User Manual

16

# EEPROM Emulation Library

## Type T02 (Tiny), European Release

16 Bit Single-chip Microcontroller
RL78 Family

Installer: RENESAS_EEL_RL78_T02E_Vx.xxx



Renesas Electronics
www.renesas.com

R01US0070ED0103
23.06.2014

# Notice

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

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

   **“Standard”:** Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   **“High Quality”:** Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti- crime systems; safety equipment; and medical equipment not specifically designed for life support.

   **“Specific”:** Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.

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

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

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

# Regional information

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

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

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

Visit

http://www.renesas.com

to get in contact with your regional representatives and distributors.

# Preface

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

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

**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) | $2^{10} = 1024$ |
| M (mega): | $2^{20} = 1024^2 = 1,048,576$ |
| G (giga): | $2^{30} = 1024^3 = 1,073,741,824$ |

**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 |
|---|---|
| R01US0061EDxxxx | Data Flash Access Library Type T02 (Tiny), European Release |

### (3) List of Abbreviations and Acronyms

| Abbreviation | Full form |
|---|---|
| API | Application Programming Interface |
| Code Flash | Embedded Flash where the application code or constant data is stored. |
| Data Flash | Embedded Flash where mainly the data of the EEPROM emulation are stored. |
| Data Set | Instance of data written to the Flash by the EEPROM Emulation Library (EEL), identified by the Data Set ID |
| DS | Short for Data Set |
| Dual Operation | Dual operation is the capability to access flash memory during reprogramming another flash memory range. Dual operation is available between Code Flash and Data Flash.<br>Between different Code Flash macros dual operation depends on the device implementation. |
| ECC | Error Correction Code |
| EEL | EEPROM Emulation Library |
| EEPROM | Electrically erasable programmable read-only memory |
| EEPROM emulation | In distinction to a real EEPROM the EEPROM emulation uses some portion of the flash memory to emulate the EEPROM behavior. To gain a similar behavior some side parameters have to be taken in account. |
| FAL | Flash Access Library (Flash access layer) |
| FCL | Code Flash Library (Code Flash access layer) |
| FDL | Data Flash Library (Data Flash access layer) |

| Abbreviation | Full form |
|---|---|
| Firmware | The Firmware is a piece of software that is located in a hidden area of the device, handling the interfacing to the flash. |
| Flash | Electrically erasable and programmable nonvolatile memory. The difference to ROM is, that this type of memory can be re-programmed several times. |
| Flash Area | Area of Flash consists of several coherent Flash Blocks |
| Flash Block | A flash block is the smallest erasable unit of the flash memory. |
| Flash Macro | A certain number of Flash blocks is grouped together in a Flash macro. |
| FW | Firmware |
| ID | Identifier of a Data Set instance in the Renesas EEPROM Emulation |
| MF3 | Name of the Flash technology used in RL78 devices. |
| 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. |
| Segment / Section | Segment of Flash is a part of the flash that might consist of several blocks. Important is, that this segment can be protected against manipulation. |
| Self-Programming | Capability to reprogram the embedded flash without external programming tool only via control code running on the microcontroller. |
| Serial programming | The onboard programming mode is used to program the device with an external programmer tool. |
| | |

# Table of Contents

# Chapter 1 Introduction

This user manual describes the internal structure, the functionality and the application programming interface (API) of the Renesas RL78 EEPROM Emulation Library (EEL) Type 02, designed for RL78 flash devices with Data Flash based on the MF3 flash technology.

While many RL78 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 to the developers, encapsulating the detailed flash management with a framework based on emulated variables. The developer can access (read and write) these variables 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 RL78 EEPROM Emulation Library Type 02 (from here on referred to as Tiny EEL) is provided for the Renesas and IAR development environments. In contrast to the also available EEL Type 01 for RL78 devices, the Tiny EEL was especially designed for reduced resource consumption (in terms of Code Flash, Data Flash and RAM) in order to bring EEPROM emulation also to small RL78 devices. As a result, the number of features and the maximum manageable data has been reduced in this library (see Section 1.2 for details).

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_RL78.

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

The Tiny EEL requires the corresponding RL78 Data Flash Access Library (FDL) Type 02 (Tiny FDL) for operation. It can be obtained from the same URL as the Tiny 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.

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

## 1.1 Flash Infrastructure

The flash technology which is utilized in RL78 devices is called MF3. Besides the Code Flash, many devices of the RL78 microcontroller family are equipped with a separate flash area—the Data Flash. This flash area is meant to be used exclusively for data. It cannot be used for instruction execution (code fetching).

### 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, RL78 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 Flash Granularity

The MF3 Data Flash of RL78 devices is separated into blocks of 1024 byte. While erase operations can only be performed on complete blocks, reading and writing of data can be done on a granularity of bytes or byte sequences. Reading from an erased MF3 flash byte will return the value 0xFF. The number of available Data Flash blocks varies between the different RL78 devices. Please refer to the corresponding user manual of your device for detailed information.

## 1.2 Feature Overview

The Tiny EEL offers easy-to-use EEPROM-like access to user-defined variable sets stored in the non-volatile Data Flash memory of RL78 microcontrollers. Aiming at low resource consumption, the Tiny EEL offers a basic function set to designers covering the following features:

- Up to 64 independent variables
- Configurable variable sizes between 1 and 255 byte
- Polling operation mode
- Minimized storage consumption for administrative data (10 bytes per flash block and 2 bytes per variable instance)
- Reset resistance
- Block rotation (balanced data flash usage)
- Applicability in operating systems
- Standby functionality for energy savings (via Tiny FDL)
- Resistance against aborted block erasures (via Tiny FDL)

Besides the Tiny EEL, which is the subject of this documentation, Renesas offers another EEL, the EEPROM Emulation Library Type 01. Compared to this more resource intensive library, the Tiny EEL offers a reduced set of features and functions (for more details on these features, please refer to the EEL Type 01 application note R01AN0707EDxxxx):

- No enforced and timing mode
- Only one EEL flash block is utilized for storing data at a time resulting in a limited number and size of variables
- No background maintenance process (i.e. refreshing of the EEL pool needs to be directly triggered from the user application)
- The variable ID is predefined by the order within the descriptor and cannot be selected arbitrarily
- No automatic checksums on data content

If you are unsure which EEL is the best choice for your project, please contact the Renesas support team at application_support.flash-eu@lm.renesas.com.

# Chapter 2 Architecture

This chapter introduces the basic software architecture of the Tiny EEL and provides the necessary background for application designers to understand and effectively use the Tiny 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.

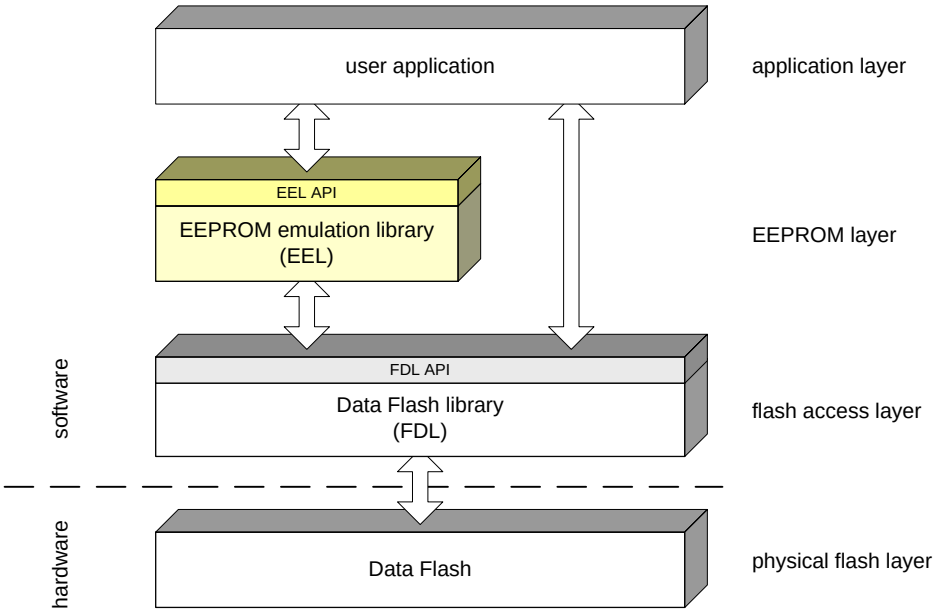


**Figure 1: Layers of the EEPROM emulation system**

The EEL does not directly access the Data Flash. The Data Flash is always accessed via the Data Flash Library (FDL). This way, the designer can use both FDL and EEL simultaneously to access the Data Flash. Of course this does not mean that the multiple accesses are handled in parallel. However, the libraries are particularly designed to not to interfere with each other and are robust against interleaved mixed Data Flash accesses via the opposite library.

A parallel usage of FDL and EEL is strongly encouraged. 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 two individual pools: The FDL pool and the EEL pool.

This separation is also 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 the sanity checks of both libraries ensure that the user application has only access to each pool via the corresponding library as shown in Figure 2. The EEL uses dedicated subroutines of the FDL which are hidden from the user and exclusively reserved for the EEL.

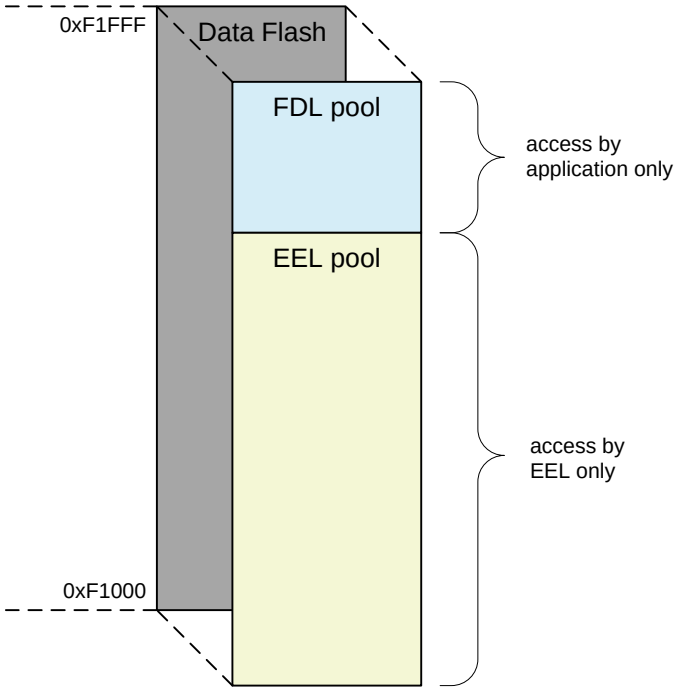


**Figure 2: Logical fragmentation of Data Flash in FDL and EEL pool (example for 4 kB Data Flash)**

The distribution of the available Data Flash to EEL and FDL pool can be configured at compile time in the descriptor of the FDL (see the RL78 FDL T02 user manual for details of the Tiny FDL configuration options). Please note that the assignment of Data Flash memory to FDL and EEL pool can only be done on a flash-block basis. The EEL pool always maps to the Data Flash blocks with the lower addresses in the device memory map.

### 2.2.1 FDL pool

The FDL pool allocates the Data Flash blocks that are directly managed by the user application via the Tiny 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 Direct Data Flash Access through 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 FDL pool. By this measure, the FDL pool can be treated as a simple array of words. To address the array elements (read/write access), word-indexes starting at 0x0000 can be used. The maximum range of the word-index depends on the pool configuration, i.e. the number of flash blocks reserved for the FDL pool.

### 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 not possible.

#### 2.2.2.1 Automated Data Flash Management through Variable Virtualization

While the data stored in the FDL pool needs to be managed completely by the user application, the usage of the EEL pool is greatly simplified by variable virtualization. The user defines a set of variables with individual sizes at compile time. These Variables 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.

## 2.3 EEL Management

While the simple usage of the Tiny 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 variables. Afterwards, the transitions, sequences and processes within the EEL are highlighted briefly.

### 2.3.1 EEL Pool Structure

In the Tiny EEL, the EEL pool is separated into blocks of 1 kB size, which match the physical block separation of the Data Flash. Each block has a status which indicates the current usage of the block. The status of an EEL block is one of the following:

- **Active:** Due to the strong resource limitations applied for the Tiny EEL, only one EEL block is active at a time and stores values for the defined EEL variables (assuming normal operation). During lifetime, the currently active block rotates through the Data Flash blocks assigned to the EEL pool.
- **Invalid:** Invalid blocks do not store any EEL variable values. Blocks are either actively marked as invalid by the library or are invalid due to unstructured content (e.g. in case of an erased block).
- **Excluded:** Whenever functional operations on a block fail too often indicating a damaged Data Flash, the EEL actively excludes the corresponding block and does not use this block for EEPROM emulation anymore.

Figure 3 shows an exemplary pool configuration for a device with 8 kB Data Flash, where three blocks are assigned to the FDL pool and the EEL operates on 5 blocks.

Whenever the active block (block 1 in the example) is full and cannot store additional data (i.e. a write command fails), a new active block is selected in a cyclic manner and the current valid data set is copied to this new active block. This process is referred to as *refresh*. The former active block is invalidated during a refresh, so that there is only one active block at a time. Excluded blocks (like block 4 in the

example) are ignored during this process and not considered as candidates for the selection of the next active block.

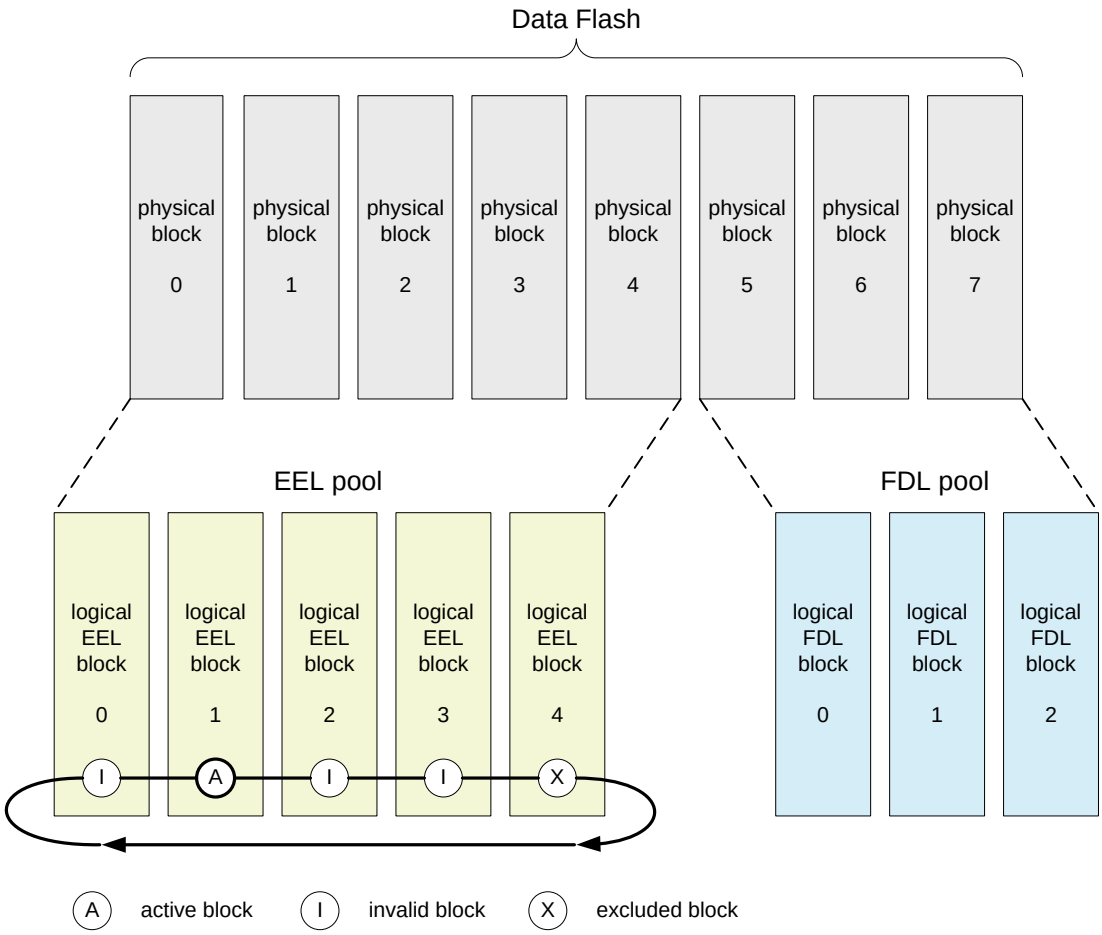


**Figure 3: Exemplary EEL pool constellation**

**Note:**
Although only one block is active in the Tiny EEL at a time, it makes sense to assign several blocks to the EEL pool. This way, hot spots on one block are reduced and wear leveling is improved. On the one hand, this means that the number of erases for one block is effectively reduced increasing the lifetime of the device. On the other hand, it enables a larger number of variable updates (writes) during the device life cycle.

The overall life cycle of a block in the EEL pool is shown in Figure 4. During normal operation, the block switches between active and invalid status. In case of a repeated error during block accessing, the affected flash block is considered to be defect and is marked as excluded. This block will not enter the lifecycle again. However, the user can try to reanimate the block by a format of the complete pool-which also erases all existing variable content (see Section 4.3.4 for details of the format command).

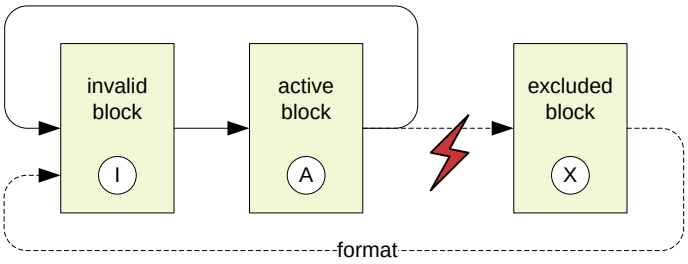


**Figure 4: Life cycle of an EEL block**

Not only the status of each EEL block is classified. There are also four different statuses of the overall EEL pool that can be distinguished:

- **Pool operational:** This is the usual case during EEL operation. All commands are available and can be executed.
- **Pool full:** The current active block is full and does not have enough space left to perform a write. This is a usual situation during library operation and indicates that a refresh needs to be executed. (There is a correlation with the size of the variable to be written.)
- **Pool exhausted:** Too many blocks have been excluded for full operation of the library. (The Tiny EEL requires at least two non-excluded blocks for operation.) Reading variables and verification of the active block is still possible, however writing variables and a refresh cannot be executed anymore. A reanimation of the library can only be attempted by a format of the pool.
- **Pool inconsistent:** The pool is not consistent, i.e. the data structures found in the EEL blocks do not match the expected structures of EEL blocks. Typically, this is the case when the pool has not yet been formatted with the Tiny EEL or when the pool content has been corrupted by flashing the device.

### 2.3.2 EEL Block Structure

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

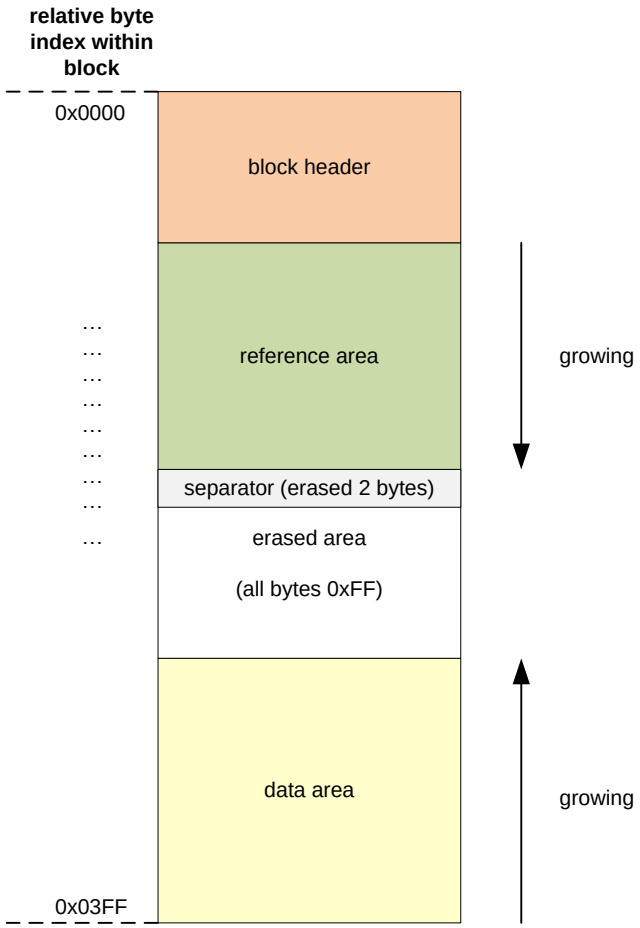


Figure 5: EEL block structure

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 8 bytes.

- **Reference area:** The reference area contains reference entries which are required for the management of EEL variables. It grows in direction of larger indexes whenever a variable is written.
- **Data area:** The data area contains the pure data values of the defined EEL variables. 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 (erased flash cells in MF3 have a value of 0xFF). With each EEL variable update (i.e. the variable is written), this area is reduced successively. However, at least two bytes of space always remain between reference and data area for management and separation of these areas. 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 of each flash block belonging to the EEL pool. It contains all information necessary for block management during EEL operation. The structure of the block header is depicted in Figure 6. It is composed of eight bytes, four of which are reserved for future use.

| relative byte index within block | | |
|---|---|---|
| 0x0000 | A | N |
| 0x0001 | B | ~N |
| 0x0002 | I | 0x00 |
| 0x0003 | X | 0x00 |
| 0x0004 | - | reserved |
| 0x0005 | - | reserved |
| 0x0006 | - | reserved |
| 0x0007 | - | reserved |

**Figure 6: Structure of each EEL block header**

Inside the header area, a set of status flags is used to code the block status in a reset-resistant manner. Each of the four flags has a size of one byte and fulfills the following purpose:

- **A and B flag:** The A and B flag are used to indicate the activation of a block. In case of an *active* block, the value of A falls in the range between 0x01 and 0x03, while the B flag contains the binary complement of A. In case of any other A/B-flag combination, the block is treated as invalid. Table 1 shows the possible A/B-flag combinations together with the corresponding block status.

**Table 1: Valid pattern combinations of the activation flags A and B**

| A(x) | B(x) | Block x status | Comment |
|---|---|---|---|
| 0x01 | 0xFE | active | 0x01 is always older than 0x02 |
| 0x02 | 0xFD | active | 0x02 is always older than 0x03 |
| 0x03 | 0xFC | active | 0x03 is always older than 0x01 |
| other combination | | invalid | other values are irregular |

- **I flag:** By means of the I flag, a block can be actively marked as invalid. The I flag is treated as a destructive flag, meaning that any value different from 0xFF (value of an erased cell) indicates an invalid block. The I flag overrules any status information of the A/B flag.
  Several processes of the EEL use this mechanism of active invalidation in order to achieve a consistent pool constellation.
- **X flag:** The X flag is a destructive flag used to exclude blocks from the block management. Any value different from 0xFF indicates an excluded block. The X flag overrules all other block flags. The only one way to reactivate a once excluded block is to format the whole EEL pool.

### 2.3.3 Structure of Variable Instances

By means of the so-called EEL descriptor, the developer can configure a set of variables to be used within the EEL. Each variable is referred to by an identification number (ID) and can have a size between 1 and 255 byte. (The exact specification of the format of the EEL descriptor can be found in Section 5.4.3.)

Whenever a variable content is updated, i.e. the variable is written, a new *instance* of the variable 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 variable.

A variable instance is composed of three parts: the start-of-record (SoR) field the end-of-record (EoR) field and the data field. SoR and EoR build up the so-called *reference* which is required for the management of the variable instance. The reference entry and data values are stored in different sections of the active block, namely the reference area and the data area, respectively.

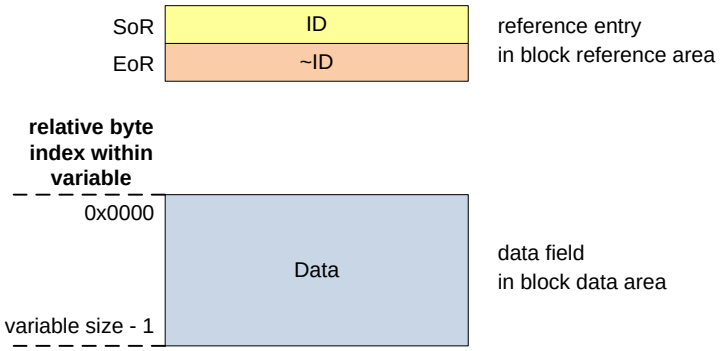


**Figure 7: Structure of an EEL variable instance**

The overall structure of a variable instance is abstracted in Figure 7. A more precise description of the three different fields of an instance is given in the following:

- **SoR field:** The 1-byte SoR field contains the ID code of the corresponding EEL variable. It indicates the start of a write sequence as described later in this section. The IDs 0x00 and 0xFF are never used and illegal in order to avoid patterns of erased cells.
- **EoR filed:** The 1-byte EoR field contains the binary inversion of ID code. It indicates the successful end of a write sequence. When not completely written—e.g. due to a reset of the device—the corresponding instance is ignored by the EEL.
- **Data field:** The data field contains the net data of the EEL variable. The size of the data filed matches the size of the corresponding variable and ranges from 1 to 255 bytes. Lower variable indices (in case of multi-byte variables) are stored at lower addresses in the data section of the active block.

**Note 1:**
The total size of the reference consumed by each variable instance is 2 bytes. This should be considered when evaluating the free space in a block before writing the variable through the EEL_GetSpace function (see Section 4.2.7).

**Note 2:**
Due to strict resource limitations applied for the Tiny EEL, the data filed is not protected automatically by any checksum. If required, the developer can integrate such checksums in the net data of the variable manually.

### 2.3.4 EEL Block Overview

Putting together the information of the previous sections, Figure 8 shows a detailed example of an active EEL block containing several variable instances. The following variables are declared in this example:

- ID 0x01 with size = 0x04,
- ID 0x02 with size = 0x01,
- ID 0x03 is defined but not written here,
- ID 0x04 with size = 0x02.

The variables have been written in the sequence ID 0x01 → ID 0x04 → ID 0x02. In this example, the variable with ID 0x03 has not been written yet.

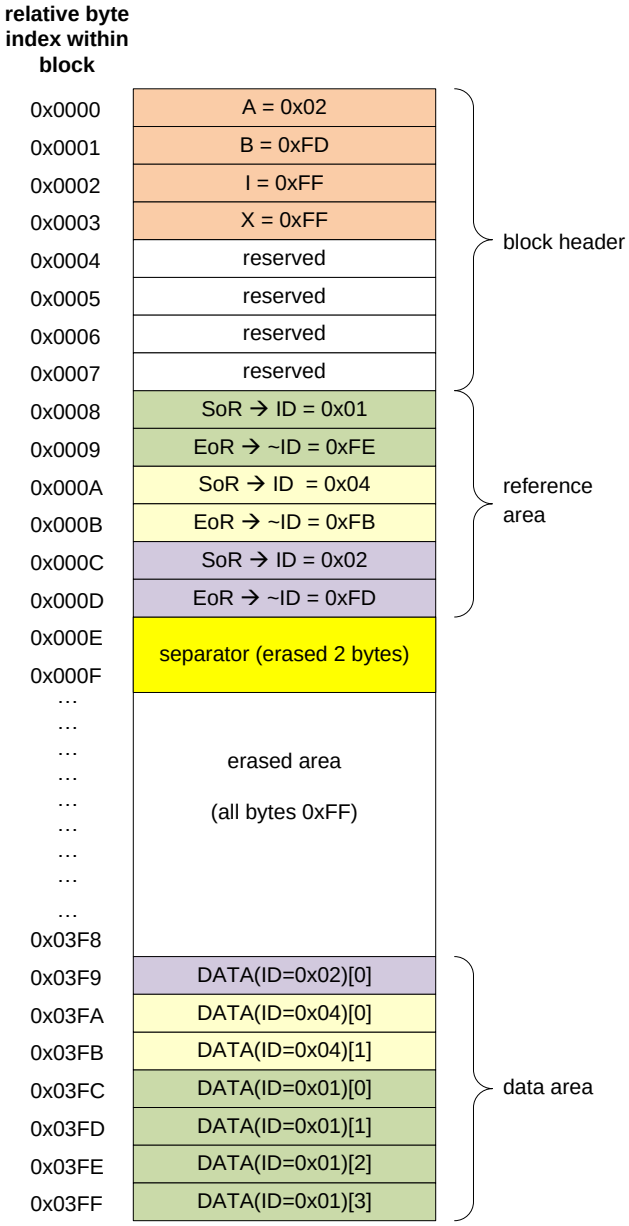


**Figure 8: Exemplary detailed content of an active EEL block**

# Chapter 3 EEL Operation

This chapter introduces the operation of the Tiny 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 Tiny EEL are introduced in the following. The API of the Tiny 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 EEL variables.

### 3.1.1 EEL Functions

The following functions are provided to control the Tiny EEL:

- **EEL_Init:** This function is used to initialize the internal data structures of the Tiny EEL. It is mandatory to call EEL_Init before any utilization of the library.
- **EEL_Open:** EEL_Open activates the library operation.
- **EEL_Close:** The library can be deactivated again via EEL_Close.
- **EEL_Execute:** By means of the EEL_Execute function, the user can issue commands to access and manage the EEL variables. 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 to EEL_Handler.
- **EEL_Handler:** The EEL_Handler needs to be called regularly to drive pending commands and observe their progress.
- **EEL_GetDriverStatus:** This function opens a way to check the internal status of the EEL driver before placing a request.
- **EEL_GetSpace:** By means of this function, the developer can check the remaining free space in the current active block, thereby aiding the decision when a refresh (i.e. transition to a new active block) is required.
- **EEL_GetVersionString:** This function provides library version information at runtime.

### 3.1.2 EEL Commands

Commands are used to manage the EEL pool and to access the EEL variable set. The actual EEL variable set is defined by the designer at compile time and can be configured in terms of number and size of variables.

Commands are initiated via EEL_Execute and processed stepwise by consecutive calls of 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 Tiny EEL:

**Pool-oriented Commands**

- **Startup:** The startup command analyses the current EEL pool and needs to be executed before EEL variables can be accessed.
- **Shutdown:** Shutdown is the counterpart to the startup command and required for a controlled ending of the EEL operation.
- **Format:** By means of the format command, an initial empty EEL pool structure is created. All existing EEL variable content is lost during this step.

- **Refresh:** The refresh command triggers the activation of a new EEL block. Thereby the old active block is invalidated and all current variable instances are copied to the new block. A refresh needs to be performed whenever the current active block does not provide enough space to update a variable via a write command.
- **Verify:** The verify command provides a way to check for full data retention of the flash cells in the active EEL block.

**Variable-oriented Commands**

- **Write:** The write command is used to update the content of an EEL variable.
- **Read:** The read command provides read access to EEL variables.

### 3.1.3 Request-Response-oriented Dialog

The Tiny EEL utilizes a request-response architecture to initiate the commands. This means any "requester" (any tasks in the user application) has to prepare a request variable as a kind of "request form sheet" and pass it by reference to the Tiny EEL driver using its EEL_Execute function. The Tiny 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 EEL_Handler periodically as long the request remains "busy".

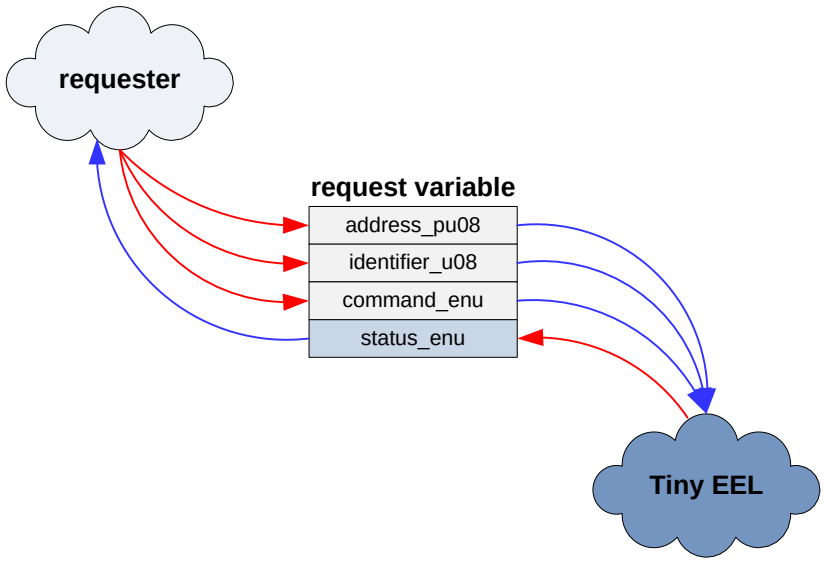


Figure 9: Schematic usage of the request variable

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. An additional advantage is the possibility to isolate the dialog between several requesters and the driver in multi-tasking systems. Thereby, multiple request variables can be used from different tasks.

The details on the request variable structure and its members are given later in Section 4.1.5. Please also note that not all structure members are required for all commands. The individual command descriptions in Section 4.3 provide the corresponding detailed information.

### 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 EEL_Execute.

2. Processing of the requested command state by state using 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 tasks.

Please note that 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 the EEL_Handler, where the internal sequences of the command are executed state by state. Together with the background-operation feature of the Tiny 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.

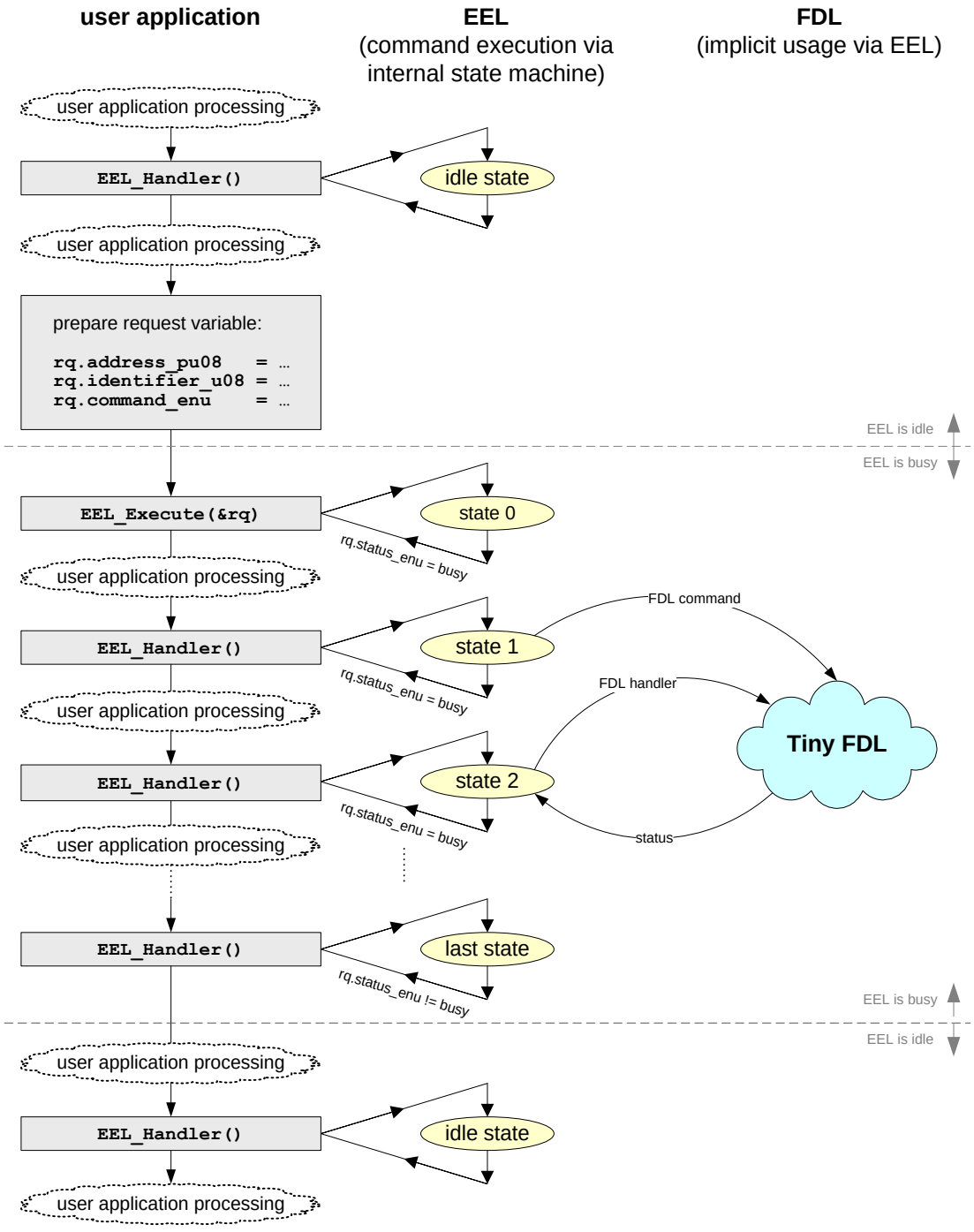


**Figure 10: Schematic illustration of the polling operation mode**

This way of controlling the EEL is also referred to as *polling mode*. Figure 10 illustrates the schematic flow of this operation mode.

**Note 1:**
While no command is being processed, EEL_Handler consumes a few CPU cycles only. The polling mode avoids blocking by the EEL.

**Note 2:**
In order to minimize the code size and resource consumption of the library, the Tiny EEL only supports the polling mode. Other operation modes like the enforced and timeout mode known from the RL78 EEL Type T01 have been omitted in the Tiny EEL.

**Note 3:**
As the first state of the EEL command is already executed by the call of EEL_Execute, there are commands which can finish their operation immediately after EEL_Execute and do not require to call the EEL_Handler. Please see the individual command description for details.
From perspective of the user application, the safest way to deal with the execution of commands is to always follow the above mentioned procedure and check the status after each call of EEL_Execute and EEL_Handler.

## 3.2 Basic Workflow

The basic workflow for utilizing the Tiny EEL is shown in Figure 11. The internal states of the EEL are depicted in ellipses with the state name printed in bold letters. Along with the internal state, the driver status is given, which can be retrieved any time via the EEL_GetDriverStatus function. This driver status can be one of the following:

- **PASSIVE:** The EEL is not started up completely. It is not possible to issue EEL commands (except startup and format in opened state).
- **BUSY:** The EEL is currently processing a command and cannot accept a new one for execution.
- **IDLE:** The EEL has currently no running tasks and is accepting new commands.

States and transitions depicted in bold lines indicate normal library operation, while the states and transitions in thin lines indicate exceptional operation. The detailed flow is described in the following.

After powering up the device, the Tiny FDL needs to be initialized and opened. This must be done before any EEL function is executed. EEL_Open and EEL_Close can then be used to activate and deactivate the EEL, i.e. to switch between *closed* and *opened* state.

When the user application is started for the very first time, it is usually necessary to format the EEL pool before any EEL variable can be written. In order not to process random data during startup, it is necessary to perform this initial format from the *opened* state.

If the EEL pool contains valid data already, it is necessary to perform a startup command from the *opened* state, which processes the current pool structure and builds up internal lookup tables for fast read/write access to the EEL variables. Thereby, the library is driven into the *started-up* state—in case no error occurs.

From the *started-up* state, the commands for usual operation can be executed (i.e. read, write, refresh and verify).

Problems during startup can either indicate an inconsistent pool (e.g. not formatted before first usage) or an exhaustive pool (i.e. there are not sufficient non-excluded blocks left for library operation). While the library returns into the *opened* state in the former case, it moves to the *exhausted* state in the latter. From the *exhausted* state, only read and verify commands can be executed. Writing and refreshing is not possible.

Please note that an exhausted-pool scenario can also occur during a refresh command. Therefore, a transition from the *full-access execution* state to the *exhausted* state is possible.

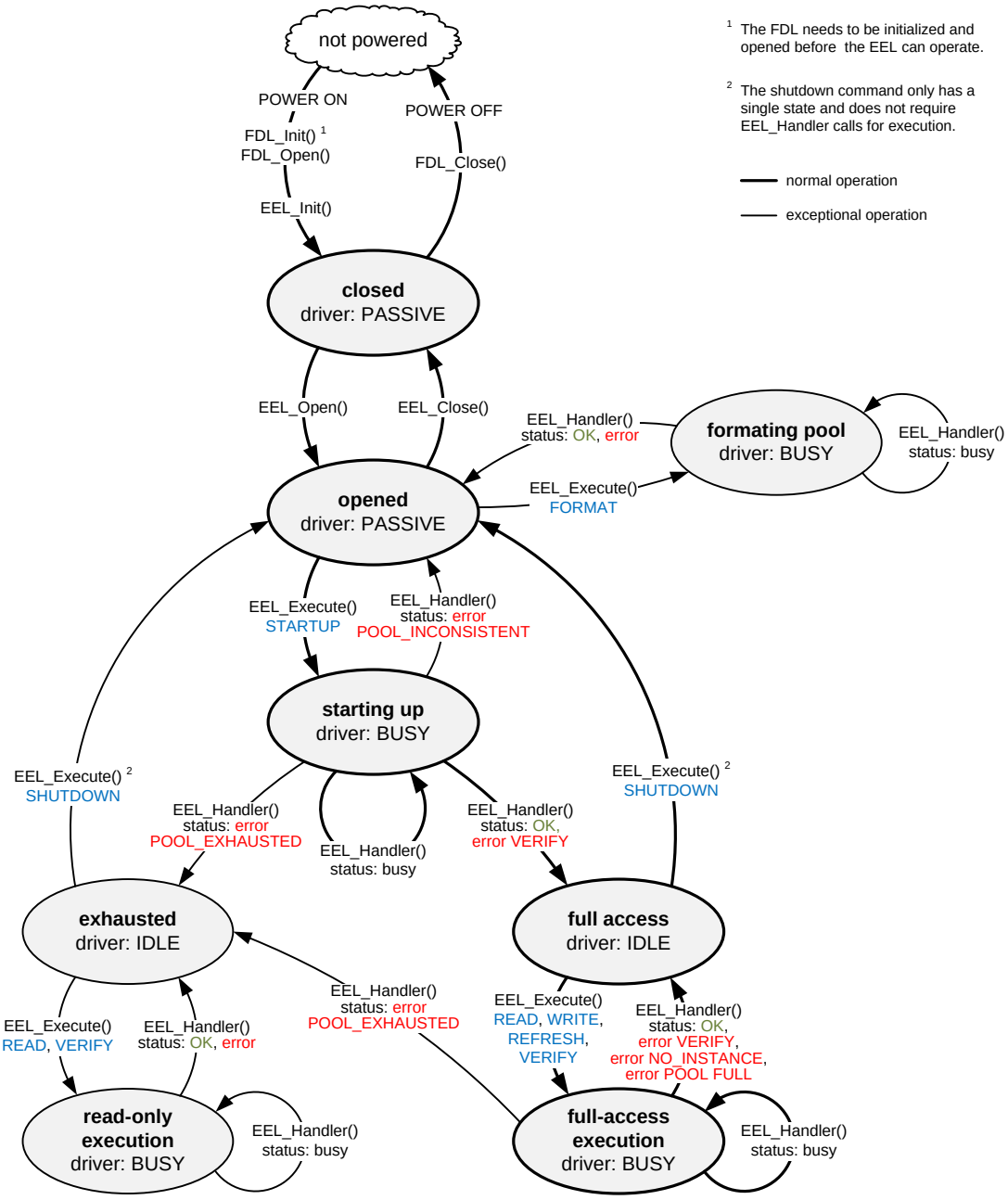


**Figure 11: Basic Tiny EEL workflow**

**Note 1:**
Figure 11 shows only the basic workflow. Few other transitions are possible and not blocked by the EEL, however the developer is strongly advised to use only the depicted transitions as long as there is no urgent reason to deviate from the proposed flow. Other possible state transitions are listed in the following:

- EEL_Close and EEL_Init can be called from every state. Please note however, that this interrupts any EEL processing and can lead to unpredictable behavior.
- The format command can be triggered also from the *started-up* state. However, the library will always return to the opened state after completion of the command.
- The startup command can be triggered also from the *started-up* state. The library will perform a complete startup procedure and then return to the *started-up* state (as long as no error occurs). During normal operation however, such processing is not meaningful.
- A power-off transition can happen in any state and leads to state “not powered”.

- When the Tiny EEL is busy, the library cannot accept new commands and a request will be terminated with status “rejected”.

**Note 2:**
Once the format command is started, it must be finished before the Tiny EEL can be used again.

# Chapter 4 Application Programming Interface (API)

This chapter provides the formal description of the application programming interface of the EEPROM Emulation Library Type T02 (Tiny EEL). 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 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.

**C API**

All types are defined in the header file EEL_types.h, which is part of the library installation package. It contains simple type definitions as well as library specific enumeration types and structs.

**Assembler API**

While there are no structs and enumeration types available in assembler, only constant definitions according to the C enumeration types are provided in EEL_types.inc. Please derive the order and size of struct members from the C type definitions.

### 4.1.1 Library-specific Simple-Type Definitions

**Type definition:**

```
typedef unsigned char      eel_u08;
typedef unsigned int       eel_u16;
typedef unsigned long int  eel_u32;
```

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

### 4.1.2 eel_command_t

**Type definition:**

```
typedef enum
{
    EEL_CMD_UNDEFINED          = (0x00 | 0x00),
    EEL_CMD_STARTUP            = (0x00 | 0x01),
    EEL_CMD_WRITE              = (0x00 | 0x02),
    EEL_CMD_READ               = (0x00 | 0x03),
    EEL_CMD_REFRESH            = (0x00 | 0x04),
    EEL_CMD_VERIFY             = (0x00 | 0x05),
    EEL_CMD_FORMAT             = (0x00 | 0x06),
    EEL_CMD_SHUTDOWN           = (0x00 | 0x07)
} eel_command_t;
```

**Description:** The enumeration type eel_command_t defines all allowed codes used to specify library commands. This type is used within the structure eel_request_t (see Section 4.1.5) in order to specify which command shall be executed via the function EEL_Execute. A detailed description of each command can be found in Section 4.3.

Please note that due to the fact that the library has been implemented in Assembler, it is mandatory that the enumeration type eel_command_t has a size of exactly 1 byte.

**Note:**
The GNU compiler uses 16-bit enumeration types by default. Therefore, for GNU compiler, the declaration of the enumeration type has to be extended with an attribute in order to be compiled to 1 byte: “__attribute__ ((__packed__))”.

**Member / Value:**

| Member / Value | Description |
|---|---|
| EEL_CMD_UNDEFINED | undefined command |
| EEL_CMD_STARTUP | plausibility check of the EEL data and driver |
| EEL_CMD_WRITE | creates a new instance of an EEL variable |
| EEL_CMD_READ | reads latest instance of an EEL variable |
| EEL_CMD_REFRESH | copy the latest instance of each variable into a new active block |
| EEL_CMD_VERIFY | verify the active block byte by byte to check for full data retention |
| EEL_CMD_FORMAT | format the EEL pool, all instances are lost |
| EEL_CMD_SHUTDOWN | deactivates the EEL and drives it into a secure state |

### 4.1.3 eel_driver_status_t

**Type definition:**

```
typedef enum
{
    EEL_DRIVER_PASSIVE        = (0x00 | 0x00),
    EEL_DRIVER_IDLE           = (0x30 | 0x01),
    EEL_DRIVER_BUSY           = (0x30 | 0x02)
} eel_driver_status_t;
```

**Description:** The enumeration type eel_driver_status_t defines all codes of available driver operation statuses.
This type is used in the function EEL_GetDriverStatus (see Section 4.2.6) providing access to the driver status of the library.

Please note that due to the fact that the library has been implemented in Assembler, it is mandatory that the enumeration type eel_driver_status_t has a size of exactly 1 byte.

**Note:**
The GNU compiler uses 16-bit enumeration types by default. Therefore, for GNU compiler, the declaration of the enumeration type has to be extended with an attribute in order to be compiled to 1 byte: “__attribute__ ((__packed__))”.

**Member / Value:**

| Member / Value | Description |
|---|---|
| EEL_DRIVER_PASSIVE | library is not yet started up completely |
| EEL_DRIVER_IDLE | no command is being executed, new commands can be accepted |
| EEL_DRIVER_BUSY | a command is just being executed, new commands cannot be accepted |

### 4.1.4 eel_status_t

**Type definition:**

```
typedef enum
{
    EEL_OK                    = (0x00 | 0x00),
    EEL_BUSY                  = (0x00 | 0x01),
    EEL_ERR_CONFIGURATION     = (0x80 | 0x02),
    EEL_ERR_INITIALIZATION    = (0x80 | 0x03),
    EEL_ERR_ACCESS_LOCKED     = (0x80 | 0x04),
    EEL_ERR_PARAMETER         = (0x80 | 0x05),
    EEL_ERR_VERIFY            = (0x80 | 0x06),
    EEL_ERR_REJECTED          = (0x80 | 0x07),
    EEL_ERR_NO_INSTANCE       = (0x80 | 0x08),
    EEL_ERR_POOL_FULL         = (0x80 | 0x09),
    EEL_ERR_POOL_INCONSISTENT = (0x80 | 0x0A),
    EEL_ERR_POOL_EXHAUSTED    = (0x80 | 0x0B),
    EEL_ERR_INTERNAL          = (0x80 | 0x0C)
} eel_status_t;
```

**Description:** The enumeration type eel_status_t defines all codes of function and command statuses and errors. On the one hand, it is used as return type of the functions EEL_Init (see Section 0) and EEL_GetSpace (see Section 4.2.7). On the other hand it is used within the structure eel_request_t (see Section 4.1.5) in order to capture the processing of currently running command. Thereby, the possible error codes are command specific and described in detail in Section 4.3 along with the commands.

Please note that due to the fact that the library has been implemented in Assembler, it is mandatory that the enumeration type eel_status_t has a size of exactly 1 byte.

**Note:**
The GNU compiler uses 16-bit enumeration types by default. Therefore, for GNU compiler, the declaration of the enumeration type has to be extended with an attribute in order to be compiled to 1 byte: “__attribute__ ((__packed__))”.

**Member / Value:**

| Member / Value | Description |
|---|---|
| EEL_OK | no error occurred |
| EEL_BUSY | request is under processing |
| EEL_ERR_CONFIGURATION | bad FDL/EEL configuration or combination |
| EEL_ERR_INITIALIZATION | call to EEL_Init(), EEL_Open() missing |
| EEL_ERR_ACCESS_LOCKED | execution of startup command missing |
| EEL_ERR_PARAMETER | wrong parameter (wrong command or identifier) |
| EEL_ERR_VERIFY | at least one byte in the active block does not provide the full data retention |
| EEL_ERR_REJECTED | request rejected as another request is already being processed |
| EEL_ERR_NO_INSTANCE | no instance found (variable never written) |
| EEL_ERR_POOL_FULL | not enough space for writing data |
| EEL_ERR_POOL_INCONSISTENT | EEL pool is inconsistent |
| EEL_ERR_POOL_EXHAUSTED | EEL pool is too small for write/refresh operation |
| EEL_ERR_INTERNAL | internal error (should never occur), please contact support |

### 4.1.5 eel_request_t

**Type definition:**

```
typedef struct
{
    __near eel_u08*       address_pu08;
           eel_u08        identifier_u08;
           eel_command_t  command_enu;
           eel_status_t   status_enu;
} eel_request_t;
```

**Description:** The structured type eel_request_t defines the structure of EEL request variables. It is the central type for the request-response-oriented dialog for the execution of commands (see Section 3.1.3). Not every element of this structure is required for each command. However, all members of the request variable must be initialized once before usage. Please refer to Section 4.3 for a more detailed description and the command-specific usage of the structure elements.

Please note that any variable of the type eel_request_t needs to be located at an even address.

**Note:**
The GNU compiler does not require the "__near" keyword to declare near pointers. All pointers are near by default as long as the "__far" keyword is not used.

**Member / Value:**

| Member / Value | Description |
|---|---|
| address_pu08 | source/destination RAM-address |
| identifier_u08 | variable identifier |
| command_enu | command to be processed (see Section 4.1.2) |
| status_enu | status/error code after command execution (see Section **Error! Reference source not found.**) |

### 4.1.6 eel_descriptor

**Definition:**

```
__far const eel_u08 eel_descriptor[EEL_VAR_NO+2] =
{
    (eel_u08)(EEL_VAR_NO),      /* variable count   */  \
    (eel_u08)(sizeof(type_A)),  /* id = 1           */  \
    (eel_u08)(sizeof(type_B)),  /* id = 2           */  \
    ...
    (eel_u08)(sizeof(type_Z)),  /* id = EEL_VAR_NO  */  \
    (eel_u08)(0x00),            /* zero terminator  */  \
};
```

**Description:** The variable eel_descriptor is actually not a type. Nevertheless, it is included in the list of data types here, because its definition is mandatory for the operation of the EEL. The eel_descriptor specifies the EEL variable set by means of a constant byte array.

An exemplary descriptor can be found in eel_descriptor.c and needs to be adapted according to the application requirements. The first element of the array represents the number of defined variables. This element is followed by the size of each variable which may be between 1 and 255 bytes. Thereby, the index within the array matches the corresponding variable ID. The variable list needs to be zero terminated in the last element.

Further information on how to configure the variable set can be found in Section 5.4.3.

## 4.2 Functions

Due to the request-oriented interface of the Tiny EEL, the functional interface is very narrow. Beside the initialization and some administrative functions, the access to the EEL pool concentrates on two functions only: EEL_Execute and EEL_Handler.

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

**Note:**
The register naming conventions defined by the GNU Compiler differs from the standard one provided by Renesas. The mapping of registers used by this library is as follows:

**Table 2: Register naming conventions (GNU Compiler vs. standard)**

| Renesas | GNU Compiler |
|---|---|
| X (Register bank 1) | R8 |
| A (Register bank 1) | R9 |
| C (Register bank 1) | R10 |
| B (Register bank 1) | R11 |

### 4.2.1 EEL_Init

**Outline:** EEL_Init has to be called before any EEL usage. The function initializes all internal data and variables and performs a plausibility check of the configuration.

**Interface:** **C Interface for Renesas Compiler**

```
eel_status_t __far EEL_Init(void);
```

**C Interface for IAR Compiler**

```
__far_func eel_status_t EEL_Init(void);
```

**C Interface for GNU Compiler**

```
eel_status_t EEL_Init(void) __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_Init
```

**Arguments:** **Parameters**

none

**Return value**

| Type | Passed via | | |
|---|---|---|---|
| | REN | IAR | GNU |
| eel_status_t | C | A | R8 (X bank 1) |
| Status of the function execution. May be one of the following:<br>• EEL_OK: initialization done without problems<br>• EEL_ERR_CONFIGURATION: pool configuration error (EEL descriptor not plausible) or wrong FDL linked to the project | | | |

**Destructed registers**

none

**Pre-conditions:** The Tiny FDL must be initialized and opened successfully before EEL_Init can be executed.

**Post-conditions:** none

**Description:** EEL_Init has to be called before any EEL usage. The function initializes all internal data structures and variables and performs a plausibility check of the configuration (EEL descriptor).

First, the signature of the used FDL is checked to ensure that the right FDL (Type T02) is the counterpart of the EEL.

After that, the plausibility of the EEL descriptor is checked by this function. Thereby the following properties are evaluated:

- the number of defined EEL variables (at least 1, maximum 64),
- the variable count matches the number of entries in the eel_descriptor array (see descriptor.c),
- the zero-termination of eel_descriptor array (see eel_descriptor.c), and
- the total size of all defined variables is small enough to keep library operational (i.e. when all variables are instantiated once in the active block, the remaining free space is sufficient to write the largest variable at least once).

Please note that the plausibility checks only aim at checks for operability. Still, the developer can create configurations which are not efficient (e.g. require to trigger frequent refreshes in case of very large EEL variable sets).

**Example:**

```
eel_status_t    my_eel_status;

my_eel_status = EEL_Init();
if (my_eel_status != EEL_OK)
{
    MyErrorHandler();
}
```

### 4.2.2 EEL_Open

**Outline:** This function opens the EEL logically.

**Interface:** **C Interface for Renesas Compiler**

```
void __far EEL_Open(void);
```

**C Interface for IAR Compiler**

```
__far_func void EEL_Open(void);
```

**C Interface for GNU Compiler**

```
void EEL_Open(void) __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_Open
```

**Arguments:** **Parameters**

none

**Return value**

none

**Destructed registers**

none

**Pre-conditions:** EEL_Init has to be executed successfully before. (This implies that the Tiny FDL has been initialized and opened beforehand.)

**Post-conditions:** none

**Description:** By means of the EEL_Open function, the Tiny EEL can be opened logically for usage. Please note that this does not activate the data flash. Enabling the data flash has to be done via the Tiny FDL by the corresponding FDL_Open function.

**Example:**

```
EEL_Open();
```

### 4.2.3 EEL_Close

**Outline:** This function closes the EEL logically.

**Interface:** **C Interface for Renesas Compiler**

```
void __far EEL_Close(void);
```

**C Interface for IAR Compiler**

```
__far_func void EEL_Close(void);
```

**C Interface for GNU Compiler**

```
void EEL_Close(void) __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_Close
```

**Arguments:** **Parameters**

none

**Return value**

none

**Destructed registers**

none

**Pre-conditions:** none

**Post-conditions:** none

**Description:** By means of the EEL_Close function, the Tiny EEL can be closed logically. Please note that this does not deactivate the data flash. Disabling the data flash has to be done via the Tiny FDL by the corresponding FDL_Close function.

**Example:**

```
EEL_Close();
```

### 4.2.4 EEL_Execute

**Outline:** This function initiates the execution of an EEL command.

**Interface:** **C Interface for Renesas Compiler**

```
void __far EEL_Execute(__near eel_request_t* request_pstr);
```

**C Interface for IAR Compiler**

```
__far_func void EEL_Execute(__near eel_request_t __near* request_pstr);
```

**C Interface for GNU Compiler**

```
void EEL_Execute(eel_request_t* request_pstr)
                                __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_Execute
```

**Arguments:** **Parameters**

| Argument | Type | Access | Passed via | | |
|---|---|---|---|---|---|
| | | | REN | IAR | GNU |
| request_pstr | eel_request_t* (near) | rw | AX | AX | stack |
| This argument points to a request structure defining the command the EEL should execute. The request structure is used for bidirectional information exchange between EEL and application during command execution.<br>A detailed description of the eel_request_t structure can be found in Section 4.1.5. | | | | | |

**Return value**

none

**Destructed registers**

| Tool chain | Destructed registers |
|---|---|
| Renesas | AX |
| IAR | AX |
| GNU | none |

**Pre-conditions:** EEL_Init must be executed successfully before.
EEL_Open must be executed before.

**Post-conditions:** none

**Description:** By means of the EEL_Execute function, the execution of EEL commands can be initiated. The type and parameters of the command to be executed are collected in the request structure pointed to by request_pstr (see also Section 4.1.5).

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

A detailed description of all available commands can be found in Section 4.3. 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 command description.

EEL_Execute only *initiates* the command execution. Please be aware that it is necessary for (almost) all commands to complete the command execution by EEL_Handler calls as introduced in Section 3.1.4. However, depending on the command it is possible that the EEL_Handler is executed implicitly by EEL_Execute. Therefore, commands may finish directly after the EEL_Execute call, indicated by an update of the status member of the request variable.

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. Especially in case of operating systems, tasks can use their own request variables in order to decouple the EEL accesses between the tasks.

**Note 1:**
Although there are commands that do not require all request structure elements to be specified, the whole structure needs to be initialized before calling EEL_Execute. Otherwise, a RAM parity/ECC error may cause a reset of the device. For details, please refer to the document “User's Manual: Hardware” of your RL78 product.

**Note 2:**
The request structure used for execution has to be word-aligned, i.e. located at an even memory address.

**Example:** Code examples for the execution of commands can be found in Section 4.3.8.

### 4.2.5 EEL_Handler

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

**Interface:** **C Interface for Renesas Compiler**

```
void __far EEL_Handler(void);
```

**C Interface for IAR Compiler**

```
__far_func void EEL_Handler(void);
```

**C Interface for GNU Compiler**

```
void EEL_Handler(void) __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_Handler
```

**Arguments:** **Parameters**

none

**Return value**

none

**Destructed registers**

none

**Pre-conditions:** EEL_Init must be executed successfully before.

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

**Description:** The function 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 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.

Please be aware that the EEL_Handler may implicitly call the FDL_Handler and therefore finish pending FDL commands. However, the EEL_Handler shall not be used instead of the FDL_Handler to finish FDL commands, as it is not guaranteed that the EEL_Handler invokes the FDL_Handler always.

**Note:**
When no command is being processed, EEL_Handler consumes a few CPU cycles only.

**Example:**

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

### 4.2.6 EEL_GetDriverStatus

**Outline:** This function opens a way to check the internal status of the EEL driver before placing a request.

**Interface:** **C Interface for Renesas Compiler**

```
void __far EEL_GetDriverStatus(__near eel_driver_status_t*
                                                eel_driver_status_penu);
```

**C Interface for IAR Compiler**

```
__far_func void EEL_GetDriverStatus(__near eel_driver_status_t __near*
                                                eel_driver_status_penu);
```

**C Interface for GNU Compiler**

```
void EEL_GetDriverStatus(eel_driver_status_t* eel_driver_status_penu)
                                __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_GetDriverStatus
```

**Arguments:** **Parameters**

| Argument | Type | Access | Passed via | | |
|---|---|---|---|---|---|
| | | | REN | IAR | GNU |
| eel_driver_status_penu | eel_driver_status_t* (near) | w | AX | AX | stack |
| Pointer to the driver status variable which is updated by the EEL_GetDriverStatus function, which may be one of the following:<br>• EEL_DRIVER_PASSIVE: driver passive, EEL not started up correctly<br>• EEL_DRIVER_BUSY: driver busy, EEL cannot accept new commands<br>• EEL_DRIVER_IDLE: driver idle, EEL awaits new commands<br>The eel_driver_status_t enumeration type is defined in Section 0. | | | | | |

**Return value**

none

**Destructed registers**

| Tool chain | Destructed registers |
|---|---|
| Renesas | AX |
| IAR | AX |
| GNU | none |

**Pre-conditions:** none

**Post-conditions:** none

**Description:** The EEL_GetDriverStatus function opens a way to check the internal status of the EEL driver before placing a request. The returned status can be one of the following:

- EEL_DRIVER_PASSIVE:
  The EEL is not started up completely. It is not possible to issue EEL commands (except startup and format when the library is initialized and opened). An access to the EEL data is not possible.
- EEL_DRIVER_BUSY:
  The EEL is currently processing a command and cannot accept a new one for execution.
- EEL_DRIVER_IDLE:
  The library has been initialized, opened and started up (see Section 4.3.1) successfully. The EEL has currently no running tasks and is accepting new commands.

A more detailed correlation between driver status and the EEL workflow is given in Figure 11 (Section 3.2).

**Example:**

```
eel_driver_status_t    my_eel_driver_status;


EEL_GetDriverStatus((__near eel_driver_status_t*)&my_eel_driver_status);

if (my_eel_driver_status == EEL_DRIVER_PASSIVE)
{
    /* initialize EEL and execute the STARTUP command */
}
else
{
    /* ensure that the EEL driver is idle before issuing request */
    while (my_eel_driver_status == EEL_DRIVER_BUSY)
    {
        EEL_Handler();
        EEL_GetDriverStatus((__near eel_driver_status_t*)&my_eel_driver_status);
    }
}

/* the user can place requests here */
```

### 4.2.7 EEL_GetSpace

**Outline:** By means of this function, the developer can check the remaining free space in the current active block, thereby aiding the decision when a refresh (i.e. transition to a new active block) is required.

**Interface:** **C Interface for Renesas Compiler**

```
eel_status_t __far EEL_GetSpace(__near eel_u16* space_pu16);
```

**C Interface for IAR Compiler**

```
__far_func eel_status_t EEL_GetSpace(__near eel_u16 __near* space_pu16);
```

**C Interface for GNU Compiler**

```
eel_status_t EEL_GetSpace(eel_u16* space_pu16)
                                __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_GetSpace
```

**Arguments:** **Parameters**

| Argument | Type | Access | Passed via | | |
|---|---|---|---|---|---|
| | | | REN | IAR | GNU |
| space_pu16 | eel_u16* (near) | w | AX | AX | stack |
| Address of the space information variable which is updated during function execution. | | | | | |

**Return value**

| Type | Passed via | | |
|---|---|---|---|
| | REN | IAR | GNU |
| eel_status_t | C | A | R8 (X bank 1) |
| Status/error code of the execution of EEL_GetSpace. May be one of the following:<br>• EEL_OK: space value has been computed correctly<br>• EEL_ERR_INITIALIZATION: EEL not initialized, space cannot be computed<br>• EEL_ERR_ACCESS_LOCKED: EEL has not been started up, space cannot be computed<br>• EEL_ERR_REJECTED: an EEL command is running, space cannot be computed | | | |

**Destructed registers**

| Tool chain | Destructed registers |
|---|---|
| Renesas | AX |
| IAR | AX |
| GNU | none |

**Pre-conditions:** The Tiny EEL must be initialized, opened and started up before the free space in the active block can be calculated.

**Post-conditions:** none

**Description:** By means of this function, the developer can check the remaining free space in the current active block, thereby aiding the decision when a refresh (i.e. transition to a new active block) is required.

It is important to note that each instance of a variable consists of a reference entry of 2 bytes and the actual data section (see Section 2.3.3). Therefore, writing a variable of size *n* requires *n+2* bytes of free space.

**Note 1:**
In case the EEL pool is exhausted, the returned free space will always be 0.

**Note 2:**
In case of an error return value, the space parameter is not changed.

**Example:**

```
eel_status_t    my_eel_status_enu;
eel_u16         my_eel_space_u16;

/* execute the function */
my_eel_status_enu = EEL_GetSpace((__near eel_u16*)&my_eel_space_u16);

if (my_eel_status_enu != EEL_OK)
{
    MyErrorHandler();
}
else
{
    /* my_eel_space_u16 is valid */
    /* do something */
}
```

### 4.2.8 EEL_GetVersionString

**Outline:** This function provides library version information at runtime.

**Interface:** **C Interface for Renesas Compiler**

```
__far eel_u08* __far EEL_GetVersionString(void);
```

**C Interface for IAR Compiler**

```
__far_func eel_u08 __far* EEL_GetVersionString(void);
```

**C Interface for GNU Compiler**

```
eel_u08 __far* EEL_GetVersionString(void)
                                    __attribute__ ((section ("EEL_CODE")));
```

**ASM function label**

```
EEL_GetVersionString
```

**Arguments:** **Parameters**

none

**Return value**

| Type | Passed via | | |
|---|---|---|---|
| | REN | IAR | GNU |
| eel_u08* (far) | BC (low), DE (high) | HL (low), A (high) | R8-R11 (AX, BC bank 1) |
| Far (24 bit) address of the zero-terminated EEL version string. | | | |

**Destructed registers**

none

**Pre-conditions:** none

**Post-conditions:** none

**Description:** For version control at runtime the developer can use this function to find the starting character of the library version string.

The version string is a zero-terminated string constant that covers library-specific information and is based on the following structure: NMMMMTTTCCCCCXVVV..V, where:

- N : library type specifier (here ‘E’ for EEL)
- MMMM : series name of microcontroller (here ‘RL78’)
- TTT : type number (here T02)
- CCCCC : compiler information
  - ‘Rxxx_’ for Renesas compiler version xxx
  - ‘Ixxx_’ for IAR compiler version xxx
  - ‘Uxxxx’ for GNU compiler version xxxx
- X : memory/register models (here ‘G’ for general, i.e. all memory models)
- VVV..V : library version

Examples:

The version string of the Tiny EEL V1.00 for the Renesas compiler version 1.10 is: "ERL78T02R110_GV100"

The version string of the Tiny EEL V1.00 for the IAR compiler version 1.20 is: "ERL78T02I120_GV100"

The version string of the Tiny EEL V1.00 for the GNU compiler version 13.02 is: "ERL78T02U1302GV100"

**Example:**

```
__far  eel_u08*   my_version_string_pu08;

my_version_string_pu08 = EEL_GetVersionString();

PrintMyVersion(my_version_string_pu08);
```

## 4.3 Commands

Commands are used to manage the EEL pool and to access the EEL variable set (see also Section 3.1.2). They are initiated via EEL_Execute and processed stepwise by consecutive calls of EEL_Handler. All EEL commands have to be initiated by passing a completed EEL-request structure (see Section 4.1.5) as parameter of EEL_Execute. As depicted in Figure 12, the three structure elements command_enu, identifier_u08 and address_pu08 need to be specified by the user while status_enu is set by the library.

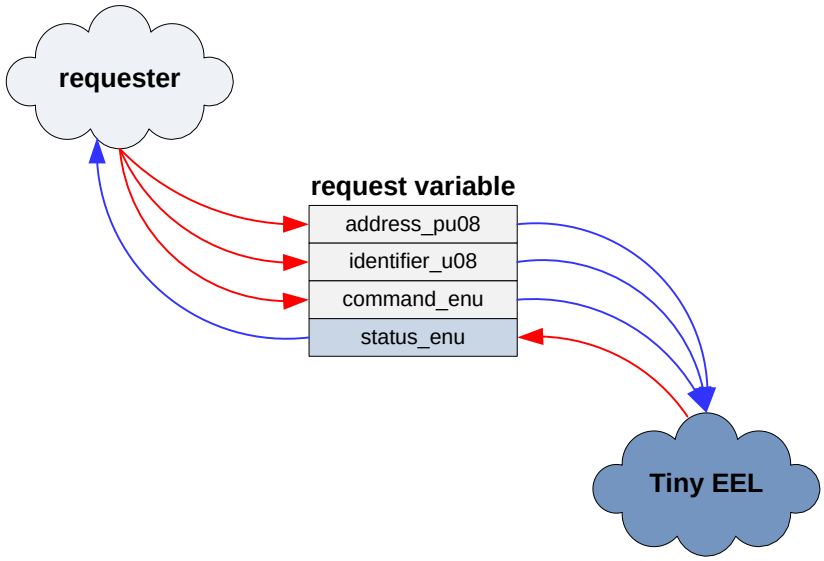


Figure 12: Schematic usage of the request variable (reprise of Figure 9)

While the mandatory element command_enu specifies which command is to be executed, the usage of the elements identifier_u08 and address_pu08 depends on the command. Therefore these parameters and the possible return statuses are described individually for each command in the subsequent sections.

An overview of all available command codes including the usage of request variable members is listed in Table 3.

Table 3: Command codes and usage of request variable members

| Command identifier | value | address_pu08 | identifier_u08 | Comment |
|---|---|---|---|---|
| EEL_CMD_STARTUP | 0x01 | unused | unused | startup the library |
| EEL_CMD_SHUTDOWN | 0x07 | unused | unused | shutdown the library |
| EEL_CMD_VERIFY | 0x05 | unused | unused | verify the current active block |
| EEL_CMD_REFRESH | 0x04 | unused | unused | copy latest instances to new block |
| EEL_CMD_FORMAT | 0x06 | unused | unused | format the EEL pool |
| EEL_CMD_READ | 0x03 | used | used | read variable content |
| EEL_CMD_WRITE | 0x02 | used | used | update variable content |

**Note:**
Although it depends on the command which members of the request structure are necessary for the execution, all members of the request variable must be initialized once (for instance directly after variable declaration). Thereby, unused members in the request variable can be to set arbitrary values.

In general, all EEL commands can be handled in the same way as illustrated in Figure 13 (a corresponding code example can be found in Section 4.3.8):

1. The requester fills up the private request variable my_request (command definition)
2. The requester tries to initiate the command execution by EEL_Execute(&my_request)
3. In case of a command rejection, the requester has to call the EEL_Handler first to proceed the processing of the already pending request and can retry by continuing with step 2.
4. The requester has to call 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 requester has to analyze the status to detect potential errors.

Not all commands can/should be executed from every library state. For details, please have a look at the basic workflow shown in Figure 11, Section 3.2.

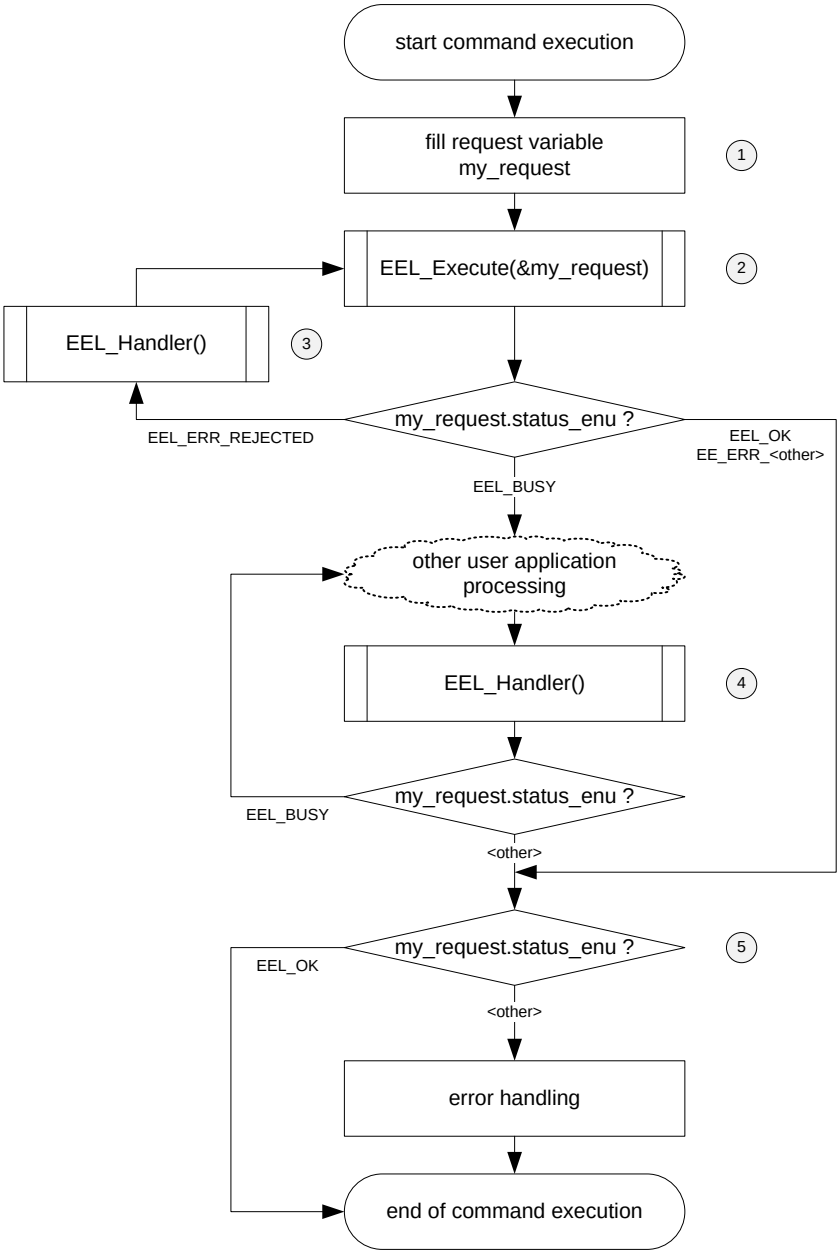


**Figure 13: Generic command execution flow**

The detailed command descriptions given in the subsequent sections include a classification of the corresponding returned statuses according to their severity:

- **normal:** normal operation status by the application has to handle at runtime,
- **light:** wrong handling that can be corrected by the application at runtime,
- **heavy:** wrong handling that can be corrected by redesign of the application,
- **fatal:** cannot be corrected without loss of data.

### 4.3.1 Startup

One main task of the startup command is to check the consistency of the EEL-pool. Especially the structure and consistency of the active block is checked (see also Section 2.3). Thereby, scenarios of reset-caused incomplete command executions are detected. Pool data structures of incomplete write and refresh commands are ignored and the previous data set is treated as valid.

Furthermore, during startup a reference table with indices of the effective instances of all EEL variables is built up in the RAM for fast access to the variable content.

Besides these two main tasks, the startup command also checks important parts of the active EEL block for full data retention (c.f. internal-verify command of Tiny FDL). If it is detected that the full data retention is not given, this is indicated by the error code EEL_ERR_VERIFY. In this case, the user should consider invoking a refresh command.

A successfully executed startup command enables read and write access to the EEL variables.

**Table 4: Request variable usage for startup command**

| command_enu | address_pu08 | identifier_u08 |
|---|---|---|
| EEL_CMD_STARTUP | unused | unused |

**Table 5: Statuses returned by the startup command**

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_INITIALIZATION | heavy | meaning | EEL or FDL not initialized or not opened |
| | | reason | wrong handling on user side |
| | | remedy | initialize and open EEL before using it |
| EEL_ERR_INTERNAL | heavy | meaning | error occurred but cannot be categorized |
| | | reason | FDL might have been reinitialized with a different descriptor or an unexpected internal error occurred |
| | | remedy | check/adjust the configured FDL descriptor and re-initialize all libraries. |
| EEL_ERR_PARAMETER | light | meaning | invalid command code |
| | | reason | unknown command code used in request |
| | | remedy | use command identifiers of eel_command_t |
| EEL_ERR_VERIFY | light | meaning | found data without full data retention in the block header or the last written variable instance |
| | | reason | long period of time since last refresh, interrupted write sequence (e.g. by reset) |
| | | remedy | execute a refresh command |
| EEL_ERR_POOL_INCONSISTENT | heavy | meaning | pool structure not usable |
| | | reason | EEL-pool is inconsistent and cannot be recovered |
| | | remedy | format the EEL pool |
| EEL_ERR_POOL_EXHAUSTED | fatal | meaning | EEL pool size is smaller than 2 blocks, write and refresh is not possible, reading existing variables is still possible |
| | | reason | too many blocks excluded |
| | | remedy | try to perform a format or just read existing data |

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_REJECTED | normal | meaning | EEL cannot accept the request |
| | | reason | EEL is busy with another request |
| | | remedy | call EEL_Handler and retry |
| EEL_BUSY | normal | meaning | request is being processed |
| | | reason | request checked and accepted |
| | | remedy | call EEL_Handler until status changes |
| EEL_OK | normal | meaning | request was finished regularly |
| | | reason | no problems during command execution |
| | | remedy | nothing |

**Note:**
A code example how to use the startup command can be found in Section 4.3.8.1.

### 4.3.2 Verify

The verify command performs the verification of the active block in the EEL pool, i.e. checks if all bytes in the active block provide full data retention. The user can execute the verify command after the library has been started up in order to ensure that no weak data were produced by a reset during writing. The developer can also execute the verify command before a shutdown in order to ensure that the EEL data will remain stable for the full data retention time after ending the EEL operation.

**Table 6: Request variable usage for verify command**

| command_enu | address_pu08 | identifier_u08 |
|---|---|---|
| EEL_CMD_VERIFY | unused | unused |

**Table 7: Statuses returned by the verify command**

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_INITIALIZATION | heavy | meaning | EEL or FDL not initialized or not opened |
| | | reason | wrong handling on user side |
| | | remedy | initialize and open EEL before using it |
| EEL_ERR_INTERNAL | heavy | meaning | error occurred but cannot be categorized |
| | | reason | FDL might have been reinitialized with a different descriptor or an unexpected internal error occurred |
| | | remedy | check/adjust the configured FDL descriptor and re-initialize all libraries. |
| EEL_ERR_PARAMETER | light | meaning | invalid command code |
| | | reason | unknown command code used in request |
| | | remedy | use command identifiers of eel_command_t |
| EEL_ERR_ACCESS_LOCKED | light | meaning | no access to EEL pool |
| | | reason | startup not executed or not successful |
| | | remedy | execute startup command |
| EEL_ERR_VERIFY | heavy | meaning | found data without full data retention in the active block |
| | | reason | long period of time since last refresh, interrupted write sequence (e.g. by reset) |
| | | remedy | execute refresh soon |
| EEL_ERR_REJECTED | normal | meaning | EEL cannot accept the request |
| | | reason | EEL is busy with another request |
| | | remedy | call EEL_Handler and retry |
| EEL_BUSY | normal | meaning | request is being processed |
| | | reason | request checked and accepted |
| | | remedy | call EEL_Handler until status changes |
| EEL_OK | normal | meaning | request was finished regularly |
| | | reason | no problems during command execution |
| | | remedy | nothing |

**Note:**
A code example how to use the verify command can be found in Section 4.3.8.4.

### 4.3.3 Shutdown

The shutdown command locks the logical access to the EEL data. To ensure full data retention time for all data in the active block, it is recommended to execute the verify command and—in case of an error—to perform a refresh before shutting down the library.

**Note:**
The shutdown command is the only command which has single processing state and hence finishes with EEL_Execute directly (i.e. without additional EEL_Handler call, c.f. Figure 11 in Section 3.2). However, the developer is advised to follow the common command flow proposed in Figure 13 also for the shutdown command.

**Table 8: Request variable usage for startup command**

| command_enu | address_pu08 | identifier_u08 |
|---|---|---|
| EEL_CMD_SHUTDOWN | unused | unused |

**Table 9: Statuses returned by the shutdown command**

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_INITIALIZATION | heavy | meaning | EEL or FDL not initialized or not opened |
| | | reason | wrong handling on user side |
| | | remedy | initialize and open EEL before using it |
| EEL_ERR_PARAMETER | light | meaning | invalid command code |
| | | reason | unknown command code used in request |
| | | remedy | use command identifiers of eel_command_t |
| EEL_ERR_ACCESS_LOCKED | light | meaning | no access to EEL pool |
| | | reason | startup not executed or not successful |
| | | remedy | execute startup command |
| EEL_ERR_REJECTED | normal | meaning | EEL cannot accept the request |
| | | reason | EEL is busy with another request |
| | | remedy | call EEL_Handler and retry |
| EEL_BUSY | normal | meaning | request is being processed |
| | | reason | request checked and accepted |
| | | remedy | call EEL_Handler until status changes |
| EEL_OK | normal | meaning | request was finished regularly |
| | | reason | no problems during command execution |
| | | remedy | nothing |

**Note:**
A code example how to use the startup command can be found in Section 4.3.8.4.

### 4.3.4 Format

The format command destroys all EEL variable content and creates an empty EEL pool with one active block. All remaining blocks are erased, independently of their block status. This means that also excluded blocks are tried to be erased, offering the developer a way to try to reanimate these blocks. However, in case of repeated erase problems, the format command excludes blocks automatically.

Typically, the format command is executed before the library is started up. However, it is also possible to execute the format command when the library has been started up already. In any case, the library needs to be started up again after a successful format.

In contrast to all other Tiny EEL commands, the format command is *not* robust against resets. In case a reset occurs during a format, there is a chance that the EEL pool is coincidentally valid and will be accepted during the next startup. Please ensure to always finish a once started format command.

Please note that this ostensible limitation is not severe in the typical use case as a format command destroys all EEL variable content and is therefore usually executed once at the beginning of a product lifetime.

**Note:**
When executing a format command, all EEL variable data will be lost immediately.

**Caution 1:**
It is imperative to finish a once started format command. The format command is not robust against resets. Unfinished format commands may not be detected and can lead to unpredictable behavior (i.e. data sets) during the next startup.

**Caution 2:**
When changing the FDL descriptor, an EEL format command must be executed successfully before the EEL can be started up and used again.

**Table 10: Request variable usage for format command**

| command_enu | address_pu08 | identifier_u08 |
|---|---|---|
| EEL_CMD_FORMAT | unused | unused |

**Table 11: Statuses returned by the format command**

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_INITIALIZATION | heavy | meaning | EEL or FDL not initialized or not opened |
| | | reason | wrong handling on user side |
| | | remedy | initialize and open EEL before using it |
| EEL_ERR_INTERNAL | heavy | meaning | error occurred but cannot be categorized |
| | | reason | FDL might have been reinitialized with a different descriptor or an unexpected internal error occurred |
| | | remedy | check/adjust the configured FDL descriptor and re-initialize all libraries. |
| EEL_ERR_PARAMETER | light | meaning | invalid command code |
| | | reason | unknown command code used in request |
| | | remedy | use command identifiers of eel_command_t |

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_POOL_EXHAUSTED | fatal | meaning | EEL pool size is smaller than 2 blocks,<br>write and refresh is not possible,<br>reading existing variables is still possible |
| | | reason | too many blocks excluded |
| | | remedy | try to perform a format or just read existing data |
| EEL_ERR_REJECTED | normal | meaning | EEL cannot accept the request |
| | | reason | EEL is busy with another request |
| | | remedy | call EEL_Handler and retry |
| EEL_BUSY | normal | meaning | request is being processed |
| | | reason | request checked and accepted |
| | | remedy | call EEL_Handler until status changes |
| EEL_OK | normal | meaning | request was finished regularly |
| | | reason | no problems during command execution |
| | | remedy | nothing |

**Note:**
A code example how to use the format command can be found in Section 4.3.8.2

### 4.3.5 Refresh

The refresh command copies the latest instance of each variable from the active block (source block) to the next block (destination block) in the EEL pool ring counting clockwise (see Section 2.3.1). Thereby, the destination block becomes the new active block.

By means of the refresh command, new free space is created which can be used to write (i.e. update) variable contents. Another reason for executing a refresh could be to ensure full data retention of all variables for instance before powering down the device.

The refresh sequence is designed in a way that a valid data set is available at any time. Therefore, the refresh command is robust against reset scenarios. The proper variable set will be reconstructed during the next startup.

**Note:**
Before a refresh command can be executed, the library needs to be started up successfully.

**Table 12: Request variable usage for refresh command**

| command_enu | address_pu08 | identifier_u08 |
|---|---|---|
| EEL_CMD_REFRESH | unused | unused |

**Table 13: Statuses returned by the refresh command**

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_INITIALIZATION | heavy | meaning | EEL or FDL not initialized or not opened |
| | | reason | wrong handling on user side |
| | | remedy | initialize and open EEL before using it |
| EEL_ERR_INTERNAL | heavy | meaning | error occurred but cannot be categorized |
| | | reason | FDL might have been reinitialized with a different descriptor or an unexpected internal error occurred |
| | | remedy | check/adjust the configured FDL descriptor and re-initialize all libraries. |
| EEL_ERR_PARAMETER | light | meaning | invalid command code |
| | | reason | unknown command code used in request |
| | | remedy | use command identifiers of eel_command_t |
| EEL_ERR_ACCESS_LOCKED | light | meaning | no access to EEL pool |
| | | reason | startup not executed or not successful |
| | | remedy | execute startup command |
| EEL_ERR_POOL_EXHAUSTED | fatal | meaning | EEL pool size is smaller than 2 blocks,<br>write and refresh is not possible,<br>reading existing variables is still possible |
| | | reason | too many blocks excluded |
| | | remedy | try to perform a format or just read existing data |
| EEL_ERR_REJECTED | normal | meaning | EEL cannot accept the request |
| | | reason | EEL is busy with another request |
| | | remedy | call EEL_Handler and retry |

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_BUSY | normal | meaning | request is being processed |
| | | reason | request checked and accepted |
| | | remedy | call EEL_Handler until status changes |
| EEL_OK | normal | meaning | request was finished regularly |
| | | reason | no problems during command execution |
| | | remedy | nothing |

**Note:**
A code example how to use the refresh command can be found in Section 4.3.8.3.

### 4.3.6 Write

The write command writes a new value to an EEL variable. The variable to be updated is specified via the identifier element of the request structure. The data to be written is copied from a buffer pointed to by the address element.

In case of insufficient free space in the active block, the write command returns EEL_ERR_POOL_FULL. In this case the developer needs to execute a refresh and then try to write the variable again.

In time critical scenarios it is also possible to check the remaining free space in the active block beforehand by means of the EEL_GetSpace function (see Section 4.2.7). This enables to detect when free space is getting scarce and issue a refresh already early when there is enough processing time. When evaluating the free space returned by EEL_GetSpace, please recall that a variable instance always requires two administrative bytes of data besides the actual variable contend.

The write sequence is designed in a way that a valid data set is available at any time. Therefore, the write command is robust against reset scenarios. The proper variable content will be reconstructed during the next startup.

**Note 1:**
Please ensure that the size of the data buffer used for writing the variable is at least as large as the size of the variable to be written and that all relevant buffer elements are initialized.

**Note 2:**
Before a write command can be executed, the library needs to be started up successfully.

**Table 14: Request variable usage for write command**

| command_enu | address_pu08 | identifier_u08 |
|---|---|---|
| EEL_CMD_WRITE | pointer to data buffer | identifier of variable to be written |

**Table 15: Statuses returned by the write command**

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_INITIALIZATION | heavy | meaning | EEL or FDL not initialized or not opened |
| | | reason | wrong handling on user side |
| | | remedy | initialize and open EEL before using it |
| EEL_ERR_INTERNAL | heavy | meaning | error occurred but cannot be categorized |
| | | reason | FDL might have been reinitialized with a different descriptor or an unexpected internal error occurred |
| | | remedy | check/adjust the configured FDL descriptor and re-initialize all libraries. |
| EEL_ERR_PARAMETER | light | meaning | invalid command code |
| | | reason | unknown command code used in request |
| | | remedy | use command identifiers of eel_command_t |
| EEL_ERR_ACCESS_LOCKED | light | meaning | no access to EEL pool |
| | | reason | startup not executed or not successful |
| | | remedy | execute startup command |
| EEL_ERR_POOL_EXHAUSTED | fatal | meaning | EEL pool size is smaller than 2 blocks,<br>write and refresh is not possible,<br>reading existing variables is still possible |
| | | reason | too many blocks excluded |
| | | remedy | try to perform a format or just read existing data |

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_POOL_FULL | normal | meaning | not enough space in active block to write the variable instance |
| | | reason | • space was consumed by previous write commands<br>• flash error occurred during writing data<br>• the active block contains an invalid reference (space consumed by previous startup command in case of invalid reference)<br>• the active block has weak data (previous startup command returned EEL_ERR_VERIFY error) |
| | | remedy | execute a refresh and write again |
| EEL_ERR_REJECTED | normal | meaning | EEL cannot accept the request |
| | | reason | EEL is busy with another request |
| | | remedy | call EEL_Handler and retry |
| EEL_BUSY | normal | meaning | request is being processed |
| | | reason | request checked and accepted |
| | | remedy | call EEL_Handler until status changes |
| EEL_OK | normal | meaning | request was finished regularly |
| | | reason | no problems during command execution |
| | | remedy | nothing |

**Note:**
A code example how to use the refresh command can be found in Section 4.3.8.3.

### 4.3.7 Read

The read command reads the current value of an EEL variable. The variable to be read is specified via the identifier element of the request structure. The data is copied to a buffer pointed to by the address element.

**Note 1:**
Please ensure that the size of the data buffer used for reading the variable is at least as large as the size of the variable.

**Note 2:**
Before a read command can be executed, the library needs to be started up successfully.

**Table 16: Request variable usage for read command**

| command_enu | address_pu08 | identifier_u08 |
|---|---|---|
| EEL_CMD_READ | pointer to data buffer | identifier of variable to be read |

**Table 17: Statuses returned by the read command**

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_ERR_INITIALIZATION | heavy | meaning | EEL or FDL not initialized or not opened |
| | | reason | wrong handling on user side |
| | | remedy | initialize and open EEL before using it |
| EEL_ERR_INTERNAL | heavy | meaning | error occurred but cannot be categorized |
| | | reason | FDL might have been reinitialized with a different descriptor or an unexpected internal error occurred |
| | | remedy | check/adjust the configured FDL descriptor and re-initialize all libraries. |
| EEL_ERR_PARAMETER | light | meaning | invalid command code |
| | | reason | unknown command code used in request |
| | | remedy | use command identifiers of eel_command_t |
| EEL_ERR_ACCESS_LOCKED | light | meaning | no access to EEL pool |
| | | reason | startup not executed or not successful |
| | | remedy | execute startup command |
| EEL_ERR_NO_INSTANCE | light | meaning | no instance of the variable with the specified identifier found |
| | | reason | no initial value written |
| | | remedy | write an initial value of the variable via the write command |
| EEL_ERR_REJECTED | normal | meaning | EEL cannot accept the request |
| | | reason | EEL is busy with another request |
| | | remedy | call EEL_Handler and retry |
| EEL_BUSY | normal | meaning | request is being processed |
| | | reason | request checked and accepted |
| | | remedy | call EEL_Handler until status changes |

| Status | Class | Status meaning and handling | |
|---|---|---|---|
| EEL_OK | normal | meaning | request was finished regularly |
| | | reason | no problems during command execution |
| | | remedy | nothing |

**Note:**
A code example how to use the refresh command can be found in Section 4.3.8.3.

### 4.3.8 Code examples

In the following, some code examples are given how to initiate and execute EEL commands.

#### 4.3.8.1 Library Startup

```
eel_status_t    my_eel_status;
eel_request_t   my_eel_request_str;

...

/* It is assumed at this point, that the FDL has been initialized and opnened
correctly. */

my_eel_status = EEL_Init();
if (my_eel_status != EEL_OK)
{
    MyErrorHandler();
}
EEL_Open();

/* Ensure initialized request structure ------------------------------ */
my_eel_request_str.command_enu      = EEL_CMD_UNDEFINED;
my_eel_request_str.identifier_u08   = 0;
my_eel_request_str.address_pu08     = NULL;
my_eel_request_str.status_enu       = EEL_ERR_PARAMETER;

/* Initiate STARTUP command ----------------------------------------- */
my_eel_request_str.command_enu      = EEL_CMD_STARTUP;

/* command cannot be rejected here as the library has just been opened */
EEL_Execute(&my_eel_request_str);

while (my_eel_request_str.status_enu == EEL_BUSY)
{
    EEL_Handler();
}

if (my_eel_request_str.status_enu != EEL_OK)
{
    MyErrorHandler();
}

/* Library is started up now and should accept accesses to the EEL variables */

/* Initiate READ command -------------------------------------------- */
my_eel_request_str.command_enu      = EEL_CMD_READ;
my_eel_request_str.address_pu08     = (__near eel_u08*)&A[0];
my_eel_request_str.identifier_u08   = 1;

do
{
    EEL_Handler();
    EEL_Execute(&my_eel_request_str);
}
while (my_eel_request_str.status_enu == EEL_ERR_REJECTED);

while (my_eel_request_str.status_enu == EEL_BUSY)
{
    EEL_Handler();
}

if (my_eel_request_str.status_enu != EEL_OK)
{
    MyErrorHandler();
}
```

### 4.3.8.2 EEL Pool Format

```
eel_status_t    my_eel_status;
eel_request_t   my_eel_request_str;

/* Ensure initialized request structure ------------------------------ */
my_eel_request_str.command_enu      = EEL_CMD_UNDEFINED;
my_eel_request_str.identifier_u08   = 0;
my_eel_request_str.address_pu08     = NULL;
my_eel_request_str.status_enu       = EEL_ERR_PARAMETER;

...


/* It is assumed at this point, that the FDL has been initialized and opnened
correctly. */

my_eel_status = EEL_Init();
if (my_eel_status != EEL_OK)
{
    MyErrorHandler();
}
EEL_Open();


/* Initiate FORMAT comamnd ------------------------------------------- */
my_eel_request_str.command_enu      = EEL_CMD_FORMAT;

/* command cannot be rejected here as the library has just been opened */
EEL_Execute(&my_eel_request_str);

while (my_eel_request_str.status_enu == EEL_BUSY)
{
    EEL_Handler();
}

if (my_eel_request_str.status_enu != EEL_OK)
{
    MyErrorHandler();
}

/* EEL pool is formatted now, all previous variable content is lost */
```

### 4.3.8.3 Variable Access

```
eel_status_t    my_eel_status;
eel_request_t   my_eel_request_str;
eel_request_t   my_refr_eel_request_str;

/* Ensure initialized request structure ------------------------------ */
my_eel_request_str.command_enu          = EEL_CMD_UNDEFINED;
my_eel_request_str.identifier_u08       = 0;
my_eel_request_str.address_pu08         = NULL;
my_eel_request_str.status_enu           = EEL_ERR_PARAMETER;

my_refr_eel_request_str.command_enu     = EEL_CMD_REFRESH;
my_refr_eel_request_str.identifier_u08  = 0;
my_refr_eel_request_str.address_pu08    = NULL;
my_refr_eel_request_str.status_enu      = EEL_ERR_PARAMETER;

...

/* It is assumed at this point, that the EEL has been initialized, opnened and
started up correctly. */


/* Initiate READ command --------------------------------------------- */
my_eel_request_str.command_enu      = EEL_CMD_READ;
my_eel_request_str.address_pu08     = (__near eel_u08*)&A[0];
my_eel_request_str.identifier_u08   = 2;

do
{
    EEL_Handler();
    EEL_Execute(&my_eel_request_str);
}
while (my_eel_request_str.status_enu == EEL_ERR_REJECTED);

while (my_eel_request_str.status_enu == EEL_BUSY)
{
    EEL_Handler();
}

if (my_eel_request_str.status_enu != EEL_OK)
{
    MyErrorHandler();
}


/* Initiate WRITE command -------------------------------------------- */
my_eel_request_str.command_enu      = EEL_CMD_WRITE;
my_eel_request_str.address_pu08     = (__near eel_u08*)&B[0];
my_eel_request_str.identifier_u08   = 1;

do
{
    do
    {
        EEL_Handler();
        EEL_Execute(&my_eel_request_str);
    }
    while (my_eel_request_str.status_enu == EEL_ERR_REJECTED);

    while (my_eel_request_str.status_enu == EEL_BUSY)
    {
        EEL_Handler();
    }

    if (my_eel_request_str.status_enu == EEL_ERR_POOL_FULL)
    {
```

```
        /* in case of a full pool, a refresh needs to be executed
        in order to free space */

        /* please note that a different request variable is used for the
        refresh so that the result of the write command can be checked in the
        outer loop */

        do
        {
            EEL_Handler();
            EEL_Execute(&my_refr_eel_request_str);
        }
        while (my_refr_eel_request_str.status_enu == EEL_ERR_REJECTED);

        while (my_refr_eel_request_str.status_enu == EEL_BUSY)
        {
            EEL_Handler();
        }

        if (my_refr_eel_request_str.status_enu != EEL_OK)
        {
            MyErrorHandler();
        }
    }
    else if (my_eel_request_str.status_enu != EEL_OK)
    {
        MyErrorHandler();
    }
}
while (my_eel_request_str.status_enu == EEL_ERR_POOL_FULL)
```

### 4.3.8.4 Library Shutdown

```
eel_status_t    my_eel_status;
eel_request_t   my_eel_request_str;

/* Ensure initialized request structure ------------------------------ */
my_eel_request_str.command_enu      = EEL_CMD_UNDEFINED;
my_eel_request_str.identifier_u08   = 0;
my_eel_request_str.address_pu08     = NULL;
my_eel_request_str.status_enu       = EEL_ERR_PARAMETER;

...

/* It is assumed at this point, that the library has been initialized, opnened
and started up correctly. */

/* Initiate VREIFY comamnd ------------------------------------------- */
my_eel_request_str.command_enu      = EEL_CMD_VERIFY;

do
{
    EEL_Handler();
    EEL_Execute(&my_eel_request_str);
}
while (my_eel_request_str.status_enu == EEL_ERR_REJECTED);

while (my_eel_request_str.status_enu == EEL_BUSY)
{
    EEL_Handler();
}
```

```
/* Check if a refresh is required for full data retention before device is
powered down. */
my_do_refresh = false;
if (my_eel_request_str.status_enu == EEL_ERR_VERIFY)
{
    my_do_refresh = true;
}
else if (my_eel_request_str.status_enu != EEL_OK)
{
    MyErrorHandler();
}

if (my_do_refresh)
{
    /* Initiate REFRESH comamnd ---------------------------------------- */
    my_eel_request_str.command_enu      = EEL_CMD_REFRESH;
    do
    {
        EEL_Handler();
        EEL_Execute(&my_eel_request_str);
    }
    while (my_eel_request_str.status_enu == EEL_ERR_REJECTED);

    while (my_eel_request_str.status_enu == EEL_BUSY)
    {
        EEL_Handler();
    }

    if (my_eel_request_str.status_enu != EEL_OK)
    {
        MyErrorHandler();
    }
}

/* Initiate SHUTDOWN command ---------------------------------------- */
my_eel_request_str.command_enu      = EEL_CMD_SHUTDOWN;

do
{
    EEL_Handler();
    EEL_Execute(&my_eel_request_str);
}
while (my_eel_request_str.status_enu == EEL_ERR_REJECTED);

while (my_eel_request_str.status_enu == EEL_BUSY)
{
    EEL_Handler();
}

if (my_eel_request_str.status_enu != EEL_OK)
{
    MyErrorHandler();
}

EEL_Close();
```

# Chapter 5 Library Setup and Usage

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

## 5.1 Obtaining the Library

The Tiny EEL is provided by means of an installer via the Renesas homepage at http://www.renesas.eu/updates?oc=EEPROM_EMULATION_RL78. Please follow the instructions of the installer carefully and read the user manual which is also available there. For operation, the Tiny EEL requires the corresponding FDL (RL78 FDL Type T02, Tiny FDL). Please ensure to always work on the latest version of both libraries.

## 5.2 File Structure

The Tiny EEL is delivered as precompiled library for Renesas, IAR and GNU environments. The library and its header files are stored in the *lib* subdirectory inside the installation folder. The *smp* directory contains sample setups which are no integral part of the library itself and should be modified according to the project needs. All files are listed in Table 18.

**Table 18: File structure of the Tiny EEL**

| File | Description |
|---|---|
| **<installation folder>** | |
| eel_info.txt | contains release-specific information about the installed library |
| **<installation folder>/lib** | |
| eel.h | EEL header file, EEL interface definition |
| eel_types.h | EEL header file, EEL types definition |
| eel.inc | EEL assembler include file with interface definition |
| eel_types.inc | EEL assembler include file with EEL types definition |
| eel.lib (REN)<br>eel.r87 (IAR)<br>eel.a (GNU) | precompiled library file |
| **<installation folder>/smp/C** | |
| eel_descriptor.c | user defined EEL-variable descriptor |
| eel_descriptor.h | EEL configuration part |
| eel_user_types.h | sample user types for EEL variables |
| eel_sample_linker_file.dr (REN)<br>eel_sample_linker_file.xcl (IAR)<br>eel_sample_linker_file.ld (GNU) | linker sample file |
| **<installation folder>/smp/asm** | |
| eel_descriptor.asm (REN) | user defined EEL-variable descriptor and sample user types |
| eel_descriptor.inc (REN) | EEL configuration part |
| eel_sample_linker_file.dr (REN) | linker sample file |

## 5.3 Library Resources

In the subsequent sections, the resources utilized by the library are described in detail.

### 5.3.1 Resource Consumption

The Tiny EEL is allocated together with the program in the user area, occupying an area equal to the size of the library. Furthermore, the library itself uses the CPU, the stack, internal data and data buffers (for read/write access to the EEL variables).

Detailed information about the required software resources is listed in Table 19. All resource consumption numbers are based on the Tiny EEL V1.00. Please note, that the Tiny EEL requires the Tiny FDL (Type T02) for operation. The resource consumption of the Tiny FDL can be found in the associated user manual. It is not possible to use the Tiny EEL with any other FDL than the Tiny FDL (Type T02), i.e. FDL and EEL have always to be used as a bundle (RL78 FDL T01 with RL78 EEL T01, and RL78 FDL T02 with RL78 EEL T02).

**Table 19: Resource consumption of the Tiny EEL V1.00**

| Resource | Renesas Compiler | IAR Compiler | GNU Compiler |
|---|---|---|---|
| maximum code size (code flash) | 2746 bytes | 2768 bytes | 2769 bytes |
| constants (code flash) | <EEL variable count> + 4 bytes | < EEL variable count > + 4 bytes | < EEL variable count > + 4 bytes |
| internal data (RAM, short address area) | 1 byte Note | 1 byte Note | 1 byte Note |
| maximum stack usage (including FDL) | 80 bytes | 80 bytes | 92 bytes |

**Note:**
Depending on the used device, the Tiny EEL may use a fraction of the user RAM as working area. Size and location of this area is strictly device dependent, see Section 5.3.3 for more details.

### 5.3.2 Linker Sections

The following sections are related to the Tiny EEL and need to be defined in the linker file (please see eel_sample_linker_file.dr or eel_sample_linker_file.xcl for an example):

- **FDL_CODE:** Segment containing the code of the Tiny FDL.
- **FDL_SDAT:** Data segment of the Tiny FDL containing all FDL-internal variables. Needs to be placed in the short address area of the RAM.
- **FDL_CNST:** Segment for Tiny FDL constants.
- **EEL_CODE:** Segment containing the code of the Tiny EEL.
- **EEL_SDAT:** Data segment of the Tiny EEL containing all EEL-internal variables. Needs to be placed in the short address area of the RAM.
- **EEL_CNST:** Segment containing EEL constants.

**Note 1:**
The FDL_CODE and FDL_CNST as well as the EEL_CODE and EEL_CNST segments can be located anywhere inside the code flash, but must be inside the same 64 kB page.

**Note 2:**
The FDL_SDAT and EEL_SDAT segments must be located in the short address area of the RAM.

**Note 3:**
Not all library functions are linked. Unused functions are not included in the linked executable.
For the Renesas assembler, linking can be reduced to a subset of the library functions by deleting unnecessary functions from the include file.

### 5.3.3 Prohibited RAM Area

The Tiny EEL may use a fraction of the user RAM as working area, referred as prohibited RAM area. The size and position of this area is strictly device dependent (many devices do not even have this area) and vary between the different RL78 products. For details, please refer to the document “User's Manual: Hardware” of your RL78 product.

If a prohibited RAM area is specified for the utilized device, it is not allowed to access this area while the Tiny EEL is active. Whenever EEL functions are called, the data in the prohibited area may be rewritten.

### 5.3.4 Stack and Data Buffer

The Tiny EEL has to handle some rather complex tasks in order to manage the EEL variables. As this management is run directly on the CPU, the library utilizes the same stack as specified in the user application. It is the developer’s duty to reserve enough free stack for the operation of both, user application and EEL.

The data buffer used by the Tiny EEL refers to the RAM area in which data is located that is to be written into the data flash and where data is to be copied to when an EEL variable is read. These buffers need 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.

**Caution:**
In contrast to the internal EEL data (EEL_SDAT segment), both stack and data buffer may not be allocated in the short address range from 0xFFE20 to 0xFFEFF—and also not in the prohibited RAM area, if it exists in the target device.

### 5.3.5 Register Usage

The Renesas and IAR releases of the Tiny EEL use the registers of the currently selected register bank. No implicit register bank switch is performed by the library.

For the GNU release of the Tiny EEL, it is mandatory that register bank 0 is active on function entry. No implicit register bank switch is performed by the library. Return values are placed in register bank 1. For details on GNU calling conventions, please refer to the GNU documentation for RL78 devices.

**Note:**
A detailed description of the registers used for parameter passing and return values can be found in Section 0 along with the detailed function descriptions.

### 5.3.6 Library Timings

In the following, certain timing characteristics of the Tiny EEL are specified. As the Tiny EEL accesses the Tiny FDL directly, the FDL timings affect also those of the EEL. All timing specifications are based on the following library versions:

- Tiny EEL: V1.00
- Tiny FDL: V1.00

Please note that there might be deviations from the specified timings in case you are using other library versions than the ones mentioned.

#### 5.3.6.1 Maximum Function Execution Times

The maximum function execution times are listed in Table 20. These timings can be seen as worst case durations of the specific EEL function calls and therefore can aid the developer for time critical considerations, e.g. when setting up the watchdog timer. Please note however, that the typical and minimum function execution times can be much shorter. Especially the EEL_Handler consumes only a few clock cycles when the EEL driver is in idle state.

Please note that the Tiny EEL function execution times are independent of the operation mode of the device (full speed or wide voltage). Also, these timings do not contain any frequency-independent part and are therefore specified in terms of numbers of cycles.

**Table 20: Maximum function execution times of the Tiny EEL V1.00 (using Tiny FDL V1.00)**

| Function name | Maximum function execution time [cycles] |
|---|---|
| EEL_Init | 708 + 40 per EEL variable |
| EEL_Open | 14 |
| EEL_Close | 17 |
| EEL_Execute | 320 |
| EEL_Handler | 4582 |
| EEL_GetDriverStatus | 245 |
| EEL_GetSpace | 47 |
| EEL_GetVersionString | 14 |

### 5.3.6.2 Typical Command Execution Times

The EEL command execution times can depend on several factors as for instance the system clock frequency, the device operation mode (full speed or wide voltage), the current pool constellation, the number and size of the utilized EEL variables and the period between the handler calls. The typical command execution time examples listed in Table 21 have been measured based on the following scenario:

- The EEL_Handler is called continuously after issuing the command via EEL_Execute until it is completed.
- The system clock frequency is 20 MHz.
- The device operates in full speed mode.
- All measurements have been performed with the same EEL descriptor (variable set) as it is used in the example descriptor provided with the library. Descriptor properties which have an influence on the execution time are mentioned in the column “Command-specific condition” in the table.
- All commands are finished successfully.
- Further command-specific conditions are listed in the table.

**Table 21: Typical command execution time examples of the Tiny EEL V1.00**

| Command | Command-specific condition | Typical command execution time [µs] |
|---|---|---|
| Format | 3 EEL blocks, all non-blank | 19100 |
| Startup | 3 EEL blocks, last block active and almost full, 8 EEL variables. | 1245 |
| Shutdown | - | 12 |
| Refresh | 3 EEL blocks, no excluded blocks, 8 variables (all written) with a total size of 305 bytes (example descriptor) | 41057 |
| Verify | The internal verify (see FDL) passes for the whole active block | 4073 |
| Write | variable size 2 bytes (type_A from example descriptor), variable has been written before | 760 |
| | variable size 4 bytes (type_C from example descriptor), variable has been written before | 865 |

| Command | Command-specific condition | Typical command execution time [µs] |
|---|---|---|
| | variable size 10 bytes (type_F from example descriptor), variable has been written before | 1179 |
| | variable size 255 bytes (type_Z from example descriptor), variable has been written before | 14028 |
| Read | variable size 2 bytes (type_A from example descriptor), variable has been written before | 38 |
| | variable size 4 bytes (type_C from example descriptor), variable has been written before | 39 |
| | variable size 10 bytes (type_F from example descriptor), variable has been written before | 43 |
| | variable size 255 bytes (type_Z from example descriptor), variable has been written before | 215 |

**Note:**
The timings listed in Table 21 are only example measurements and no specification. Deviations have to be expected based on the target setup and situation.

## 5.4 Library Setup

The Tiny EEL can be configured and adapted to a certain degree in order to fit the requirements of the user application. Most important to mention here is the possibility to configure an application-specific set of variables of different sizes. The details how to perform this setup are described in the following sections.

### 5.4.1 EEL Pool Configuration

As already described in Section 2.2, the Data Flash can be separated in an FDL and an EEL pool which coexist and can be used independently. The actual number of blocks assigned to EEL and FDL can be configured in the descriptor of the FDL. Please have a look at the user manual of the Tiny FDL (FDL type T02) for details.

Please note that the minimum number of EEL blocks required for operation is 2.

Increasing the number of available EEL blocks does not increase the capacity of the variable set that can be used with the Tiny EEL (see also Section 5.4.3). However, it makes sense to assign more than two blocks to the EEL pool. This way, hot spots on one block are reduced and wear leveling is improved. On the one hand, this means that the number of erases for one block is effectively reduced increasing the lifetime of the device. On the other hand, it enables a larger number of variable updates (writes) during the device life cycle.

Deriving the actual number of required EEL blocks for a certain scenario (application and required device lifetime) is a nontrivial task, which is directly related to the endurance of Data Flash and detailed in the following section.

In this context of flash endurance, please recall that whenever the number of non-excluded blocks falls below 2 (i.e. there is only one non-excluded block left), write access and refresh is not possible anymore. However, read access can still be performed (c.f. Section 2.3.1). A reanimation of the library can only be attempted by a format of the pool (see Section 4.3.4).

### 5.4.2 Endurance Calculation

Every write operation of a new EEL variable instance occupies space in the Data Flash. Whenever the active block is full, the current variable set needs to be copied to a new block by means of the refresh operation during which the target block is erased (see also Section 2.3.1).

This process is repeated many times over the device lifetime. However, the endurance of the Data Flash blocks regarding the number of erase cycles is limited. Hence, it is necessary to calculate the application-specific number of erase cycles required over the device lifetime and to ensure that the specified Data Flash endurance is not exceeded.

Renesas provides an endurance calculation sheet which can be filled with the different data sets sizes and the required write cycles. The sheet can estimate the expected number of Flash erase cycles and indicate an exceedance of the Data Flash specification. Thereby, the number of available EEL blocks has a major impact the overall endurance of the pool. By using varying numbers of EEL blocks for the calculation, the required number of EEL blocks can be derived for the target application scenario.

**Note:**
The endurance calculation sheet is a very helpful tool. However, please note that the result is only an estimate and the sheet cannot produce absolutely accurate numbers. The exact endurance depends on additional constraints—like for instance the write sequence of the variables—which are not captured by the sheet. Therefore, any result of the endurance calculation sheet must be confirmed in the actual user application.

The calculation sheet can be obtained from the Renesas Flash support, which can be contacted via Email at the following address: application_support.flash-eu@lm.renesas.com.

### 5.4.3 EEL Variable Configuration

The characteristics of the precompiled EEL can be configured by the user in two source files: eel_descriptor.h and eel_descriptor.c (eel_descriptor.inc and eel_descriptor.asm for assembler).

In the header file eel_descriptor.h, the developer has to set the number of EEL variables to be handled by the library by means of the constant EEL_VAR_NO. Any integer value from 1 to 64 is allowed. If this maximum number of variables is not sufficient in your case, please contact the Renesas support in order to find the proper solution for your application scenario at application_support.flash-eu@lm.renesas.com.

```
#define   EEL_VAR_NO                  8
```

The actual set of variables is defined in eel_descriptor.c by means of the constant array eel_descriptor. The first element of the array represents the number of defined variables. This element is followed by the size of each variable which may be between 1 and 255 bytes. Thereby the index within the array matches the corresponding variable ID. As a result the first EEL variable ID is always ‘1’. A user-specified ID as it is featured in the EEL Type T01 is not supported by the Tiny EEL. The variable list needs to be zero terminated in the last element.

```
__far const eel_u08 eel_descriptor[EEL_VAR_NO+2] =
{
  (eel_u08)(EEL_VAR_NO),      /* variable count   */  \
  (eel_u08)(sizeof(type_A)),  /* id=1             */  \
  (eel_u08)(sizeof(type_B)),  /* id=2             */  \
  (eel_u08)(sizeof(type_C)),  /* id=3             */  \
  (eel_u08)(sizeof(type_D)),  /* id=4             */  \
  (eel_u08)(sizeof(type_E)),  /* id=5             */  \
  (eel_u08)(sizeof(type_F)),  /* id=6             */  \
  (eel_u08)(sizeof(type_X)),  /* id=7             */  \
  (eel_u08)(sizeof(type_Z)),  /* id=8             */  \
  (eel_u08)(0x00),            /* zero terminator  */  \
};
```

Clearly, the overall consumption of the variable instances may not exceed the available space in the active block. Hence the defined variable set needs to fulfill the following relation in order to maintain operational:

$$2(N+1) + \sum s(i) + MAX(s(i)) \leq 1014 \quad ,$$

where N is the number of variables and s(i) is the size of variable i. However, it is strongly recommended not to exhaust the space as this leads to very frequent and costly refreshing of the pool (in the worst case after each variable write). As a rule of thumb, the variable set should fulfill the following relation:

$$2(N+1)+\sum s(i) \leq 507 \quad .$$

If the variable setup is exceeding this limit, it can be considered to move seldom changed data from the EEL to the FDL pool (where it has to be managed by the developer). Depending on the scenario, the EEL type T01 might also be an alternative as it can handle a larger amount of data. In case of any doubts which library suits your application scenario best, please contact the Renesas support at application_support.flash-eu@lm.renesas.com.

**Note:**
In case the variable setup is changed, it is imperative to format the EEL pool before continuing the operation on the pool (i.e. before startup).
The only exception to this rule is that appending variables to the existing list is allowed as this does not change the ID and size of the existing variables.

### 5.4.4 EEL Variable Initialization

Before being able to regularly use the Data Flash for EEPROM emulation, the EEL pool must be formatted and—depending on the application—be filled with initial values for the EEL variables. This can be done using different approaches. Two very common ways are presented in the following:

- The application itself executes the format operation and then writes initial instances of the variables. As the format operation deletes all data, it needs to be carefully considered in this scenario how to prevent accidental pool formatting by the application.
- A serial programming tool (e.g. PG-FP5) or debugger is used to program the Data Flash in the same flow that also programs the Code Flash.

For the later approach (using a programming tool or a debugger), a hex file is required which contains the Data Flash content (i.e. the complete EEL pool in raw format). This content can be gained by

- dumping the content of the Data Flash with an already formatted EEL pool into a hex file using a serial programming tool or the debugger, or by
- using a tool chain of Data Flash Converter and/or Data Flash Editor to convert EEL variable values into a hex file.

Data Flash Converter and Data Flash Editor can be obtained from the following URLs (including dedicated user manuals):

- http://www.renesas.eu/updates?oc=DATAFLASHCONVERTER.
- http://www.renesas.eu/updates?oc=DATAFLASHEDITOR

Please note that the Tiny EEL is supported only by the Data Flash Converter since version V6.00 and by Data Flash Editor versions V4.00 or higher.

### 5.4.5 Distributing Data between FDL and EEL

As already described in Section 2.2, the Data Flash can be separated in an FDL and an EEL pool which coexist and can be used independently. As a consequence, the developer has the freedom to choose for each set of data whether to manage it himself in the FDL pool or to assign it to an EEL variable.

In general, data which is constant over the lifetime of the device (or is updated very seldom), is placed best in the FDL pool. This enables early access after power up (before the EEL is started up). However, the developer has to take care himself of data management and reset robustness. The overall size of the utilizable FDL pool is thereby manly limited by the number of blocks in the Data Flash.

Data which is updated frequently is usually managed best in the EEL pool. A variable-oriented update of data is provided in a way that wear-leveling and reset-robustness considered. However, the utilizable space is limited (only one active block).

### 5.4.6 Building Efficient EEL Variable Sets

Whenever defining variables to be used with the EEL, the following two main characteristics should be considered:

- A variable instance always creates additional administrative usage of the active block by means of a reference entry of two bytes. As a consequence, many small variables lead to a reduction of space utilizable for data.
- When updating a variable (write), the whole data of the variable needs to be written in the empty space of the active block. Hence, it is inefficient to assemble several small variables which are updated independently into a larger single variable.

Therefore, variables which are typically updated concurrently (like for instance data and checksum) should be assembled in a single EEL variable.

## 5.5 Practical Aspects of Library Usage

The following sections collect assorted aspects of practical usage of the Tiny EEL.

### 5.5.1 When to use EEL_Handler and FDL_Handler

One of the most typical usage scenarios for the Tiny EEL is that also the Tiny FDL is used with its own assigned flash pool. Both libraries provide a handler to drive pending commands and update the status of the corresponding request structure. On top of that, the EEL_Handler might even execute the FDL_Handler, when the EEL has issued an FDL command on its own and is waiting for the result. However, it is not guaranteed that every EEL_Handler call also results in an FDL_Handler call. Therefore, it is not sufficient to only call the EEL_Handler in a mixed FDL/EEL application.

Nevertheless, both FDL and EEL are robust against calls of the opposite handler which could only lead to a completion of the currently running command. Therefore, the application designer should stick to a very simple rule:

> When issuing a command from a scope, always call the handler corresponding to the issued command from that scope in order to finish the command.

Thereby, it can be guaranteed that commands do finish, although EEL commands can be interrupted by FDL commands and EEL commands drive running FDL commands by means of the EEL_Handler.

### 5.5.2 EEL_Handler Calls

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

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

- **Calling EEL_Handler repeatedly after issuing a command execution:** This approach is also utilized in most of the code examples you can find in this manual. Typically realized in a loop waiting for the operation status not to be busy anymore, this approach results in the best EEL operation performance. However, the CPU is fully loaded and blocked for other tasks as long as the EEL command is being executed. This approach can also be seen as using the library with a blocking function-oriented interface.
- **Calling EEL_Handler in a timed task:** By calling the EEL_Handler periodically, EEL commands can be driven forward while other tasks are processed by the CPU. The period between the handler calls can have significant impact on the EEL operation performance. Shorter calling intervals result in better EEL performance, but also increase the CPU load by the EEL. Due to this tradeoff, a general advice for the calling interval cannot be given. It needs to be analyzed and tailored individually for each target application.

  However, selecting a calling interval shorter than the typical command execution time for a single-byte FDL write command (see the documentation of RL78 FDL T02) does not improve the EEL performance significantly and should be avoided.

- **Calling EEL_Handler in the idle task:** If it is ensured that the idle task is called often enough, this method might result in a good EEL performance, as the handler can be called continuously. However, this approach is not deterministic in case of a high CPU load by the application itself.

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

**Note 1:**
When evaluating concepts for calling the EEL_Handler, please be aware that all FDL/EEL functions are not re-entrant. That means it is not allowed to call FDL/EEL functions from interrupt service routines while any FDL/EEL function is already running.

**Note 2:**
In contrast to the RL78 EEL T01, the Tiny EEL does not have any background processing for pool maintenance. Hence, it is only mandatory to call the EEL_Handler as long as there is an unfinished EEL command. It is not necessary to call the EEL_Handler regularly for pool maintenance. However, in case there is nothing to process, the EEL_Handler consumes a few clock cycles only and does no harm.

### 5.5.3 When to issue a Refresh

In general, there are two strategies a developer can follow with respect to the ever reoccurring EEL refresh:

- **Refresh on error during write:** Whenever a write command results in an EEL_ERR_POOL_FULL error, issue a refresh and restart the write command. Thereby, the EEL pool is used in an efficient way with respect to keeping the number of erase cycles low. The drawback of this approach is the fact that writing an EEL variable can require a time consuming flash block erase which can be a problem in time-critical scenarios.
- **Pre-emptive refresh based on free-space analysis:** By means of the function EEL_GetSpace, the developer can regularly monitor the current free space in the active block and issue the refresh when the library space is getting short and there is sufficient processing time for it (including the erase of the new active block). Thereby, it can be guaranteed that a write does not result in a time-consuming refresh (i.e. block erase). The drawbacks of this approach are on the one hand an increased administrative effort in the user application, and on the other hand a more frequent occurrence of block erasures which can result in a reduction of the overall lifetime of the device.

The developer should analyze his application carefully to choose one of these two methods. Even a mixture of both approaches in a single application could be considered.

### 5.5.4 Using the Standby and Abort Feature of the Tiny FDL

The Tiny FDL offers a standby/wakeup feature in order to temporarily turn off the data flash (e.g. for energy savings). This mechanism is transparent for the usage of the EEL, meaning that it is allowed to even interrupt running EEL commands temporarily by calling FDL_StandBy.

Please note however, that setting the FDL into standby will prevent also EEL command execution. However, new commands will not be rejected but simply not completed in this case. This means that it is not possible to check through the EEL API whether the FDL has been sent to standby. The developer has to use FDL functions in order to do this.

After the execution of FDL_WakeUp, the EEL will continue the pending command executions with the next EEL_Handler call.

**Note:**
Please read the Tiny FDL user manual for a more detailed description of the standby/wakeup feature and its usage.

Similarly, the abort feature of the Tiny FDL can be used transparently when the EEL is used. In case the EEL has issued an FDL erase command (e.g. during refresh), it is not forbidden to abort this command via FDL_Abort. In that case the EEL will resume the erase the next time the EEL_Handler is called. Considering also the reset robustness of the Tiny EEL, it is explicitly allowed to use FDL_Abort in emergency situations (e.g. in case of a voltage drop) in order to ensure a fast saving of important data.

### 5.5.5 Using the Tiny EEL in Operating Systems

As many time-consuming flash operations are performed in the background and only their progress is only checked by means of a request-response mechanism, the Tiny FDL and EEL can be efficiently used inside an operating system.

**Note:**
Please read the cautions for the EEL described in Chapter 6 and also the cautions chapter of the Tiny FDL manual carefully before utilizing the Tiny FDL and Tiny EEL in an operating system.

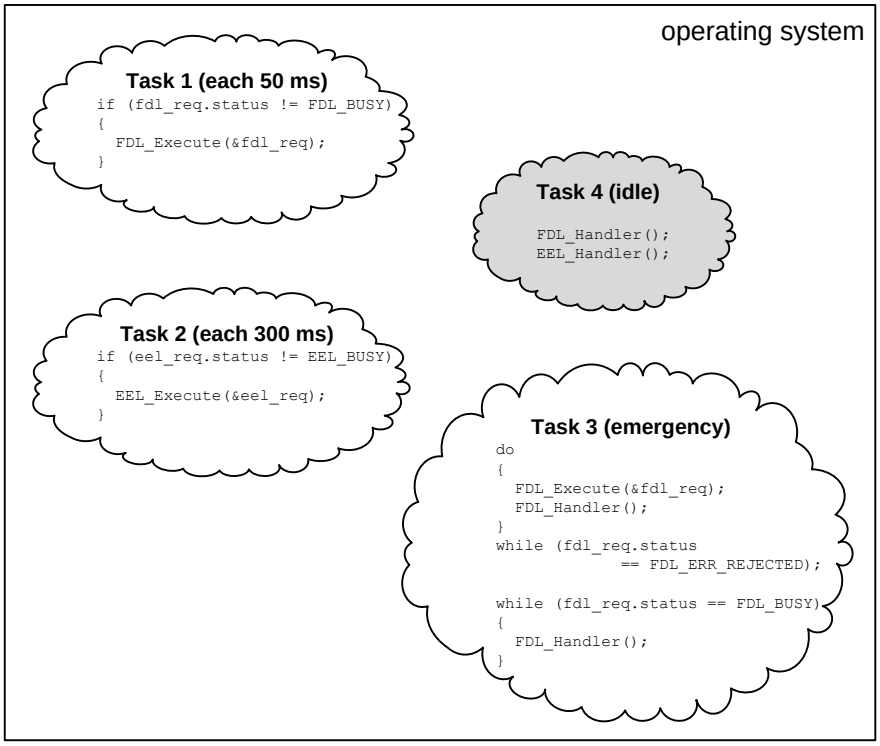


**Figure 14: Schematic illustration of Tiny EEL integration into an operating system**

Figure 14 illustrates a schematic example how to integrate the Tiny EEL into an operating system. Please note that this is only an abstract example to show the principle of the mechanism.

Three different types of tasks have to be distinguished:

- **Requesting tasks:** Requesting tasks issue any FDL or EEL command via FDL_Execute and EEL_Execute, respectively, assuming that that the command execution will be finished by the IDLE task. In the example, task 1 issues an FDL command, while an EEL command is started from task 2.
- **IDLE task:** The IDLE task (task 4 in the example) is used to finish any running FDL or EEL command by means of FDL_Handler and EEL_Handler calls. (It is assumed here that the idle task is executed whenever no other task is running.) Please note that it is necessary to call both handlers form such a function in order to ensure the completion of both types of commands.
- **Emergency task:** This task is triggered in an emergency situation as for instance in case of a voltage drop in order to save important data before the device is powered off. In the shown example, task 3 is the emergency task. In such a task it is important to complete the issued command as fast as possible by repeated handler calls. Please note that the example uses the FDL for writing emergency data. It is recommended to do so as the processing time for a write command is less compared to the EEL and contingent and time-consuming refreshs can be avoided.

**Note:**
For emergency tasks, it can be a good idea to take advantage of the abort feature of the Tiny FDL in order to avoid lengthy erase processes in such a situation. Please see Section 5.5.4 and the Tiny FDL user manual for details.

### 5.5.6 Reset Robustness Considerations

In general, the sequences of EEL commands modifying the EEL pool have been designed in a way that incomplete command executions are detected during the next startup and one valid set of EEL data can always be restored.

Hence, during normal operation, there will be no loss of EEL data due to reset. Nevertheless, please be aware that depending on the progress of the interrupted last command, the old or the new variable set will be active after restarting the library.

There is one exception to this rule: The format command is *not* robust against resets. In case a reset occurs during a format, there is a chance that the EEL pool is coincidentally valid and will be accepted during the next startup. In order to minimize this risk, all blocks are invalidated as a first step during a pool format. However, please ensure to always finish a once started format command.

Please note that this limitation is not severe in the typical use case as a format command destroys all EEL variable content and is therefore usually executed once at the beginning of a product lifetime.

### 5.5.7 Pitfalls with Request Variables declared in a local Scope

The request-response oriented command execution of the Tiny EEL enables a comfortable way of issuing different commands quasi independently from different scopes of the user application. The developer should thereby always be aware that the request data structure is updated later by the library via handler calls. Therefore, it is imperative to ensure the existence of the data structure as long as the command is being processed.

This is of special importance when using local C request variables which are placed in the stack. When the application leaves the scope of such a local variable, the reserved space on the stack is deallocated and probably reused for other variables or return addresses. When an EEL command is finished in such a situation, this new data is overwritten, leading to unpredictable behavior of the application. Such situations are often hard to detect and debug. Hence, the developer should always be pay special attention to the scope of the used request variables.

# Chapter 6 Cautions

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

- The library code and constants must be located completely in the same 64KB flash page.
    - For REN compiler, the library takes care in the code to define these sections with UNIT64KP relocation attribute.
    - For IAR compiler, the user has to ensure that the linker file specifies the Flash page size equal to 64KB when defining FDL_CODE, FDL_CNST, EEL_CODE and EEL_CNST sections.
    - For GNU compiler, the user shall take care that FDL_CODE, FDL_CNST, EEL_CODE and EEL_CNST sections are not mapped across any boundary of 64KB Flash page.
- The FDL library initialization by means of FDL_Init must be performed before the execution of FDL_Open, FDL_Close, FDL_Handler, FDL_Execute, FDL_Abort, FDL_StandBy, FDL_WakeUp and all EEL functions.
- The EEL library initialization by means of EEL_Init must be performed before the execution of EEL_Open, EEL_Close, EEL_Execute, EEL_Handler, EEL_GetDriverStatus, EEL_GetSpace.
- It is not allowed to read the data flash directly (meaning without FDL) during a command execution of the FDL/EEL.
- Do not execute the STOP or HALT instruction during the execution of the Tiny FDL/EEL without driving the library into standby first.
- The watchdog timer does not stop during the execution of the Tiny FDL/EEL.
- Each request variable must be located at an even address.
- Before executing any command, all members of the request variable must be initialized once. If there are any unused members in the request variable, please set arbitrary values to these members. Otherwise, a RAM parity error may cause a reset of the device. For details, please refer to the document “User's Manual: Hardware” of your RL78 product.
- All functions are not re-entrant. That means it is not allowed to call FDL/EEL functions from interrupt service routines while any FDL/EEL function is already running.
- Task switches, context changes and synchronization between FDL functions:
    - All FDL/EEL functions depend on FDL/EEL global available information and are able to modify this information. In order to avoid synchronization problems, it is necessary that at any time only one FDL/EEL function is executed. So, it is not allowed to start an FDL/EEL function, then switch to another task context and execute another FDL/EEL function while the other one is not yet finished.
    - Example for a forbidden sequence:
        1. Task 1: Start an FDL/EEL operation with FDL_Execute/EEL_Execute.
        2. Interrupt the execution and switch to task 2, executing FDL_Handler/EEL_Handler.
        3. Return to task 1 and finish the FDL_Execute/EEL_Execute function.
- After the execution of FDL_Close/EEL_Close, all requested/running commands will be aborted and cannot be resumed. The user has to take care that all running commands are finished before calling FDL_Close/EEL_Close.
- It is not possible to modify the Data Flash in parallel to a modification of the Code Flash.
- The internal high-speed oscillator must be started before using the FDL/EEL.
- Do not locate any function arguments, data buffers or stack to the short address area from 0xFFE20 to 0xFFEFF.

- When the Data Transfer Controller (DTC) is used during control of data flash, do not locate the RAM area for the DTC to the short address area from 0xFFE20 to 0xFFEFF.
- Do not use the RAM area used by the Tiny FDL/EEL (including the prohibited RAM area) before both libraries have been closed.
- The reset robustness of each written EEL instance is inherently provided by the Tiny ELL. However, the data itself is not secured directly by any checksum. In case the user application destroys the data buffer during the write process or by any other accident, the EEL will not be able to detect this and will return wrong data values when reading this instance. If your scenario requires such type of data protection, please assemble each eel instance from data and checksum and perform consistency checks after each read access manually.
- In case of an accidental write to the eel pool (e.g. due to stack runaway, misuse of FDL etc.) the following scenarios may occur:
  - In case of damaged block flags the whole eel pool could become inconsistent. As a result, all valid instances could be lost.
  - In case of damaged instance entry
    - all or only a part of instances could become invalid.
    - old instances could become valid.
- It is not allowed to continue the execution of the EEL in case the FDL or EEL descriptor has been changed. In such a situation, the initialization of the FDL and EEL shall be performed as well as the EEL pool format via the EEL_CMD_FORMAT command.

# Revision History

| Chapter | Page | Description |
|---|---|---|
| All | | Rev. 1.00:<br>Initial document |
| 4.1.6<br>5.4.1<br>5.4.2<br>5.4.4<br>5.5.2<br>All | 31<br>70<br>70<br>72<br>73 | Rev. 1.01:<br>Section added describing the variable eel_descriptor<br>Paragraph about flash endurance considerations added<br>Section for endurance calculation added<br>Section added describing EEL variable initialization<br>Section on EEL_Handler calls added<br>Corrected typos, updating links for download page |
| All | | Rev 1.02:<br>Document type changed from Application Note to User Manual |
| 4.1<br>4.2<br>5.2<br>5.3.1<br>5.3.5<br>6 | 27<br>34<br>68<br>69<br>70<br>80 | Rev. 1.03:<br>Added type description notes for GNU<br>Added description for GNU API<br>Extend file structure for GNU<br>Added resource consumption for GNU<br>Added register bank usage for GNU<br>Explanation added how library can be mapped in the same 64K flash page for each compiler |

EEPROM Emulation Library

Renesas Electronics Corporation

R01US0070ED0103