updated by the command initiating function `R_EEL_Execute` as well as by the handler function `R_EEL_Handler`. The notes describe, which status value can be set by which function:

<*1>: Value can be set by `R_EEL_Execute` only.

<*2>: Value can be set by both functions.

No note: Value can be set be `R_EEL_Handler` only.

4.5.2 R_EEL_CMD_WRITE — DS write

This command reads data from an application buffer and writes the data to a DS identified by a given ID.

The write command will write always a complete DS and so always requires the complete data in the buffer. A partial write is not possible.

Table 8: Configuration of the request structure for DS Write

Request structure element	Data	Comment
command_enu	R_EEL_CMD_WRITE	Write command.
address_pu08	{pointer to uint8_t buffer}	Pointer to the write data source buffer.
identifier_u16	{uint16 number}	Defines the ID of the DS to write.
offset_u16	-	
length_u16	-	
status_enu	-	This is an output member. It contains the status of the command during and after the execution. Possible values are described in the next table.

Table 9: Possible statuses of DS-write command returned to status_enu

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Command finished successfully.

Status	Class	Background and handling	
		reason	-
		remedy	-
R_EEL_BUSY <*1>	normal	meaning	Command started successfully.
		reason	-
		remedy	-
R_EEL_ERR_ERASESUSPEND_ OVERFLOW	warning	meaning	Warning: Background Flash block erase interrupted too often.
		reason	Each DS read or write command will interrupt a possibly ongoing background Flash erase operation (part of the prepare process). If the threshold (acceptable number of interruptions) defined by `eraseSuspendThreshold_u16` (see Section 4.2.2, "EEL Runtime Configuration Parameters") is exceeded, the warning is returned.
		remedy	The return value is just a warning without impact on the operations. Reduce the load to the EEL caused by read/write commands to give potential background Flash erase the time to finish
R_EEL_ERR_PARAMETER <*1>	error	meaning	Execution of the requested command is rejected.
		reason	There is/are wrong parameter(s) in the request structure. Parameter checks done by the library: • `identifier_u16` invalid (configured ID not available in the ID-L table)
		remedy	Fix the parameters
R_EEL_ERR_REJECTED <*1>	error	meaning	Execution of the requested command is rejected.
		reason	A write, incremental write or invalidate command is ongoing and not yet finished. Those commands are all based on the same internal process and it is not allowed to invoke a 2nd command basing on the same process. Note: the immediate commands are based on another process and thus, invoking those is possible (and vice versa).
		remedy	Request the command again after the end of the former command.
R_EEL_ERR_ACCESS_LOCKED <*2>	error	meaning	Execution of the requested command is rejected.
		reason	The library internal status does not allow execution of the command. Either it is in start-up processing and the access is not yet unlocked or the library that was previously unlocked has been locked by suspend or shut-down processing (User requests) or a library internal problem.

Status	Class	Background and handling	
		remedy	If the error is caused by a library internal problem, use the function R_EEL_GetDriveStatus to read the library status. Then, react according to the observed error (see Section 3.6, "Driver Status"). The other described reasons are based on wrong library handling by the user application. In that case, stop the application, investigate in the root cause and fix the application
R_EEL_ERR_POOL_FULL	error	meaning	Execution of the requested command is rejected.
		reason	Due to subsequent write commands (incl. immediate and incremental as well as invalidation commands), the EEL pool got full (no more space to write new DS instances). The library has got no time (and R_EEL_Handler calls) to prepare new space in the pool by executing the background prepare and refresh processes.

Status	Class	Background and handling	
		remedy	Do repeated R_EEL_Handler calls until enough space is prepared again. The library provided 3 mechanisms to check if space is prepared which can be used depending on the application needs: • Use R_EEL_GetDriverStatus (default way): After each call of R_EEL_Handler call R_EEL_GetDriverStatus. The later function will return the driver status by a structure. When the library is busy with any background or user operation, the structure element operationStatus_enu will be set to R_EEL_OPERATION_BUSY (or R_EEL_OPERATION_STARTUP in the start-up phase). So continue calling the functions until operationStatus_enu shows R_EEL_IDLE. • Use R_EEL_GetSpace: After each call of R_EEL_Handler call R_EEL_GetSpace. The later function will calculate the space for new DS instances available in the pool. Call the functions until enough space is prepared to write the new DS. This method allows writing the new DS as soon as possible. • Repeat the command invocation: After each call of R_EEL_Handler call try again to invoke the write command. Repeat this until the command does not fail anymore. This method allows writing the new DS as soon as possible. Note: In addition to the methods mentioned above, the handler shall be called continuously even after reaching idle status. Only then, the library can execute background bit error checks (robustness feature to find and fix possible single bit errors in the Flash).
R_EEL_ERR_FLASH_ERROR	error	meaning	The emulation should be considered as defect as some Flash contents could not be erased or written.
		reason	A background Flash erase or write operation ended with an error. The Flash range has no defined status, allowing using it later on.
		remedy	Stop the emulation and investigate in the root cause.
R_EEL_ERR_INTERNAL <*2>	error	meaning	A library internal problem occurred, that cannot be related to a concrete root cause. The library will be locked.
		reason	Some library internal checks that should never fail, failed. These can be checks on hardware or software values.

Status	Class	Background and handling	
		remedy	Stop emulation and investigate the root cause. In some cases, e.g. in case of external influence on the EEL or FDL variables or the Flash hardware (e.g. caused by wild running application pointers or PC), re-initialization of the FDL and EEL may help. A reasonable proceeding might be: During development: Stop the emulation and investigate in the root cause In the field: 2~3 times try to re-initialize the library (Or reset the device). If the problem still exists, stop the emulation and investigate in the root cause

The command status can be updated by the command-initiating function `R_EEL_Execute` as well as by the handler function `R_EEL_Handler`. The notes describe, which status value can be set by which function:

<*1>: Value can be set by `R_EEL_Execute` only.

<*2>: Value can be set by both functions.

No note: Value can be set be `R_EEL_Handler` only.

4.5.3 R_EEL_CMD_WRITE_INC — Incremental DS write

This command compares data in an application buffer with the last DS instance identified with a given ID. On mismatch, the DS is written with the updated data from the application buffer. On data match, no write operation is executed.

Except the comparison before writing, the command completely matches the write command of Section 4.5.2, “R_EEL_CMD_WRITE — DS write ”. So, the configuration and the returned data are identically except the following:

Table 10: Configuration difference for DS incremental write (compared to DS Write)

Request structure element	Data	Comment
command_enu	R_EEL_CMD_WRITE_INC	Incremental write command.

4.5.4 R_EEL_CMD_WRITE_IMM — Immediate DS write

From functionality point of view, this command exactly matches the write command of Section 4.5.2, “R_EEL_CMD_WRITE — DS write ”.

The difference is a higher priority of the immediate command. So, an immediate command can interrupt a normal priority operation.

The configuration and the returned data are identically to 4.5.2, “R_EEL_CMD_WRITE — DS write ” except the following:

Table 11: Configuration difference for DS immediate write (compared to DS Write)

Request structure element	Data	Comment
command_enu	R_EEL_CMD_WRITE_IMM	Immediate write command.

4.5.5 R_EEL_CMD_WRITE_INC_IMM — Incremental immediate DS write

From functionality point of view, this command exactly matches the incremental write command of Section 4.5.3, “R_EEL_CMD_WRITE_INC — Incremental DS write”.

The difference is a higher priority of the immediate command. So, an immediate command can interrupt a normal priority operation.

The configuration and the returned data are identically to 4.5.2, “R_EEL_CMD_WRITE — DS write ” except the following:

Table 12: Configuration difference for DS immediate incremental write (compared to DS Write)

Request structure element	Data	Comment
command_enu	R_EEL_CMD_WRITE_INC_IMM	Immediate incremental write command

4.5.6 R_EEL_CMD_INVALIDATE — DS invalidation

This command invalidates the data of a formerly written DS with a given ID. Instead of erasing the data which is not possible (and not necessary) in an EEPROM emulation, the EEL will write a new DS Reference zone entry with invalidation marker and without data part. A later read command on the same ID will return `R_EEL_ERR_NO_INSTANCE` instead of reading any data.

Use case example:
A window position is stored in by the EEL. The position is invalidated before the window is moved. On moving end, the new position is stored again. In case of power fail during window moving, the application will know by the EEL return `R_EEL_ERR_NO_INSTANCE`, that there was a problem during window moving and the window position need to be calibrated new.

From implementation point of view, the invalidation command is very similar to the write process as described in Section 4.5.2, “R_EEL_CMD_WRITE — DS write ”. In fact, both commands use the same process.

The configuration and the returned data are identically to 4.5.2, “R_EEL_CMD_WRITE — DS write ” except the following:

Table 13: Configuration difference for DS invalidation (compared to DS Write)

Request structure element	Data	Comment
command_enu	R_EEL_CMD_INVALIDATE	DS invalidation command
address_pu08	-	

4.5.7 R_EEL_CMD_INVALIDATE_IMM — Immediate DS invalidation

From functionality point of view, this command exactly matches the invalidation command of Section 4.5.6, “R_EEL_CMD_INVALIDATE — DS invalidation”.

The difference is a higher priority of the immediate command. So, an immediate command can interrupt a normal priority operation.

The configuration and the returned data are identically to Section 4.5.2, “R_EEL_CMD_WRITE — DS write ” except the following:

Table 14: Configuration difference for DS invalidation (compared to DS Write)

Request structure element	Data	Comment
command_enu	R_EEL_CMD_INVALIDATE_IMM	Immediate DS invalidation command
address_pu08	-	

4.5.8 R_EEL_CMD_FORMAT — EEL Pool Format

The Data Flash contents must have the correct format in order to enable successful operation of the EEL. On the one hand, formatting can be done by an external tool chain: A reference contents is written to the Data Flash, e.g. using a flash programmer or debugging tool chain. On the other hand, the EEL provides the format command for that.

Note:
The format command overwrites any potential data in the EEL pool, which is then lost.

Differing from the read/write-related commands; this command is bound to slightly different conditions:

- Format is a singular command. No other user or background operation may be executed. This is ensured after the start-up processing, when the operation status is `R_EEL_IDLE` or `R_EEL_PASSIVE` (see Section 3.6, "Driver Status").
- The command may also be executed when the start-up processing failed (operation status is `R_EEL_PASSIVE`).

Table 15: Configuration of the request structure for the format command

Request structure element	Data	Comment
command_enu	R_EEL_CMD_FORMAT	Format command.
address_pu08	-	
identifier_u16	-	
offset_u16	-	
length_u16	-	
status_enu	-	This is an output member. It contains the status of the command during and after the execution. Possible values are described in the next table.

Table 16: Possible statuses of format command returned to status_enu

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Command finished successfully.
		reason	-
		remedy	-
R_EEL_BUSY <*1>	normal	meaning	Command started successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED <*1>	error	meaning	Execution of the requested command is rejected.

		reason	The library internal status does not allow execution of the command. The operation status is not `R_EEL_OPERATION_IDLE` or `R_EEL_OPERATION_PASSIVE`.
		remedy	Stop the emulation and investigate in the root cause.
R_EEL_ERR_FLASH_ERROR	error	meaning	The emulation should be considered as defect as some Flash contents could not be erased or written.
		reason	A background Flash erase or write operation ended with an error. The Flash range has no defined status, allowing using it later on.
		remedy	Stop the emulation and investigate in the root cause.
R_EEL_ERR_INTERNAL <*2>	error	meaning	A library internal problem occurred, that cannot be related to a concrete root cause. The library will be locked.
		reason	Some library internal checks that should never fail, failed. These can be checks on hardware or software values.
		remedy	Stop emulation and investigate the root cause. In some cases, e.g. in case of external influence on the EEL or FDL variables or the Flash hardware (e.g. caused by wild running application pointers or PC), re-initialization of the FDL and EEL may help. A reasonable proceeding might be: During development: Stop the emulation and investigate in the root cause. In the field: 2~3 times try to re-initialize the library (Or reset the device). If the problem still exists, stop the emulation and investigate in the root cause.

The command status can be updated by the command-initiating function `R_EEL_Execute` as well as by the handler function `R_EEL_Handler`. The notes describe, which status value can be set by which function:

<*1>: Value can be set by `R_EEL_Execute` only.

<*2>: Value can be set by both functions.

No note: Value can be set be `R_EEL_Handler` only.

4.5.9 R_EEL_CMD_CLEANUP — EEL Pool Clean-up

During normal EEPROM emulation, the EEL pool will be filled with DSs, containing the last DS instance and so, important data. Furthermore, many elder DS instances with no longer relevant data will exist. The Clean-up command takes care for deleting the elder DS instances from the ring buffer and thereby temporarily increases the available free EEL pool space for new DS instances.

Note:
The clean-up command will just forward library internal pointers and end immediately. Triggered by that, the supervision process—running when the EEL is idle—will trigger new refresh and prepare processes until the complete ring buffer is updated once.

Differing from the read/write related commands; this operation is bound to slightly different conditions:

- Clean-up is a singular command. No other user or background operation may be executed. This is ensured after the start-up processing, when the operation status is `R_EEL_IDLE` (see Section 3.6, "Driver Status").
- The request status will be updated by `R_EEL_Execute` to the final value. Although further handling of the command itself by the handler is not required, this command has the impact on the system, that it will be busy for a longer time with updating the pool. So, the operational status will change to `R_EEL_BUSY`.

Possible use cases are:

- Remove temporary secret data from the emulation in order to avoid attacks to that data from outside.
- Prepare as much as possible free space in the pool for high frequent data write without the time to prepare new space in between.

Table 17: Configuration of the request structure for the Clean-up command

Request structure element	Data	Comment
command_enu	R_EEL_CMD_CLEANUP	Clean-up command.
address_pu08	-	
identifier_u16	-	
offset_u16	-	
length_u16	-	
status_enu	-	This is an output member. It contains the status of the command during and after the execution. Possible values are described in the next table.

Table 18: Possible statuses of Clean-up command returned to status_enu

Status	Class	Background and handling	
R_EEL_OK	normal	meaning	Command finished successfully.
		reason	-
		remedy	-
R_EEL_ERR_ACCESS_LOCKED	error	meaning	Execution of the requested operation is rejected.
		reason	The library internal status does not allow execution of the command. • The operation status is not `R_EEL_OPERATION_IDLE`. • No data instance stored to the EEL pool (e.g. immediately after Format command execution)

		remedy	Stop the emulation and investigate in the root cause. • Fix EEL handling flow • Store data to the EEL pool (Write command) before execution of Clean-up

Chapter 5 Library Setup and Usage

This chapter describes the library set-up and usage in the final application. Beside the API, it is important to understand the library modules and dependencies, build conditions and constraints as well as timing conditions.

5.1 File Structure

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

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

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

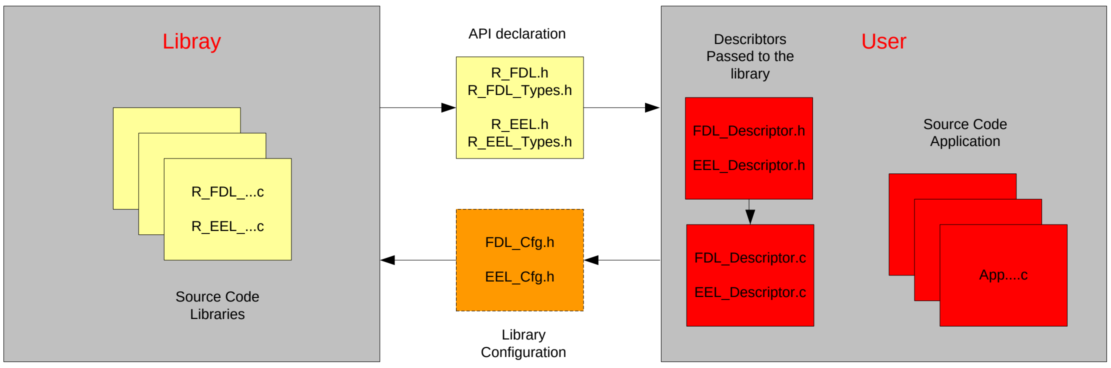
Following that, the sample project can be built in a command line interface and the resulting elf file can be run in the debugger.

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.

5.1.1 Overview

The following picture contains the library and application related files:



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

🟧 Library pre-compile configuration – File name fix, File content user configurable

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

Figure 20: Library and application file structure

The library code consists of different source files, starting with R_FDL/R_EEL_...The files shall not be touched by the user.

The file R_FDL/R_EEL.h is the library interface functions header file. The interface parameters and types are defined in the file R_FDL/R_EEL_Types.h.

The library must be configured for compilation. The files FDL/EEL_Cfg.h contain defines for that. As these are included by the library source files, the files 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 variables can be built up which need to be passed with the functions R_FDL/R_EEL_Init to the FDL/EEL for run-time configuration (see Section 4.2, “Runtime Configuration”).

The structure of the descriptor is passed to the user application by R_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.

5.1.2 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)

Table 19: Installed directory structure

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
ReadMe.txt	Containing link to retrieve the make source code
Setup.exe	Make setup file

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.h	EEPROM emulation sample code
eelapp_control.c	EEPROM emulation sample code
target.h	target device and application related definitions
r_typedefs.h	Type definitions for standard types. May be removed if the types are already defined in the application or by the compiler
fdl_cfg.h	Header file with definitions for library setup at compile time

root\<device name>\<compiler>\sample	
eel_cfg.h	Header file with definitions for library setup at compile time.
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 defines of FDL_Descriptor.h. Should not be edited by the user.
fdl_user.h	Header file for library related application functions, which may be edited by the user
fdl_user.c	Library related application functions, which may be edited by the user
eel_descriptor.h	Descriptor file header with the run-time EEL configuration. To be edited by the user.
eel_descriptor.c	Descriptor file with the run-time EEL configuration.
device header files	GHS: df<device number>_0.h df<device number>_irq.h io_macros_v2.h IAR: ior_7f< device number>.h REC: iodefine.h
start-up files	GHS: dr<dev. num.>_start-up.850 IAR: cstart-up.s85 REC cstart.asm vecttbl.asm
linker directive file	GHS: dr<dev. num.>.ld IAR: lnkr7f<dev. num.>.icf layout.icf REC: dr<dev. num.>.dir

root\<device name>\<compiler>\sample\FDL	
r_fdl.h	Header file containing function prototypes of the library user interface.

root\<device name>\<compiler>\sample\FDL	
r_fdl_types.h	Header file containing calling structures and error enumerations of the library user interface.
r_fdl_mem_map.h	Header file containing memory segment mapping directives

root\<device name>\<compiler>\sample\FDL\lib	
r_fdl_env.h	Library internal defines for accessing the Flash programming hardware and Data Flash related definitions.
r_fdl_global.h	Library internal defines, function prototypes and variables.
r_fdl_hw_access.c	Source code for the library HW interface.
r_fdl_user_if.c	Source code for the library user interface and service functions.

root\<device name>\<compiler>\sample\EEL	
r_eel.h	Header file containing all function prototypes of the library user interface.
r_eel_types.h	Header file containing calling structures and error enumerations of the library user interface.
r_eel_mem_map.h	Header file containing memory segment mapping directives

root\<device name>\<compiler>\sample\EEL\lib	
r_eel_global.h	Library internal defines, function prototypes and variables
r_eel_basic_fct.c	EEL internal functions & state machine
r_eel_user_if.c	EEL user interface functions

5.2 Library Resources

5.2.1 Linker Sections

The following sections are EEPROM emulation library related:

- R_FDL_TEXT

FDL code section, containing the hardware interface and user interface

- R_FDL_CONST

FDL data section, containing library internal constant data

- R_FDL_DATA

FDL Data section containing all FDL internal variables

- R_EEL_TEXT

EEL code section containing the state machine, user interface and FAL interface

- R_EEL_CONST

EEL data section, containing library internal constant data

- R_EEL_DATA

EEL Data section containing all EEL internal variables

5.2.2 Stack and Data Buffer

The EEL utilizes the same stack as specified in the user application. It is the developer's duty to reserve enough stack for the operation of user application, EEL and FDL. With source code library it is not possible to give an exact value for stack consumption. Furthermore, the library usesfuntion pointers, making a static analysis of the stack consumption difficult. Thus, it is recommended to evaluate the stack consumption dynamically in the user application.

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

Note:

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

5.3 Library Timings

In the following important aspects of the EEL timings are presented. Please note that the concrete function and command execution times can vary and are subject to many factors of the target system like compiler setup and current EEL pool status. Especially, it needs to be considered whether the library is in start-up phase or has already been started up.

In any case, three important times need to be considered when utilizing the EEL into a user application:

- **Operation invocation latency:**
  This is the time from calling `R_EEL_Execute` to issue the command and start an operation (e.g. read, write, etc.) up to the point where the process of the operation is really started.
  This latency is determined by execution of higher priority operations but also by the delay to suspend a lower priority operation.
  Some process steps of lower priority operations cannot be suspended because they started Flash write operations (erase can immediately be suspended).
  The 1st steps of the DS Write process until the user data is written cannot be suspended for higher priority Flash write operations because then the data consistency would be endangered.
  So, these process steps must be finished and by this determine the invocation latency of a higher priority operation

- **Handler execution time:**
  The `R_EEL_Handler` execution time should be typically below 100us on a 100MHz device in order to realize a user system with reliable timing. During normal operation this can be reached, but in the start-up phase the execution times will be longer as complex calculations and searches are executed.
  In the start-up phase this time is affected by many conditions and so can only be measured for a reference system, whereas the real timing needs to be evaluated by the customer in the user application.
  Issues affecting this time are e.g. DS Size, higher priority operations on-going, pool size, etc.

- **Overall command execution time:**
  This is the time to execute a complete command, like user DS write, user DS Read from invocation to its completion.
  This time is affected by many conditions and so can only be measured for a reference system,

whereas the real timing needs to be evaluated by the customer in the user application. Issues affecting this time are e.g. Flash Write time (in the evaluations also the worst case time need to be considered), DS Size, operation invocation latency, higher priority operations on-going, etc. Hence, this time is not mentioned again in the upcoming sections.

5.3.1 Library Timings during Start-up

The library needs to execute various process steps according to the implementation concept (see start-up phase description). The `R_EEL_Handler` execution time during these steps will be partially more than 100µs, which needs to be considered in the library implementation concept.

Note:
From implementation point of view, the start-up phase will end when the operational status changes from `R_EEL_OPERATION_STARTUP` to `R_EEL_OPERATION_BUSY/IDLE`. Then all start-up operations are finished.
From timing point of view, the start-up phase will end when the access status changes from `R_EEL_ACCESS_READ_WRITE` to `R_EEL_ACCESS_UNLOCKED`. The remaining start-up operations are executed in background and transparent for the user.

5.3.1.1 Early Read Command

A read command executed during the library start-up phase while the RAM table is not (completely) filled is called early read.

Operation Invocation Latency

The maximum latency of the read operation invocation by the `R_EEL_Execute` function is defined by the `R_EEL_Handler` execution time.

Handler Execution Time

The data of a DS with a certain ID to be read is found as follows:

- If the ID-X RAM table entry belonging to the ID is already filled, the entry addresses the data and the data can be read quickly.
- If the ID-X RAM table entry belonging to the ID is not yet filled, the DS is searched by parsing the REF entries from the youngest one backwards until a valid DS with the ID is found.

According to the possibly necessary REF entry parsing, the early read may take a longer time (more than 100us) and requires 100% CPU load. Furthermore, the data read from Flash itself needs some time. So, the required time can already exceed 100µs for DS sizes > 32 Bytes.

5.3.1.2 Early Immediate Write Command and related

The following commands are closely related to an early immediate write and can be treated equally from a timing perspective:

- Early immediate write
- Early immediate invalidate
- Early immediate incremental write

The early write sequence does not differ to the normal write. Generally, a write operation needs to wait for the end of a preceding immediate operation. Trying to invoke an immediate operation before will be rejected.

Operation Invocation Latency

The maximum latency of the write operation invocation by the `R_EEL_Execute` function is defined by the `R_EEL_Handler` execution time.

Furthermore, after invocation, starting of the write/invalidation process need to wait for

- The end of a higher priority read command.
- The end of blocking by a lower priority DS Write process invoked by user DS Write/Invalidation command or background Refresh process. In order to ensure data and ring buffer consistency, any DS Write process need to block higher priority Write commands until the process step to write the user data is reached.
  The blocking time is defined by 5 times a 1-word Data Flash Write (write SOR and RWP; possibly activate a new block by writing A0, RWP and A1.

Handler Execution Time

The execution of `R_EEL_Handler` time should be <100us on a 100MHz device.

5.3.1.3 Early Write Command and related

The following commands are closely related to an early write and can be treated equally from a timing perspective:

- Early write
- Early invalidate
- Early incremental write

The early write sequence does not differ to the normal write. Generally, a write operation needs to wait for the end of a preceding write or invalidation operation. Trying to invoke a write before will be rejected.

Operation invocation latency

The maximum latency of the write operation invocation by the `R_EEL_Execute` function is defined by the `R_EEL_Handler` execution time.

Furthermore, after invocation, starting of the write/invalidation process need to wait for

- The end of a higher priority read, immediate write or immediate invalidation command.
- The end of blocking by a lower priority DS Write process invoked by user DS Write/Invalidation command or background Refresh process. In order to ensure data and ring buffer consistency, any DS Write process need to block higher priority Write commands until the process step to write the user data is reached.
  The blocking time is defined by 5 times a 1-word Data Flash Write (write SOR and RWP; possibly activate a new block by writing A0, RWP and A1.

Handler execution time

The execution time should be <100us on a 100MHz device.

5.3.2 Library Timings during normal Operation

If not mentioned otherwise, in the normal operation phase the execution time of the `R_EEL_Handler` function should always be below 100us on a 100MHz device.

An on-going Flash erase will not block any user command. The erase will be suspended and later on resumed. Anyhow, after a configurable number of times suspending, the warning `EEL_ERR_ERASESUSPEND_OVERFLOW` is returned in order to inform the user to give sufficient time to complete the erase operation rather than extremely frequently invoking read/write/invalidation operations.

5.3.2.1 Read Command

Operation Invocation Latency

The maximum latency of the Read operation invocation by the `R_EEL_Execute` function is dominated by the `R_EEL_Handler` execution time. However, the data read from Flash itself also needs some time. So, the time can exceed 100µs for DS sizes > 32 Bytes.

Handler Execution Time

Typically the handler execution time will be below 100us.

5.3.2.2 Immediate Write Command and related

The sequences of immediate write, immediate invalidate and immediate incremental write commands during normal operation do not differ from the early commands during start-up. So please refer to Section 5.3.1.2.

5.3.2.3 Write Command and related

The sequences of write, invalidate and incremental write commands during normal operation do not differ from the early commands during start-up. So please refer to Section 5.3.1.3.

5.3.2.4 Format Command

The Format command is considered as an exclusive command and can only be executed if the background state machine is `R_EEL_OPERATION_IDLE` or `R_EEL_OPERATION_PASSIVE`. So, invocation by `R_EEL_Execute` is rejected until this state is reached.

Operation Invocation Latency

The operation is invoked without latency as no other operations are ongoing.

Handler Execution Time

The execution of `R_EEL_Handler` time should be <100us on a 100MHz device.

5.3.2.5 Clean-up Command

The Clean-up command is considered as an exclusive command and can only be executed if the background state machine is `R_EEL_OPERATION_IDLE`. So, invocation by `R_EEL_Execute` is rejected until this state is reached.

Operation Invocation Latency

The operation is invoked without latency as no other operations are ongoing.

Handler Execution Time

The Clean-up command only sets an internal variable to more often call the refresh and prepare processes in background. The execution of `R_EEL_Handler` time will be <100us on a 100MHz device.

5.4 Library Setup and Integration

There are several ways to approach the EEPROM emulation concept and the integration into the user application.

It is for sure important to understand the basics of the RENESAS EEPROM emulation concept and the library architecture, design and implementation before starting the actual integration of the library into the target application. Such integration requires careful consideration of the libraries features and requirements as well as the user application requirements.

A few things worth mentioning in this context are listed in the following:

- Start-up time until 1st data read and write
- CPU load by the EEL, during library start-up and during normal operation
- Where to call the EEL_Handler function
- Where to call the EEL_Execute function
- How to map application variables to the EEL IDs
- ...

Several importance aspects are collected in the following sections. Please go through them carefully before setting up the library configuration for your own target application.

5.4.1 EEL Pool Configuration

As mentioned in Section 2.2, “Data Flash Pool Structure”, the available Data Flash space is separated in different pools. The part used by the EEL is the EEL pool. The pools are configured by the FDL descriptor (See Section 4.2.1, “FDL Runtime Configuration Parameters”).

The EEL organizes the pool as a set of virtual blocks of equal size and uses this pool as a kind of ring buffer (See Section 4.2.2, “EEL Runtime Configuration Parameters”). The virtual block size is configurable by the EEL descriptor in terms of number of physical blocks. The major conditions for configuration are:

- EEL pool size = “number of virtual blocks” * “size of virtual blocks
- Minimum number of virtual blocks = 4
- Minimum virtual block size = Maximum DS size + REF entry size + block header size + 4Bytes
  However, from practical usage point of view, the virtual block size ought to be configured bigger (See Section 5.4.3, “EEL Data Set Configuration”)

5.4.2 Endurance Calculation

The Data Flash has a limited endurance (number of erase cycles), which is furthermore dependant of the expected data retention of the physical data. As the library organizes data storage in a kind of ring buffer, the usage of the Flash blocks is balanced in a way, that all Flash blocks of the EEL pool will have a similar number of erase cycles. This avoids “hot spots” with blocks of high erase cycles.

By storing multiple DS instances in the ring buffer, the EEL architecture allows storing certain DSs much more often than the physical Flash endurance would allow. This, requiring that other DSs are stored less often.

It is very important to judge if the EEL pool size and Data Flash endurance are sufficient to store the available user data the required number of times. At the final end, this need to be proven by the customer as much conditions influence the EEL, such as fragmentation of user data, storage sequence…
Renesas provides a tool for a first estimation of the Data Flash endurance, based on a chosen EEL and user data configuration. This tool is called “Endurance calculation sheet”. It is individual for each EEL type. If not available in the installer package of download page, please request it from the Flash support.

Note:
The endurance calculation sheet is a very helpful tool, but still the result is just an estimation and cannot be absolute accurate because the result depends on different conditions like e.g. the sequence of the written Data sets. So, the result must be confirmed in the real user application

5.4.3 EEL Data Set Configuration

Important consideration is the data fragmentation. For each application there is a need to find a balance between small data granularity, resulting in small DSs with different IDs or few big DSs.

- Each DS instance requires a REF entry. This is additional data management overhead to be stored
    - Decreasing the Data Flash endurance available for the user data
    - Decreasing the data storage and read performance
- Big DSs may often not fit into the remaining blank space of a current active virtual block, resulting in activation of the next block for storage, the remaining space in the current block is lost for further data storage in the current ring buffer loop.
    - Faster user data storage and read due to less data management overhead
    - More unused Data Flash space in full blocks

Important tool to find the right balance with respect to Data Flash usage is the “Endurance calculation sheet” (See last Section). However, some basic hints may help to find the correct configuration.

- Separate the data which is written at different times in the application lifecycle. This ensures that only the data is written which has really been updated. E.g.:
    - Store ODO meter data (Updated very often) separated from any other data to avoid that the other data is continuously re-written without being updated
    - Merge configuration data (only written few times) to few DSs with reasonable size
- Use the incremental write commands to store user data only, when it has really changed
- Don’t configure often written DSs too big in order to avoid unused space in the virtual blocks.
    - E.g.:
        - DS has max. possible size:
          In average on each storage ½ size of the current virtual block size is lost
        - DS has ½ of max. possible size:
          In average on each storage ¼ size of the current virtual block size is lost
    - Thus, it is recommended to choose the max. DS size less than ½ of max possible size

5.4.4 Distributing Data between FDL and EEL

The EEL is designed as a standard solution for storage of dynamic data to the Data Flash. It provides certain features and is bound to conditions and limitations of the library concept.

In case of very specific requirements that cannot be fulfilled by the library or purely static data, it is possible to build a solution for data storage by using the FDL only. This solution can be operated together with the EEL, however, is bound to certain conditions, such as:

- The FDL based solution may have no relation to the EEL. Especially, it may not impact the EEL operation, e.g. by modification of resources used by the EEL (RAM sections, Flash hardware, …)
- FDL access and EEL access by the user application must be synchronized as the EEL always assumes to be the only FDL usage master
    - EEL must be in non-busy operational status (Not R_EEL_OPERATION_STARTUP, R_EEL_OPERATION_BUSY) when FDL operation is started by the user application
    - The user application triggered FDL operation must be finished when the EEL is handled (Any EEL function is called)
    - Re-entrancy and function call nesting of any EEL or FDL function is forbidden at all (See Section 6.1, “Function re-entrancy” and 6.2, “Task switches, context changes and synchronization between EEL functions”)

5.4.5 R_EEL_Handler Calls

The handler function R_EEL_Handler the complete EEL state machine. Once an operation is initiated, its status need to be controlled and the the states need to be forwarded to complete the operation. This is done by regularly calling the handler function from the application.

In order to achieve an appropriate operation performance, the function need to be called frequently. The calling style depends on the user application architecture. Major possible solutions are:

- Asynchronous to EEL operation invocation by EEL_Execute in an operating system idle task

  In a normal system the CPU load is balanced in a way, that a sufficient idle time is available.
  By calling from the idle task loop, the handler can be called frequently and the EEPROM Emulation performance is quite high. However, as the idle time is not always deterministic, also the emulation performance might not be deterministic enough.

  - Advantages:
    + Usually high emulation performance
    + No blocking of other user application operations
  - Disadvantages:
    - Not always deterministic

- Asynchronous to EEL operation invocation by EEL_Execute in a timed task

  By calling in a timed task a deterministic performance can be reached. However, as the Flash operations execution (Flash Write) usually require less than 500us, for best possible performance the handler should be called in very short time slices. As these are usually not available, the performance of the emulation decreases.

  - Advantages:
    + Deterministic
  - Disadvantages:
    - Lower emulation performance

- Synchronous with EEL operation invocation by EEL_Execute

  The handler is called in the same function context as EEL_Execute. The handler call is repeated in this function in a loop until the EEL operation has finished.

  - Advantages:
    + Highest performance
  - Disadvantages:
    - function execution time is high and not deterministic

- In the Flash interrupt context

  The handler is called in the Flash interrupt context. In that case the function is not polled, but exactly called when required to finish a user or background operation. Therefore, the library must be configured to support the Flash interrupt (See 5.1, “Pre-compile configuration”)

  - Advantages:
    + Highest performance
  - Disadvantages:
    - function execution time in interrupt context
    - Synchronization between handler call in interrupt context and other EEL function in tasks
    - handler must be called in other timed tasks to do bit error checks
    (See 3.4, “Background Operations”)

5.4.6 Reset Robustness 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 is considered in the library concept and tested by various interruption, reset and power fail scenarios including logical as well as stress tests with random interruptions. Nevertheless, it is up to the user of the library to test and ensure (reset) robustness of the EEL in the user application context by appropriate test scenarios.

5.4.7 ECC Errors

The Renesas Flash technology is very robust and allows reading stable data values over a very long data retention period.

Additionally, the Data Flash is protected by Error Correction Code (ECC). This means that together with 32bit of data, additional bits are stored. A dedicated decryption logic allows to at run-time - without any delay visible to the user application – to detect and correct one wrong bit out of the data and ECC bits. 2 failing bits are detected but cannot be corrected. Reaction on multi-bit errors (>2bits) is not defined. Signalling of SED or DED is possible but not ensured.

The signalling of single bit ECC errors (SEC) or double bit ECC errors (DED) can happen on different ways. On the one hand it is possible to enable SEC and DED interrupts, on the other hand it is possible to read error results without activation of interrupts. Last option is also supported by the underlying FDL, which reads the data and returns an SED or DED error in the request structure (“Error polling”).

The ECC implementation is a very efficient mechanism to ensure stable data read values even though possibly single data bits might get “weak” or flip e.g. due to exceeding the specified data retention period by far.

The EEL basically uses the FDL read operation with “Error polling” in order to read EEL management data, where it can be expected that ECC errors are present, e.g. caused by interrupted Erase/Write operations to the Flash. The data and ECC errors are evaluated and so, stable user data is ensured even in case of EEL operation interruption.

Additionally, the EEL uses the read mechanism to scan the Flash for SED errors in order to refresh the Flash contents (See Section 3.4, “Background Operations”).

Differing from that, ECC errors in the user data are not expected, as stable conditions of the user data are ensured by the EEL. So, and in order to improve the EEL read performance, the user data read operations are executed directly by the CPU. Following that, in the unreasonable case of an ECC error, it is possible that ECC error interrupts are issued. It is up to the user application to react appropriately on such interrupts.

DED: User data is defect. The application should withdraw the data

SED: Data ought to be correct, but should be refreshed in a reasonable short time. It is recommended to frequently call the R_EEL_Handler (will check 4Bytes of the ring buffer on each call, when the EEL is idle), so that the error is also detected by the EEL and automatically corrected.

5.4.8 Relation between operation status and driver status

Important operation control signals are the status of a requested user operation and the driver status. The following diagram shows the relationship between a requested operation and execution of background operations.

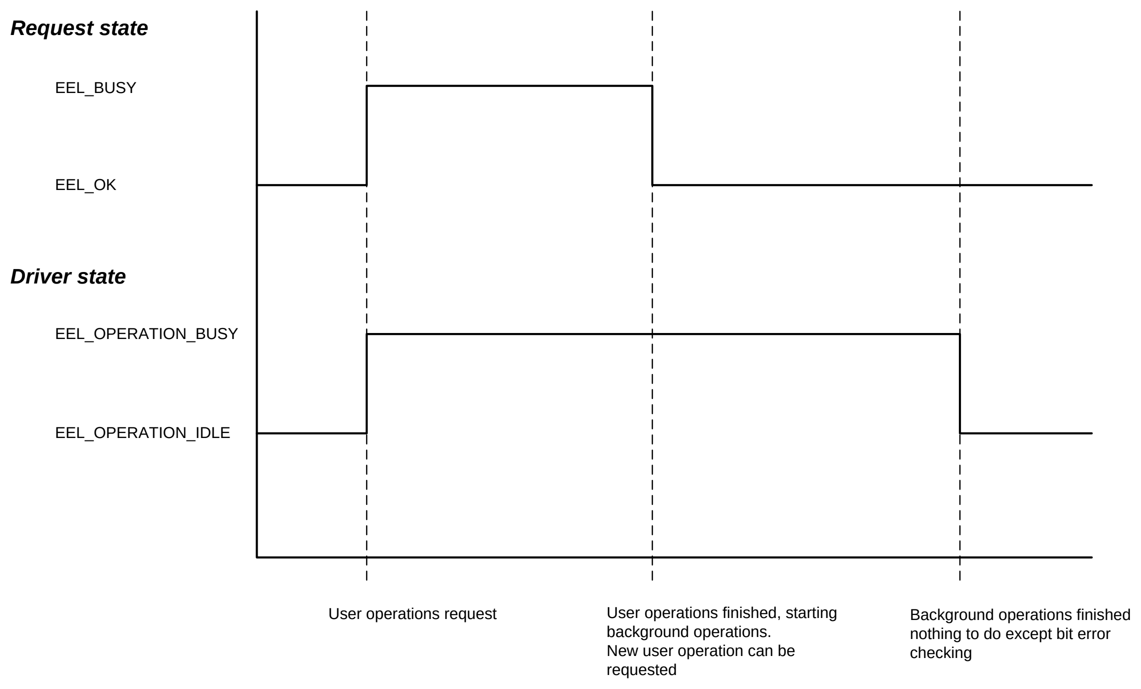

Request state
EEL_BUSY
EEL_OK
Driver state
EEL_OPERATION_BUSY
EEL_OPERATION_IDLE
User operations request
User operations finished, starting background operations. New user operation can be requested
Background operations finished nothing to do except bit error checking

Figure 21: Relation between operation status and driver status

The handler call frequency significantly determines the EEL performance. As long as the driver or request state is busy, the handler should be called with higher frequency. When the driver state is idle, the call frequency can be reduced as then only cyclical bit error checks are done by the EEL. Then, on each handler call one Flash word is checked.

5.5 Sample Application

Setting up a proper EEL configuration requires some hands on experience with the EEL. The best way after initially reading the user manual will be testing the EEL application sample. The application sample is shipped with the library installer and copied to a separate directory during installation (see Section 5.1, "File Structure").

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.

In order to get a better feeling for the source code files, the request structure mechanism and the library start-up behavior, the user should first execute and debug the existing sample. Later on, the sample might be extended by further IDs and different data read and write sequences in order to come nearer to the later application requirements (data set amount and size) and to get a feeling of the CPU load and execution time during start-up and normal operation. After this exercise, it might be easier to understand and follow the recommendations and considerations of this document.

The flow chart in Figure 22 represents the recommended EEL life cycle during device operation including the API functions to be used. It can be separated into three phases:

- In the start-up phase, the EEL is initialized by `R_EEL_Init` and the background operation is started by `R_EEL_Startup`.
- During normal operation, the foreground operations (user commands) are initiated synchronous to the application, while the background handler task ought to be executed in a task, asynchronous to the application (idle task, interrupt task, timed task).
- In the power down phase, the EEL is shut down. `EEL_Handler` needs to be executed until the library status is passive. This is required in order to complete ongoing EEL processes.

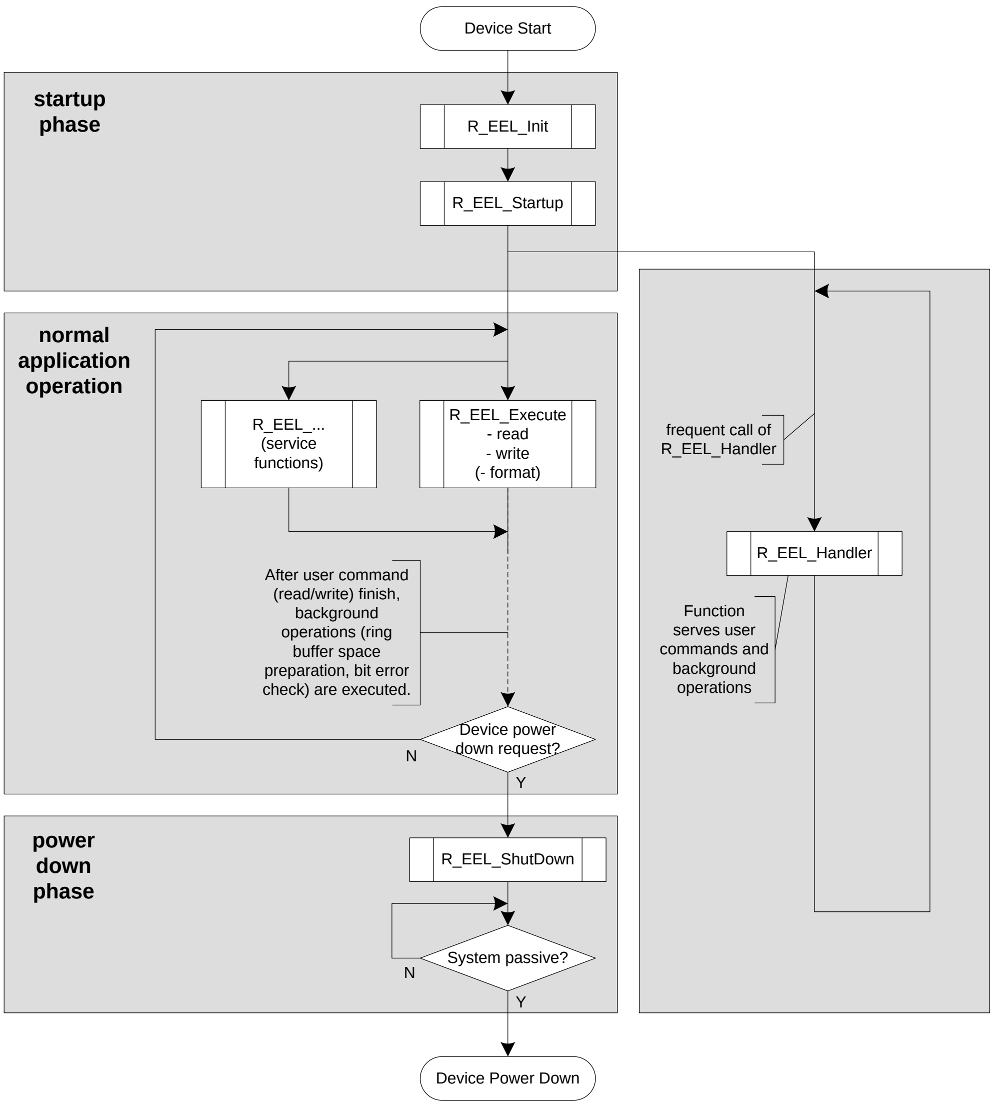

Device Start
startup phase
R_EEL_Init
R_EEL_Startup
normal application operation
R_EEL_... (service functions)
R_EEL_Execute - read - write (- format)
After user command (read/write) finish, background operations (ring buffer space preparation, bit error check) are executed.
Device power down request?
N
Y
power down phase
R_EEL_ShutDown
System passive?
N
Y
Device Power Down
frequent call of R_EEL_Handler
R_EEL_Handler
Function serves user commands and background operations

Figure 22: EEL life cycle

Device Start-up

The device boots and the application starts up. Usually, very soon some data sets need to be read. Then the EEPROM emulation has some time to come up completely before the rest of the data needs to be read (e.g. build up a RAM mirror) and written.

The example code below reads and writes data as soon as possible and then waits until the EEL is fully operational and unlocked.

```
uint08_t              buffer_au08[0x100];
r_eel_request_t       myRequest_str;
r_eel_driver_status_t driverStatus_str;
r_eel_status_t        ret;

/* ----------------------------------------------------------------------- *\
   Initialize the EEL
   - eel_RTConfiguration_str should have been set in EEL_Descriptor.c
\* ----------------------------------------------------------------------- */
ret = R_EEL_Init (R_EEL_OPERATION_MODE_NORMAL, eel_RTConfiguration_str);
```

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

ret = R_EEL_Startup();
if (R_EEL_OK != ret)
{
       /* Error handler */
}

/* ---------------------------------------------------------------------- *\
   Wait until we can read/write 1st data sets
\* ---------------------------------------------------------------------- */
do
{
    R_EEL_Handler();
    R_EEL_GetDriverStatus (&driverStatus_str);
}
/* Wait until early read/write is possible (or error) */
while (    (R_EEL_OPERATION_STARTUP == driverStatus_str.operationStatus_enu)
       && (R_EEL_ACCESS_LOCKED     == driverStatus_str.accessStatus_enu));

/* Error check */
if (R_EEL_OK != driverStatus_str.errorStatus_enu)
{
       /* Error handler */
}


/* ---------------------------------------------------------------------- *\
   Early read/write operation
\* ---------------------------------------------------------------------- */
myRequest_str.address_pu08    = (&buffer_au08[0]);
myRequest_str.identifier_u16 = 10u;
myRequest_str.length_u16     = 0x10u;
myRequest_str.offset_u16     = 0x13u;
myRequest_str.command_enu    = R_EEL_CMD_READ;

R_EEL_Execute (&myRequest_str);

/* Wait until operation end */
while (R_EEL_BUSY == myRequest_str.status_enu)
{
       R_EEL_Handler();
}

/* Error check */
if (R_EEL_OK != myRequest_str.status_enu)
{
       /* Error handler */
}

myRequest_str.address_pu08    = (&buffer_au08[0]);
myRequest_str.identifier_u16 = 10u;
myRequest_str.command_enu    = R_EEL_CMD_WRITE;

R_EEL_Execute (&myRequest_str);

/* Wait until operation end */
while (R_EEL_BUSY == myRequest_str.status_enu)
{
       R_EEL_Handler();
}

/* Error check */
if (R_EEL_OK != myRequest_str.status_enu)
{
      /* Error handler */
}

/* ---------------------------------------------------------------------- *\
   Wait for fully operational and access unlock
\* ---------------------------------------------------------------------- */
do
{
    R_EEL_Handler();
    R_EEL_GetDriverStatus (&driverStatus_str);
}
/* Wait until the system is completely up and running (or error) */
while (R_EEL_OPERATION_STARTUP == driverStatus_str.operationStatus_enu);

/* Error check */
if (R_EEL_OK != driverStatus_str.backgroundStatus_enu)
{
      /* Error handler */
}

/* ---------------------------------------------------------------------- *\
   Now the EEL is fully operational
\* ---------------------------------------------------------------------- */
```

Normal Operation

When the device has passed the start-up phase and is in normal operation, the complete functionality is available. The example code below reads and writes data sets.

```
r_eel_request_t myRequest_str;

/* ---------------------------------------------------------- *\
   read command example
\* ---------------------------------------------------------- */
myRequest_str.address_pu08   = (uint8_t*)(&buffer_au08);
myRequest_str.identifier_u16 = 10u;
myRequest_str.length_u16     = 0x10u;
myRequest_str.offset_u16     = 0x13u;
myRequest_str.command_enu    = R_EEL_CMD_READ;

R_EEL_Execute( & myRequest_str );

/* Wait until operation end */
while (R_EEL_BUSY == myRequest_str.status_enu)
{
    R_EEL_Handler();
}
/* Error check */
if (R_EEL_OK != myRequest_str.status_enu)
{
    /* Error handler */
    . . .
}

/* ---------------------------------------------------------- *\
   write command example
\* ---------------------------------------------------------- */
myRequest_str.address_pu08   = (uint8_t*)(&buffer_au08);
myRequest_str.identifier_u16 = 10u;
myRequest_str.command_enu    = R_EEL_CMD_WRITE;

R_EEL_Execute( &myRequest_str );
/* Wait until operation end */
while (R_EEL_BUSY == myRequest_str.status_enu)
```

```
{
    R_EEL_Handler();
}
/* Error check */
if (R_EEL_OK != myRequest_str.status_enu)
{
    /* Error handler */
    . . .
}
```

Note:
Please note that the state of the command reported inside `myRequest_str` does not necessarily match the internal state of the EEL driver, which can be obtained by means of the `R_EEL_GetDriverStatus` function: Even if a command has finished, the driver might still be busy with background processes. Therefore it is imperative to continue calling the `R_EEL_Handler` periodically, even if no commands are pending.

Device Power Down

On power down, the user application should give the library time to finish background operations which are under progress. This can be reached by using the service functions in the following way:

```
/* ---------------------------------------------------------------------- *\
   Request Library shutdown
\* ---------------------------------------------------------------------- */
R_EEL_Shutdown();

/* ---------------------------------------------------------------------- *\
   Wait until all background processes are finished and the supervision
   gets passive
\* ---------------------------------------------------------------------- */
do
{
    R_EEL_Handler();
    R_EEL_GetDriverStatus (&driverStatus_str);
}
while (R_EEL_OPERATION_PASSIVE != driverStatus_str.operationStatus_enu);

/* Error check */
if (R_EEL_OK != driverStatus_str. backgroundStatus_enu)
{
       /* Error handler */
}
```

5.6 Miscellaneous

5.6.1 MISRA Compliance

The EEL and FDL have been tested regarding MISRA compliance.

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

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

Chapter 6 Cautions

6.1 Function re-entrancy

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

6.2 Task switches, context changes and synchronization between EEL functions

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

Example of not allowed sequence:

- Task1: Start an EEL operation with EEL_Execute
- Interrupt the function execution and switch to task 2, executing EEL_Handler function
- Return to task 1 and finish EEL_Execute function

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

Note:
This limitation is valid also for FDL functions as well as for a mixture of EEL and FDL functions

6.3 EEL performance

The performance of the EEL operations strongly depends on the frequency of the handler calls. This especially affects operations which require many Flash write operations until the operation is finished, such as DS Write and background operations such as Start-up processing or Refresh.

As the typical Flash write operation needs less than 1ms, a lower handler call frequency significantly reduces the operation performance.

See Section 5.4.5, “R_EEL_Handler Calls”

6.4 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 (User Pool) and one part is reserved for EEPROM emulation by the Renesas EEL (EEL Pool). The FDL is prepared to split the Data Flash into an EEL Pool and a User Pool.

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

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

See Section 4.2.1, “FDL Runtime Configuration Parameters”.

6.4.1 User Data Flash access during active EEPROM emulation

While the EEL is active, any direct Data Flash access like Data Flash Read by the CPU or execution of FDL functions are not allowed at all!

The EEL can at each time erase or write Data Flash. During these operations Data Flash is not accessible for Read operations, even not on other address ranges. Furthermore, execution of FDL operations like Flash Erase or Write would be blocked.

Following that, EEL operations and user accesses to Data Flash must be synchronized. This has to be done by the application, considering the EEL as the default master. If the user application wants to get access rights, the EEL need to be suspended beforehand. The API contains the functions `R_EEL_SuspendRequest` and `R_EEL_ResumeRequest` for this. See Section 3.9, “Suspend / Resume”.

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

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

6.5 Entering power safe mode

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

In order to gain a proper synchronisation between EEL and Power safe mode entering, the library operations must be idle or suspended before entering the mode (Please check R_EEL_Suspend API description).

Note:
The user application need to ensure that the integrity of all resources required by EEL and FDL is retained when entering a power save mode. This is required if after leaving the power save mode the libraries shall continue operation without complete re-initialization. This includes:
- The FDL and EEL data in the library RAM section
- IDX table
- Flash programming hardware internal registers and memory

6.6 Library behaviour after operation interruption

Library operation might be suddenly interrupted e.g. by a power fail. Depending on the interrupted operation (E.g. Flash erase, write ....) the behaviour of the library on the next start-up might differ:

- Library was idle or at the end of an operation:

  Normal library start-up

- Flash block erase or Flash block header operation was interrupted:

  Eventually it is necessary that the library fixes a block status (e.g. block activation or block erase was interrupted). In this case additionally Flash write operations might take some more time and so, slightly enlarge the time until the driver leaves the state R_EEL_OPERATION_STARTUP.

  Furthermore, the driver will return the warning EEL_ERR_FIX_DONE as an indication that a fix was done. The library operation continues normal, the application need not react on the warning.

- DS Write was interrupted

  If the DS write proceeded up to writing the EOR0, the DS is valid. If the EOR1 has not been written, the DS will automatically be refreshed.

  In this case additionally Flash write operations might take some more time and so, enlarge the time until the driver leaves the state R_EEL_OPERATION_STARTUP.

- DS Write was interrupted

  If the DS write did not proceed up to writing the EOR0, the DS is invalid. The start-up process will recognize the block as full and continue operation in the next block. This DS instance is not considered on DS read.

6.7 Application update issues

When a user application shall be updated but the EEPROM emulation data shall be used also further on, different constraints need to be considered with respect to the ROM ID-L table.

6.7.1 Change DS length

On application update it might be required to change the DS length of some IDs. This is automatically done, when the ID-L table in ROM is updated. After that update all DS's are read/written with the new length and also the Refresh process copies the data with the new length:

- Old length < new length

  Data is extended by any data stored after the DS (data of the next DS = undefined)

  Note:
  When the read data with the new bigger length overlaps into a blank Flash range, this cannot be identified by the EEL. So, random data may be read and furthermore, the read attempt will result in an ECC error (See Section 5.4.7, "ECC Errors"). Depending on the ECC error notification configuration of the device, an error notification by exception/interrupt might be issued by the device ECC decoder logic. Thus, the user application shall refrain from reading more data than it was available before length change.

- Old length > new length

  Data is cut to new length

The DS length is not stored within the DS, but in the ID-L table. When the table is updated, the information of the former DS size get lost. So, the library provides no measure to get the length of the last stored DS instance. This information must be provided otherwise if required

Possible options to store the DS lengths are:

- Store the length of the DS in the DS itself

  If the length is stored in the 1st Bytes, a read operation on the 1st Bytes only can be done.

- Reserve a special DS only containing all available DS IDs and the length information

- Protect the DSs with a checksum

  Calculating the checksum from data with a different length result in a checksum differing from the stored one (not 100% safe!).

6.7.2 ID-L ROM table not available

Scenario: The boot loader as well as the application needs to access EEPROM emulation data with Read as well as Write. While the application requires frequent data write, the boot loader will only store a very limited amount of data, e.g. to store the application update process state.

The ROM ID-L table containing all IDs available in the emulation belongs to the application. On application update it needs to be removed together with the application. After removal of the ID-L table, normal operation of the EEPROM emulation cannot continue. In order to continue at least with limited functionality, the library provides a limited operation modes to survive this situation.

The mode configuration is done by the initialization function `R_EEL_Init`. In order to change the mode, `R_EEL_Init` need to be called again.

- `EEL_OPERATION_MODE_NORMAL`

  Full (normal) operation of the library, requires the complete ID-L table in ROM

- `EEL_OPERATION_MODE_LIMITED`

  Operation with limited ID-L-table in ROM, containing only the IDs, required by the boot loader. The DS Read and Write work on the ID-L table, so there is no issue with them. However, the Refresh process need to copy all needed DS instances in the affected block.
  The length information of DS instances in the block whose IDs are not listed in the ID-L table are not available. To copy them despite of the lacking information, special mechanisms are implemented. These mechanisms are not that performant and thus, the Refresh will consume more time as operation and also the function execution time is longer.

  Note:
  Run only in limited mode when necessary. It is not recommended to use this mode during normal operation

6.8 Changing EEL pool size and location configuration

Once selected, the EEL pool size, location as well as virtual block size cannot be changed without losing all user data. Such change will require formatting of the EEL pool. If not done, the library reaction is undefined, it might return various errors or even lock.

6.9 Precompile options

The user must not use any pre-compile configuration options that are not documented in present manual. The library behaviour might be undefined

Revision History

Chapter/Section	Page	Description
		V2.00: Initial version (V2.00 in order to align the major version number to the FDL version (currently V2.00))
 all 4.2.2 4.5.9	 43 80	V2.01: - Minor wording fixes (e.g. R_EEL_OPERATION_START-UP → R_EEL_OPERATION_STARTUP - Fixed description of EEL_CONFIG_ERASE_SUSPEND_THRESHOLD min value - Added cause/remedy for Clean-up error

EEPROM Emulation Library

R01US0116ED0201

Renesas Electronics Corporation