Application Note

16

EEPROM Emulation Library

EEL – T01

16 Bit Single-chip Microcontroller
RL78 Series

Renesas Electronics
www.renesas.com

R01AN0707ED0100
June 17, 2011

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. 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.
8. 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.
9. Although Renesas Electronics endeavors 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.
10. 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.

11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. 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.

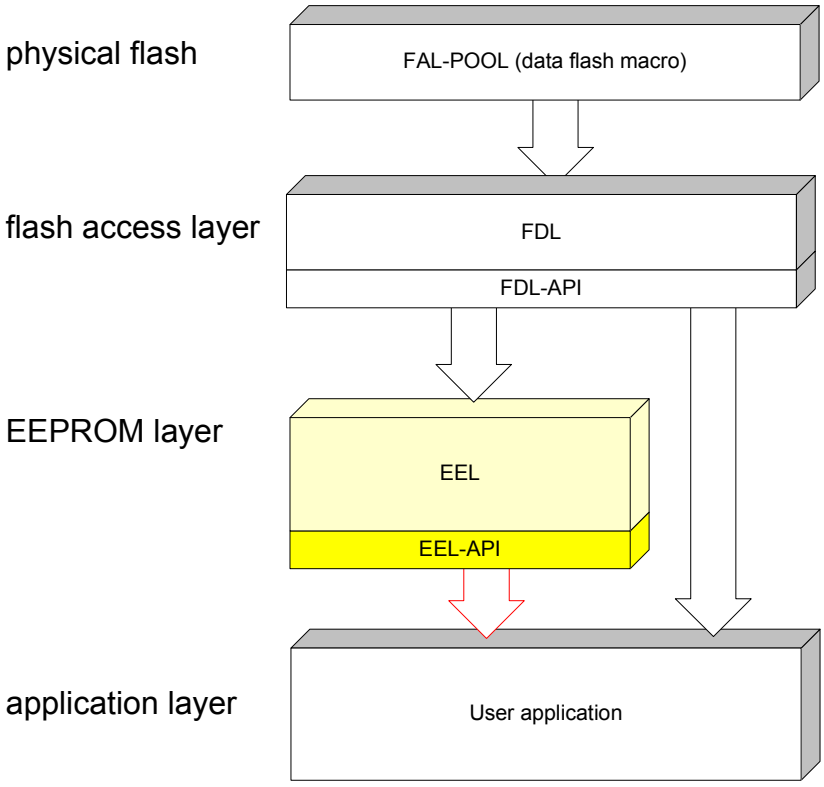
Table of Contents

Chapter 1 Introduction

This application note describes the internal structure, the functionality and the software interface (API) of Renesas RL78 EEPROM Emulation Library (EEL) Type 01, designed for RL78 flash devices with so called Data Flash based on the MF3 flash technology.

The EEL is the highest layer of Renesas EEPROM Emulation System which aspires to mime at least the functionality of an non-volatile memory (internal EEPROM) under usage of the on-chip embedded flash memory. Beyond that divers service and administrative functionality is provided by the EEL to simplify the handling at application side.

Figure 1-1 Elements of the EEPROM Emulation System



Note:
This application note describes the functional block marked in yellow

1.1 Naming convention

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

Table 1 Used abbreviations and acronyms

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

1.2 Related documents

Table 2 List of related documents

Document Number	Description
R01US0034EDxxxx	Data Flash Access Library

1.3 MF3 Data Flash

Almost all devices of the RL78 microcontroller family are equipped with a separate flash area called Data Flash.

1.3.1 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 parallel to the application software without significant on its execution timing.

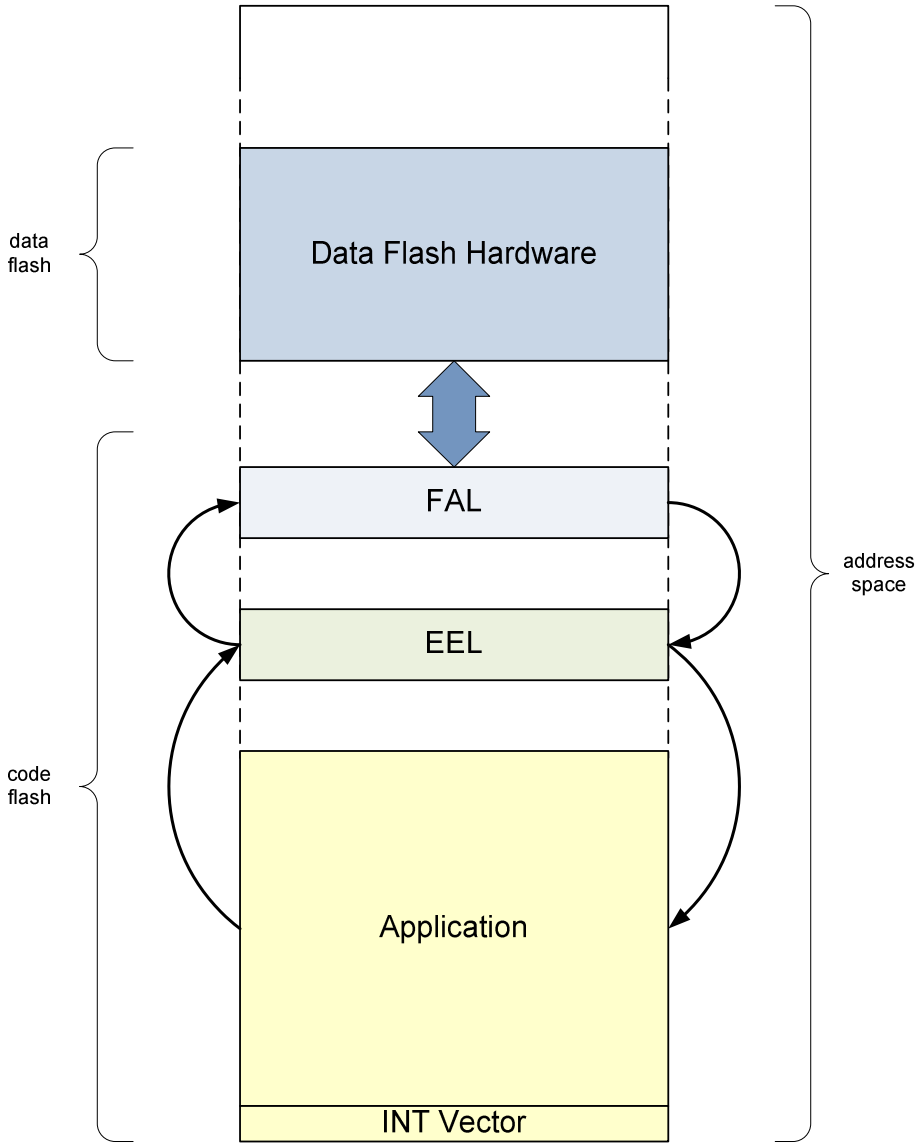
If not mentioned otherwise in the device users manuals, RL78 device with Data Flash are designed according to this standard approach.

Note:
It is not possible to modify Code Flash and Data Flash in parallel.

1.4 Functional elements within the EEPROM Emulation system

Even though this user's manual describes the functional block "EEPROM Emulation Library" a short description of all concerned functional blocks and their relationship could be beneficial for the general understanding of the system. The following figure illustrates the basic idea behind and its involved functional blocks but the shown dependencies are not complete.

Figure 1-2 Relationship between functional blocks inside the EEPROM emulation systems



Application:

The functional block "Application" contains the instruction code of user's software using the EEL.

EEPROM Emulation Library (EEL):

The functional block “EEPROM Emulation library” is the subject of this user’s manual. It offers all functions and commands the “Application” can use in order to handle its EEPROM data.

Data Flash Access Library (FAL):

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

Note:
General requirement is to be able to deliver pre-compiled EEL libraries, which can be linked to either Data Flash Access Libraries (FDL) or Code Flash Access Libraries (FCL). To support this, a unique API towards the EEL must be provided by these libraries. Following that, the standard API prefix FDL_... which would usually be provided by the FDL library, will be replaced by a standard Flash Access Layer prefix FAL_... All functions, type definitions, enumerations etc. will be prefixed by FAL_ or fal_. Independent from the API, the module names will be prefixed with FLD_ in order to distinguish the source/object modules for Code and Data Flash.

1.5 Pool structure

The EEL-pool is a part of the FDL-pool defined by the user in the file FAL_descriptor.h. in that file the user can divide the FDL-pool into two independent parts: the EEL-pool (used exclusively by the EEL only) and the USER-pool which can be freely used by the application to store any data.

To protect the content of the EEL-pool against unwanted user accesses the EEL-driver is using only hidden subroutines reserved exclusively for the EEL.

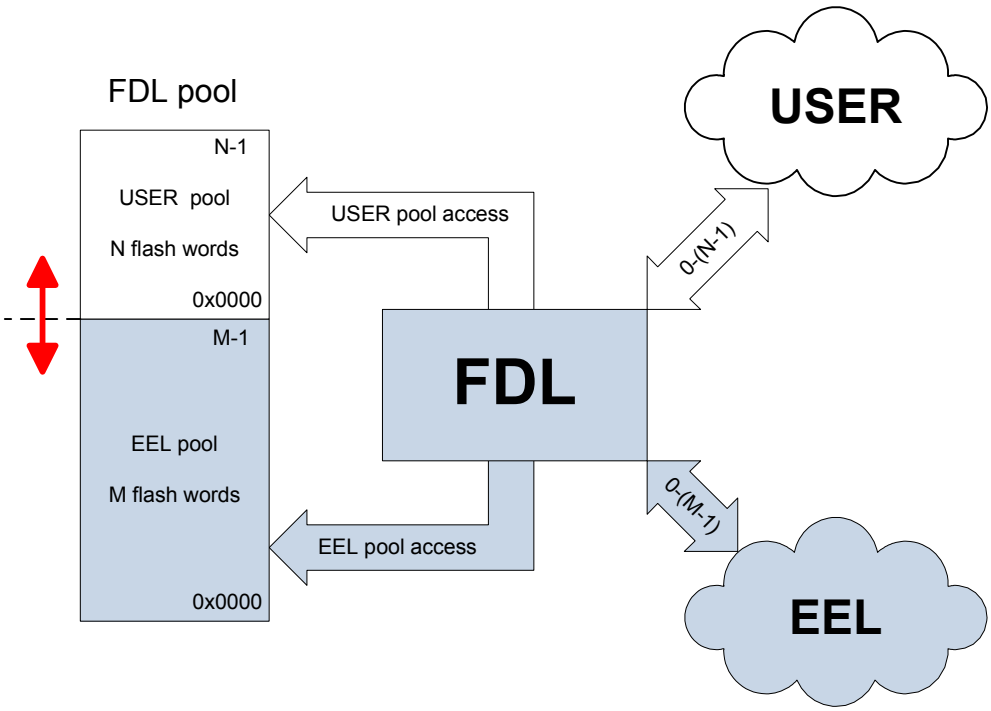
Pool details:

- **FDL-pool** allocates the physical Data Flash memory that can be handled by the FDL. It is a kind of container reserving room for the EEL-pool and USER-pool. All characteristics (valid address information, partitioning information, ...) of the FDL-pool are defined in the FDL-pool descriptor. Based on that information the FDL protect all flash content against illegal access.
- **EEL-pool** is a virtual pool inside the FDL-pool used exclusively by the EEL for storing data and control information.
- **User Pool** is completely in the hands of the user application. It can be used to build up an own user EEPROM emulation or to simply store constants.

Note:

Please refer to the FDL user’s manual for further details.

Figure 1-3 Pool access scheme, general scheme



1.6 Address virtualisation

To simplify the flash content handling as well the parameter passing between the FAL and the EEL the physical addresses used by the flash hardware were transformed into a linear 16-bit index addressing flash-words (32-bit units) inside the corresponding pool. By this measure each owner of the pool can use it as a simple array of words. To address the array elements (read/write access) word-index starting at 0x0000 can be used. The max. range of the word-index depends on the FAL-pool configuration and the number of flash blocks reserved for the particularly pool. This kind of address virtualization allows to access max 2 * 256kBytes Data Flash and offers an effective access rights management.

Note:
The user of the EEL is not touched by the above address virtualization.

Chapter 2 EEL architecture

This chapter describes the internal architecture of the EEPROM Emulation Library.

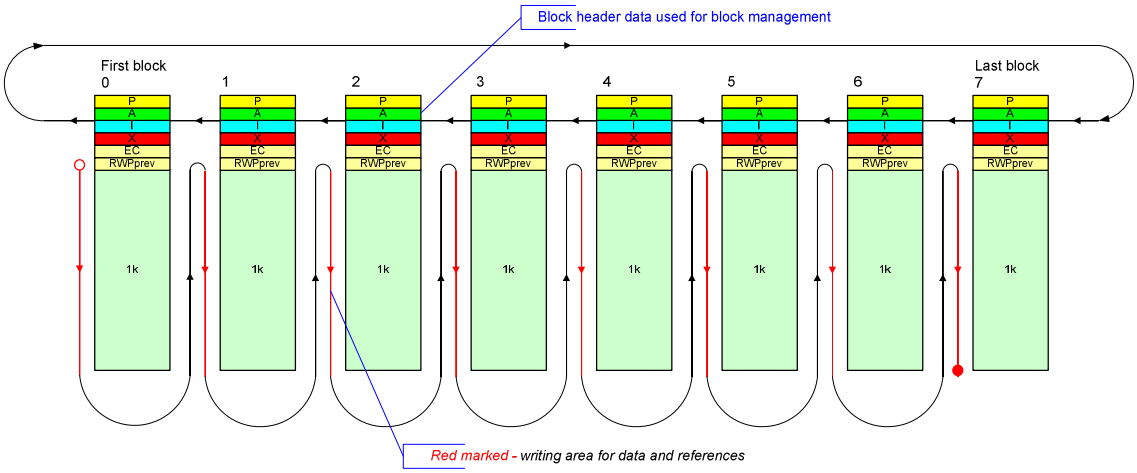
2.1 EEL pool structure

The EEL pool is the virtual storage medium used by the EEL driver for storing data and block management information during its operation. From logical point of view the EEL-pool is organized as a single-linked ring of blocks.

“Single-linked ring” means here:

a) the next block to block N is block (N+1)
b) the next block to the last one is the first one.

Figure 2-1 Structure of an empty EEL pool (no data inside)



Each block of the EEL-pool contains a block-header for storing block management information. Because the block indexing within the EEL-pool is based on the homogenous and fixed virtual block numbers 0x0000.... (EEL_POOL_SIZE - 1) it is not necessary to store the neighbors inside the block header.

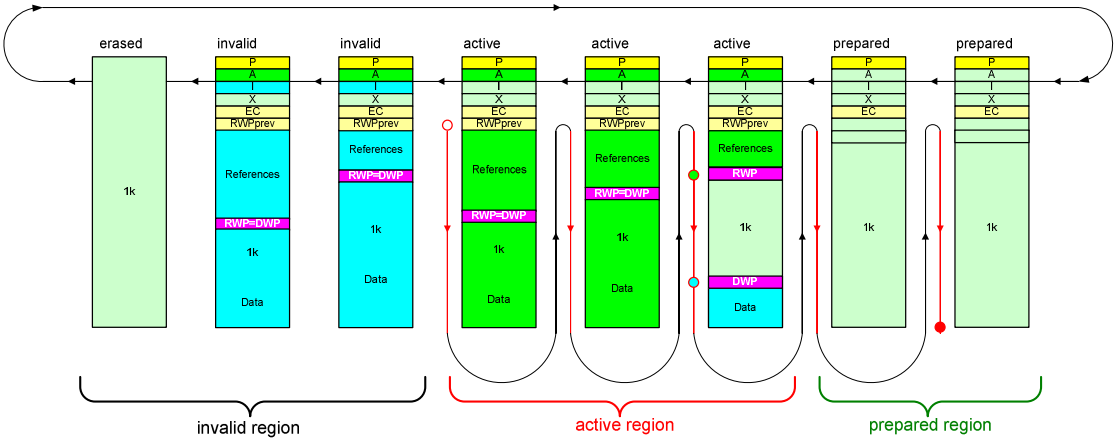
All flash-blocks of the EEL pool are grouped in three consecutive “regions” indicated by the “block status” in the block header.

“active region”	- consists of blocks containing active data
“invalid region”	- consists of blocks without active data
“prepared region”	- consists only of blocks ready to receive new data

When contemplate EEL-pool blocks clockwise the regions are always in the same fixed chronological order:

“prepared region” is before “active region”
“active region” is before “invalid region”
“invalid region” is before “prepared region”

Figure 2-2 EEL pool regions during normal operation



Block organization scheme based illustrated above offers following advantages:

a) two symmetrical sections (where always 50% of Data Flash does not contain valid data) are not needed anymore

b) the “active region” can grow and be adapted to the momentary need

c) the reference area is separated from the data inside the same EEL block

d) copy-processes are mostly much faster because reduced to the only last active block has to be released from valid instances.

e) exclude functionality does not reduce performance of the driver

2.2 EEL block structure

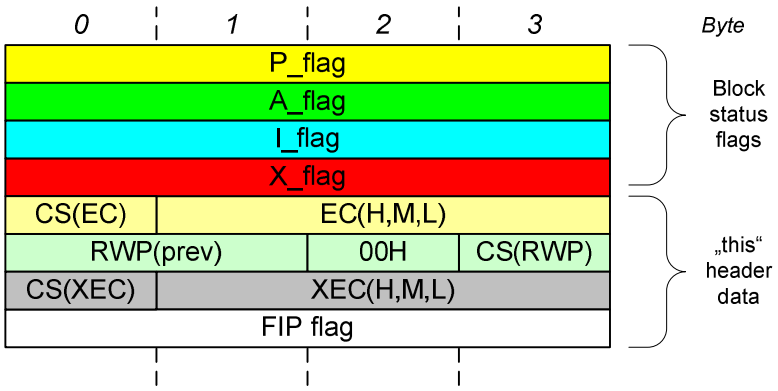
Each EEL block belonging to the EEL-pool is basically divided into three areas: the block header, reference area and the data area. The block-header contains information about the actual status of the block which is needed for the block-management within the pool. The reference area contains reference entities off all instances written into this block during its live-cycle. It is necessary for actual data localization after power-on. The data area contains the pure data belonging to the corresponding references in reference area.

Figure 2-3 EEL block, general structure

2.2.1 EEL block header

The block header is a small area on 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 the same in all blocks of the EEL-pool.

Figure 2-4 EEL block header structure



2.2.1.1 EEL block status flags

Each flag within the block header consists of one flash word (4 bytes).

There are two types of block status flags:
- “constructive status flag” used in processes like “activation” and “preparation”
- “destructive status flags” used in processes like “invalidation” and “exclusion”

When reading the exact pattern 0x55555555 a “constructive” flag is TRUE
When reading a pattern other than 0xFFFFFFFF a “destructive” flag is TRUE

When setting “constructive” flag: 0x55555555 is written into the flag-word.
When setting “destructive” flag: 0x00000000 is written into the flag-word.

P_flag: = 0x55555555 marks a “prepared” block that waits for data.
A_flag: = 0x55555555 marks an “active” block that may contain data
I_flag: ≠ 0xFFFFFFFF marks an “invalid” block (without valid data)
X_flag: ≠ 0xFFFFFFFF marks a block “excluded” from block management.

2.2.1.2 EEL block erase counter

The block header word four contains the block erase counter. Its consistency is protected by an 8 bit checksum which is used by the EEL internally only.

2.2.1.3 EEL previous reference write pointer

Its points the last RWP position of the previous block within the EEL pool.
It is used by the EEL internally only.

2.2.1.4 EEL exclusion erase counter

Stores the EC value at exclusion time.
It is used by the EEL internally only.

2.2.1.5 EEL Format In Progress (FIP) indicator

FIP<>0xFFFFFFFF indicates an FORMAT command discontinued by RESET. It marks the completely EEL pool as inconsistent and enforces the user to re-start the FORMAT command.

2.2.2 Reference area

The “reference area” is located in each EEL block directly behind the block header. It consists of so called reference entries that are used for instance identification, localization and for safeguarding during the read/write process. When writing new data into the EEL a corresponding reference entry is stacked in the reference area.

The reference area is growing upstairs from lower widx to higher.

2.2.3 Data area

The “data area” consists of data-records and is located on the bottom of each EEL pool block. Each data record within the data-area consists of pure data information without any data- frame. The data-frame information exists completely in the corresponding reference-entry in the reference-area.

When writing new data into the EEL the data area is growing downstairs from higher widx to lower.

2.3 EEL Instance structure

EEL instance is a complete data-set consisting of three components:

- 32-bit data reference pointer DRP in the reference area
- the data in the data are
- 32-bit checksum in reference area (directly behind the corresponding DRP)

Whenever the application writes a new value into the EEL pool a new EEL instance is generated.

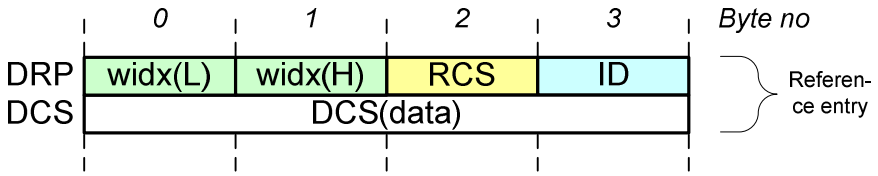
2.3.1 Data Reference Pointer, DRP

The main purpose of the DRP is referencing the data belonging to the given instance inside the data-area. The consistency of the DRP is safeguarded by an own 8-bit checksum. A DRP is always written to an even flash word index inside the reference area.

The structure of each DRP consists of:

ID:	8-bit EEL-variable identifier registered in the EEL descriptor.
widx:	16-bit virtual index inside EEL pool pointing to the data
RCS:	Reference Check Sum, 8-bit checksum across the DRP.

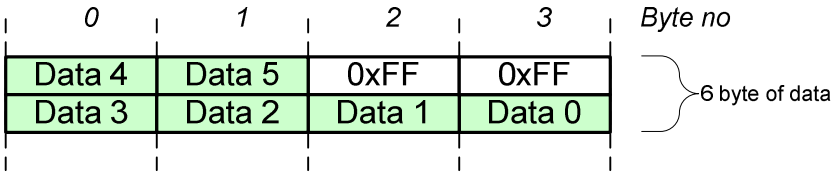
Figure 2-5 Structure of the DRP



2.3.2 Instance data

The pure instance data without any frame-information stored directly in the data area at the bottom of the corresponding block.

Figure 2-6 Example of 6-byte data entry

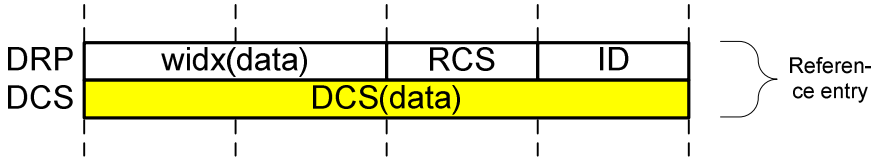


Note:
Not used bytes remain 0xFF.

2.3.3 Data Checksum, DCS

The DCS is written behind the DRP in the reference area behind the corresponding DRP after the instance data were written correctly. It ensures the plausibility of the data and the corresponding DRP.

Figure 2-7 Data Checksum of an instance



2.4 Block management

This chapter describes how the block management organizes the blocks inside the EEL pool during its operation.

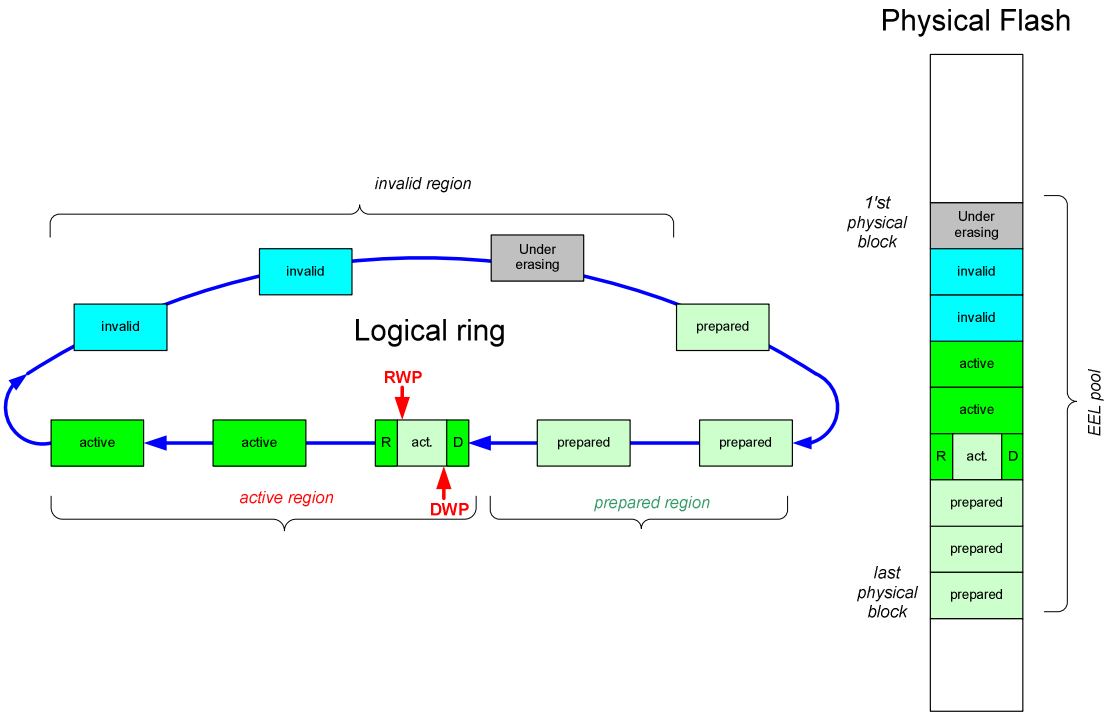
2.4.1 EEL block circulation

The block management is leaned on the concept of single linked ring. It is build based on the unique virtual block numbers inside the EEL pool. It is an easy scheme for “creation” and “consumption” of writeable space inside the EEL-pool. As already mentioned the whole EEL pool is divided into three regions organized in a fixed order.

Active region:	always in front of the invalid region
Prepared region:	always in front of the active region
Invalid region :	always in front of the prepared region

From the operation point of view the block management works like a caterpillar. The following figure should illustrate the idea behind the block management:

Figure 2-8 Circulatory block management inside the EEL pool



2.4.2 EEL block status

During the operation of the EEPROM driver the participating flash blocks change their internal status cyclically. To mark and to recognize the status of each block 32-bit block-status flags are used. The block status-flags are read and analyzed after power-on RESET to reconstruct the current EEL pool configuration. The block management based on that information is fundamental for correct operation of the EEL driver.

2.4.3 Security aspects, block exclusion

When erasing a flash block in the “preparation” process an erase-error could happen theoretically. The probability is very low but if happens, it is not allowed to write data into such a block. To fulfill this condition the “exclusion” mechanism was added to the block management

Basically during block preparation write-error can be generated when writing block header information. In that case the effected block will be excluded from block management too.

An asynchronous device RESET during operation of the EEL may cause various problems like inconsistent pool or inconsistent data. The STARTUP command detects such problems and performs fitting countermeasure to recover pool and data consistency

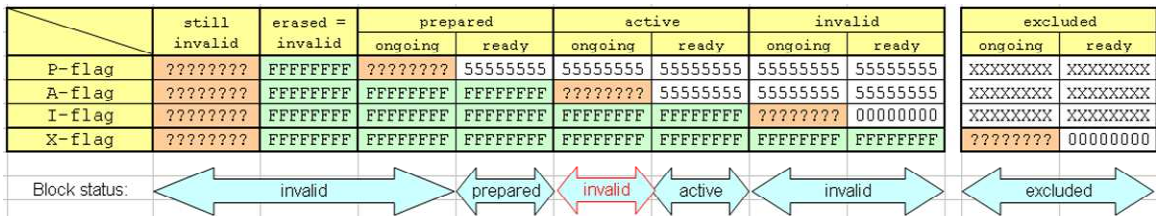
As already mentioned, there are two different types of block status flags:

1) Constructive block status flags are the P-Flag and the A-Flag.
   Coding: writing pattern 0x55555555 into the flag flash-word.
   Decoding: TRUE when read pattern is 0x55555555 otherwise FALSE.

2) Destructive block status flags are the I-Flag and the X-Flag.
   Coding: writing pattern 0x00000000 into the flag flash-word.
   Decoding: FALSE when pattern inside is 0xFFFFFFFF otherwise TRUE.

Analyzing the block header flags the EEL is in the position to recognize the status of each block of the EEL pool. Following scenarios are possible:

Figure 2-9 Block status code



Note:
Invalid block status can be produced by RESET during block activation (red marked here) is repaired in the STARTUP command sequence.

2.5 Instance management

Whenever a new instance of an EEL variable is written into the EEL-pool, the following sequence is executed by the EEL-driver internally:

Step 1)

Data-Reference-Pointer (DRP) is calculated and written into the flash word referenced by RWP. After that the space for instance data is allocated in the data area of the active region.

Step 2)

Write the complete instance data word by word into the reserved in step 1)

Step 3)

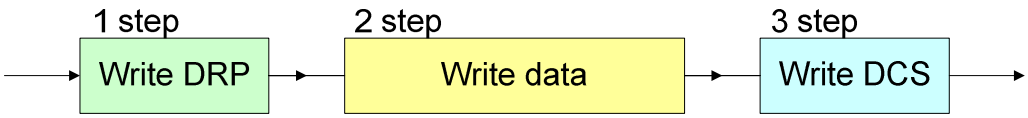
Calculate and write the checksum DCS into the word next to DRP from step 1)

2.5.1 Write instance sequence

Whenever a new instance of an EEL variable is written into the EEPROM the following sequence is executed by the EEL-driver:

Writing a new instance of an EEPROM variable consists of three successive phases.

Figure 2-10 Write instance sequence



The structure and the handling of the instance references should manage possible destructive effects caused by asynchronous power-on RESET as well as by potential flash problems.

2.5.2 Security aspects, checksums

When writing a new value of EEPROM variable into EEL the reference and the data are written flash-word wise into the EEL-pool. During this process an asynchronous RESET may happen at any time and produce rubbish data. To ensure a reliable detection of any data inconsistency within a written instance two stage checksum protection has been implemented. The first checksum (8 bit) ensures the consistency of the DRP written in phase 1). This checksum is a part of the 32-bit DRP. The second checksum is calculated and written in phase 3. It is a 32-bit checksum calculated across all data written in phase 1) and 2) (over DRP and all data words).

The consistency of the instance is checked in the STARTUP and in the READ command.

- when STARTUP command detects checksum error during instance searching (RAM reference fill process) the corresponding instance will be ignored.

- when READ command detects a checksum error the instance search will be restarted (same criteria as for STARTUP), the RAM reference table refilled and the newest instance with correct checksum will be read finally.

2.6 Processes

All things happening in the EEL (data access, CPU processing, administrative activities….) take time. Sequences of actions, measures and countermeasures to achieve any targeted effect/result are called processes here.

There are two groups of EEL processes:

Foreground process:
Initiated by the user, when requesting commands at the EEL.

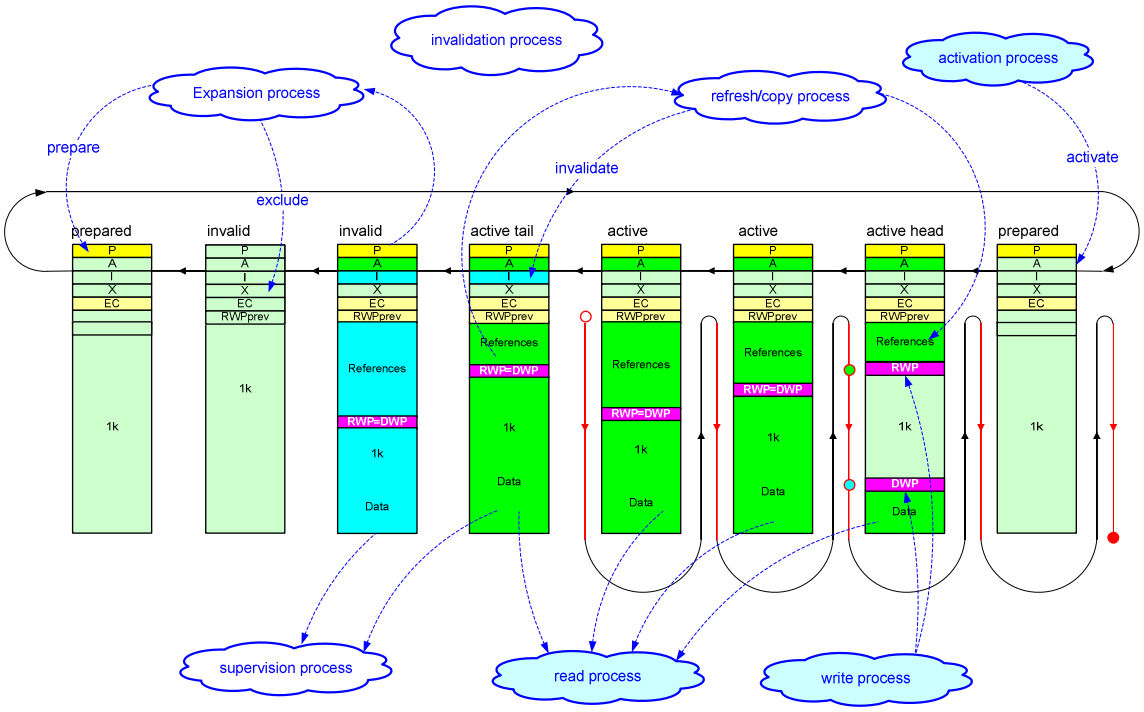
Background process:
Initiated by the EEL themselves, when it recognizes the necessity internally.

In exceptional cases foreground processes can initiate background processes.

From block management point of view each block is sorted into one of the three regions within the EEL pool (active, prepared, invalid) or it can be excluded. A block can change from one region to another one when being treated by dedicated “processes”.

Also the instance management influences the position of the instances within the EEL pool using background and/or foreground processes.

Figure 2-11 Overview of the main processes inside the EEL driver



2.7 Space treatment

Space within the EEL pool is the sum of all flash words prepared for the accommodation of data and references (exclusive block header area).

Internally the EEL driver differentiates between pool-space and active-space.

Pool-space is the space available in all prepared blocks plus the remaining space available in the active heading block.

Active-space is the space available in active heading block only.

Both can be effected by background and foreground processes as follows:

Pool-space is produced in the background PREPARATION process only.

Pool-space is consumed by foreground WRITE command or background REFRESH process.

Active-space is consumed by foreground WRITE command or background REFRESH process.

Active-space is enlarged by foreground or background ACTIVATION process.

The user does not need to take care for the space management during EEL operation. Depending on the configuration and used operation mode the EEL takes care internally for adequate space conditions.

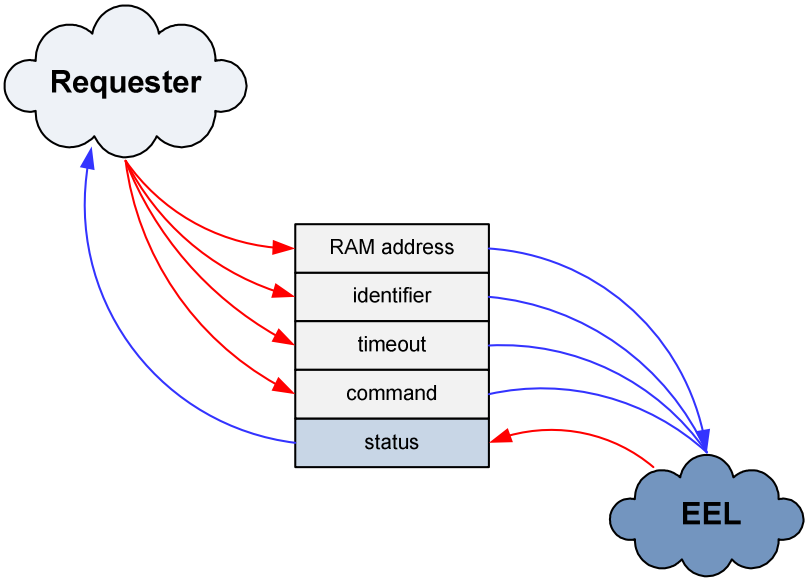
2.8 Request–Response oriented dialog

Like the FAL, the EEL is also using the Request-Response architecture to place and process the commands. This means the “requester” (normally users Application) has to fill-up a kind of “request form sheet” (the request variable) and pass it to the EEL using the reference (pointer) of the request variable for further processing. The EEL is interpreting the request variable, check its plausibility and process it for the time slice defined in the request variable. After time-out period or after finishing the execution with positive/negative command execution the EEL is updating the status code in the request variable.

The biggest advantage of the request-response architecture is the constant and narrow parameter interface. It allows constant parameter passing independent used compiler and its memory models.

Another advantage is the possibility to isolate the dialog in multi-tasking systems.

Figure 2-12 Schematic usage of the request variable



2.9 Handler oriented command execution

To satisfy operation in concurrent or distributed systems the command execution is divided generally into two phases:

1) Initiation of command execution using EEL_Execute(&my_eel_request)

2) processing of the command that is performed piece-wise (state-wise or time-slice-wise depending on the used execution mode)

The main advantage of such architecture is that maintenance and command processing can be done centrally on one place in the target system (normally the idle-loop or the scheduler loop).

The other advantage is that commands can be requested in several places in the system. Using separate request variables the EEL feedback can be directed correctly in spite of the fact, that the processing is done centrally.

The EEL is using the function EEL_Execute(&my_eel_request) for command initiation and EEL_Handler(my_eel_timeslice) for command processing.

2.10 Execution modes of the EEL

One claim of this EEPROM driver is to satisfy all the various systems and SW architectures exit in the market. Some target systems does not care about execution time and use EEL-commands like function call. Some other systems use complex operating systems to manage task execution quasi simultaneously (time sharing). Another use even driven asynchronous mechanisms only.

To fulfill the above requirements, the EEL offers several operation modes that can deal with the parameter “time” in different way.

There are two places where the “time” parameter can be treated :

a) in the request-variable passed by the EEL_Execute(&my_eel_request)

This timeout value determines the operation mode of the EEL command.

my_eel_request.timeout_u08 = 0x00 -> execution in polling mode
0x00 < my_eel_request.timeout_u08 < 0xFF -> execution in timeout mode
my_eel_request.timeout_u08 = 0xFF -> execution in enforced mode

b) by the timeout parameter of the EEL_Handler(my_eel_timeslice_u08)

my_eel_timeslice_u08 = 0x00 -> execute the actual EEL state only
my_eel_timeslice_u08 > 0x00 -> execute the time-slice EEL

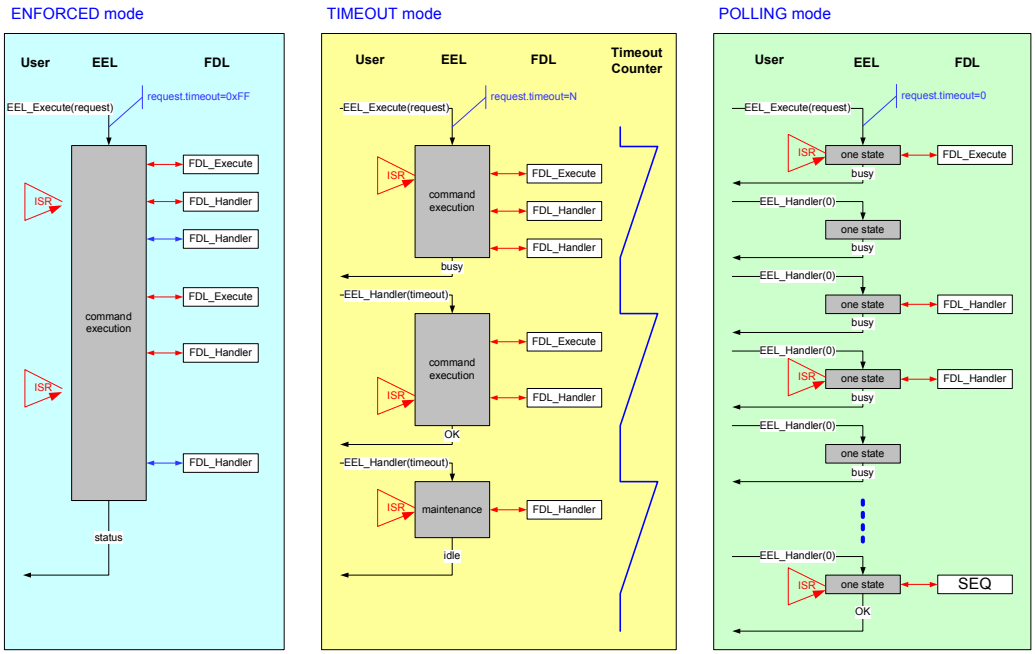
Table 3 Overview of time parameter meaning

Timeout	Execution mode	EEL behaviour
0x00	polling	**EEL_Execute(t):** starts the command and leaves EEL immediately **EEL_Handler(t):** executes next internal state of the EEL
0x00<N<0xFF	timeout	**EEL_Execute(t):** executes states until timeout or command is finished **EEL_Handler(t):** executes states until timeout or command is finished
0xFF	enforcing	**EEL_Execute(t):** executes command until it's finished **EEL_Handler(t):** executes states until command or timeout is finished

Depending on the target system architecture one of the operation modes can be used for command execution and background maintenance purpose.

Note:
The timeout used in the request variable is completely independent on the timeout used in the EEL_Handler(t) mixing of the operation modes in one target system is possible.

Figure 2-13 Overview over the EEL operation modes

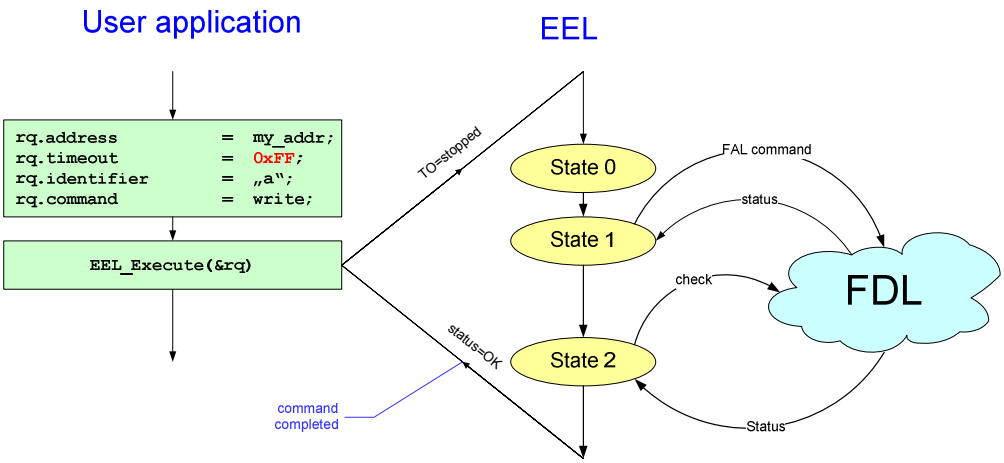


2.10.1 Enforced execution mode

This mode can be used in simple systems in that EEPROM access have to be processed like a simple function CALL. The requested command is directly and completely executed with positive or negative result. The handling is very easy, the background process that takes care for maintenance is not visible to the user.

Command execution in enforced mode is determined by timeout =0xFF in the request variable. When using enforced mode for command execution, the target system can use the EEL_Handler(t) for background maintenance (space generation) but it is not mandatory.

Figure 2-14 Schematic illustration of the enforced operation mode

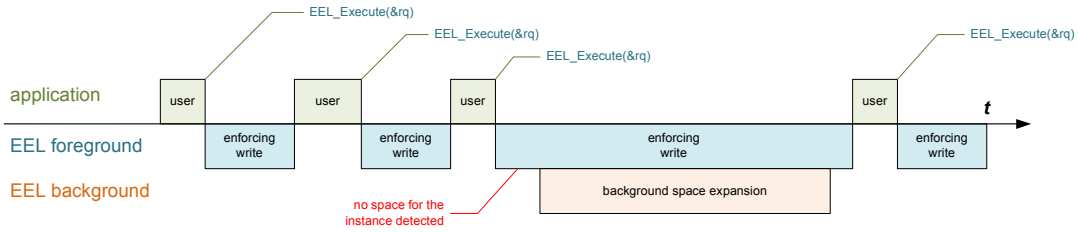


2.10.1.1 Enforced operation mode without usage of EEL_Handler(t)

The available space (inside the active and prepared regions) for accommodation of variable instances is limited. When executing commands in enforced mode without EEL_Handler(t) the available space decreases continuously during writing as long as the space becomes consumed. In that case new space must be generated internally inside the EEL before starting the command execution. This means that the execution time of “space consuming” commands (the WRITE command) cannot be constant. On the other hand the user does not need to take care for background maintenance.

When pure enforced mode is used in the target system the EEL_TimeOut_CountDown() function as well the EEL_Handler(t) are mandatory.

Figure 2-15 Timing example of enforced command execution without EEL_Handler(t)



Example conditions:

rq.address_pu08	- no meaning for the timing
rq.identifier_u08	- always same identifier used
rq.timeout_u08	- always 0xFF used
rq.command_enu	- always EEL_CMD_WRITE command used

2.10.1.2 Enforced mode with background maintenance

To enjoy the simplicity of the enforced execution mode without the disadvantage of not pre-determinable execution time the application can use EEL_Handler(t) to prepare space in advance in convenient phases.

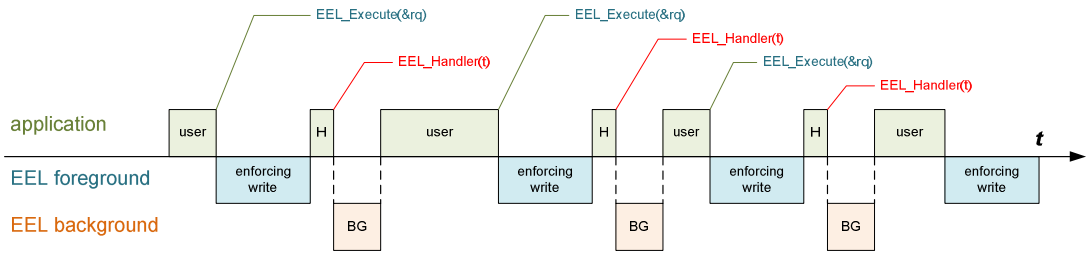
Calling EEL_Handler(t) cyclically at idle time (no EEL command under execution) the application activates the EEL background supervision and maintenance process. The background supervision checks if the momentary EEL-pool status does still correspond with the EEL-pool configuration. If not enough space detected by the background supervision, the background maintenance starts space production process autonomous. This is the instrument the application can use to produce enough space in advance and to guaranty fast and constant write execution time at any time.

Note:
The foreground writing and background maintenance are dynamical processes that influence each other. To ensure constant execution time of the WRITE command the application must provide enough CPU time to the background process. The relationship between “production of space” in the background and “consumption of space” by foreground writing must match.

The degree of “space production” is only determined by the CPU time offered to the background process via EEL_Handler(t).

The degree of “space consumption” is determined by the frequency and size of variables written into the EEL-pool, as well by the space needed for refreshing variables in background maintenance.

Figure 2-16 Timing example of enforced command execution without EEL_Handler(t)



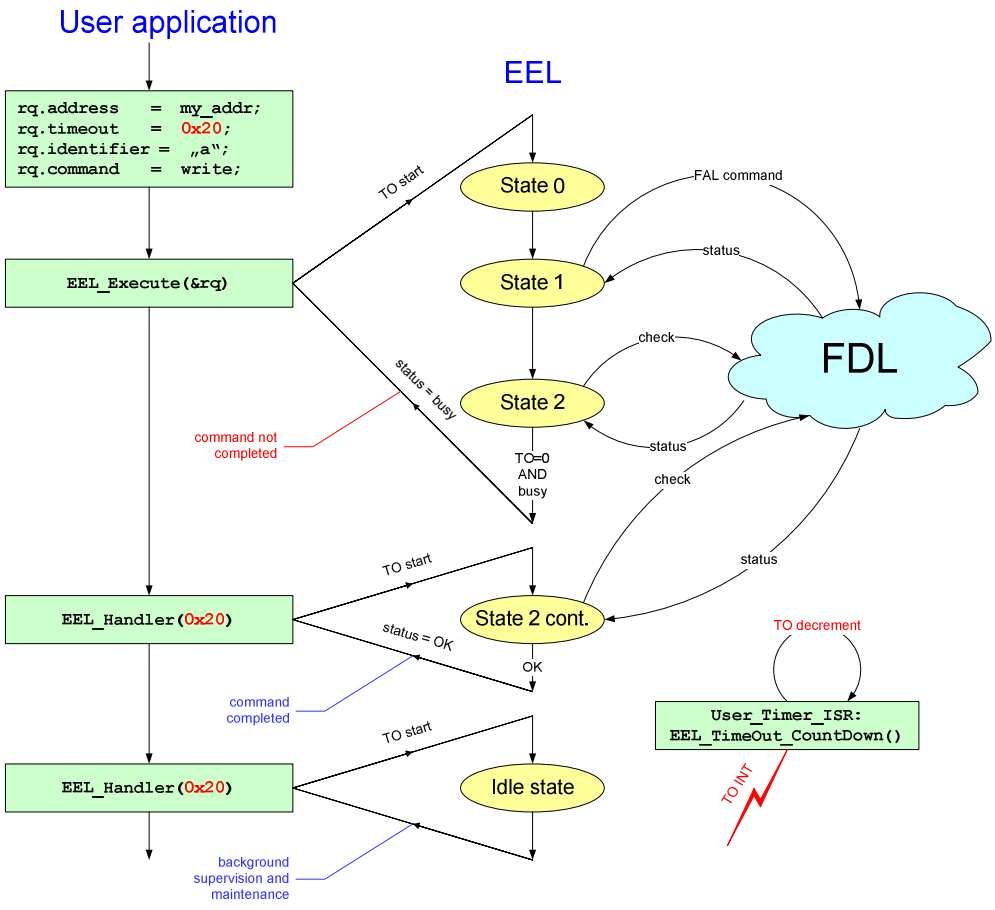
Example conditions:

rq.address_pu08- no meaning for the timing
rq.identifier_u08 - always same identifier used
rq.timeout_u08 - always 0xFF used
rq.command_enu - always EEL_CMD_WRITE command used
time - used by EEL_Handler(t) for time-slice definition

2.10.2 Timeout execution mode

In the timeout execution mode the requester can determine the CPU time for the command execution in advance. The resolution of the time period is defined freely by the user when choosing the counting interrupt source. The timeout period is defined in counting ticks. If the timeout period is longer than the real command execution time, the command is executed in the same wise as in enforced mode. If the timeout period is shorter than the command execution time the EEL_Execute(&my_eel_request) function will return with request-status “busy”. The remaining command will be continued time-slice-wise by the EEL_Handler(t). The timeout mode is intended to be used in synchronous time-slice based systems where each task allocates a fix interval of CPU time for its activity.

Figure 2-17 Schematic illustration of the timeout operation mode



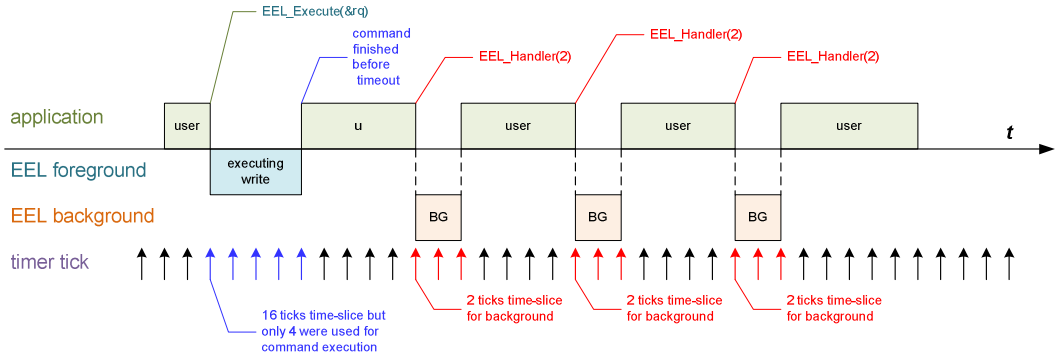
2.10.2.1 Command execution finished before timeout

When the timeout period specified in the request variable is longer than the real time needed by the EEL for command execution, the EEL_Execute(&my_eel_request) is left immediately after command completion. The EEL does not consume the remaining time during command execution. The reason is, that application normally writes variables asynchronously and wants to write as fast as possible.

Example conditions:

rq.address_pu08	- no meaning for the timing
rq.identifier_u08	- small EEL variable (i.e. 5 bytes)
rq.timeout_u08	- long timeout (16 timer ticks)
rq.command_enu	- always EEL_CMD_WRITE command used
timeslice	- 0x02 used here by EEL_Handler(t) for time-slice

Figure 2-18 Command execution completed before timeout



Note:
Black arrows symbolizes non-counting timer ticks (timeout counter is counted down to 0x00).

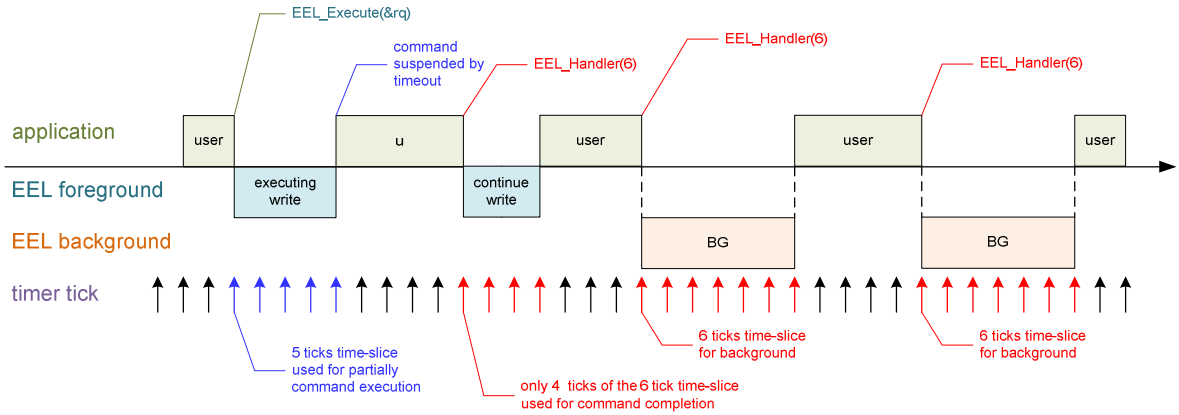
2.10.2.2 Timeout before command execution finished

When the timeout period specified in the request variable is shorter than the real time needed by the EEL for command execution, the EEL_Execute(&my_eel_request) is suspended with status=BUSY. The uncompleted command must be continued by using the EEL_Handler(t) function. When the remaining command is completed before time-slice is passed, the EEL_Handler(t) will be terminated immediately. The status inside the request variable changes from busy to finished. EEL does not consume the remaining time of the time-slice when command is finished. The reason is, that application normally writes asynchronously and want to write as fast as possible.

Example conditions:

rq.address_pu08	- no meaning for the timing
rq.identifier_u08	- larger EEL variable (i.e. 125 bytes)
rq.timeout_u08	- execution timeout (5 timer ticks)
rq.command_enu	- always EEL_CMD_WRITE command used
timeslice	- 6 ticks, used here by EEL_Handler(t) for time-slice

Figure 2-19 Command execution completed in EEL_Handler(t)



Note:
The 1'st EEL_Handler(t) call continues the command execution. If the command is finished in that time-slice, the EEL_Handler(t) will return immediately before timeout is elapsed.

The next EEL_Handler(t) calls are managing the BG processes according to the internal status of the EEL-pool:

- when no maintenance *) is necessary, supervision is running for full 6 ticks
- when any background process (REFRESH/PREPARATION) was interrupted by a write command, it will be continued in EEL_Handler(t) after write completion

*) maintenance means refresh or space expansion

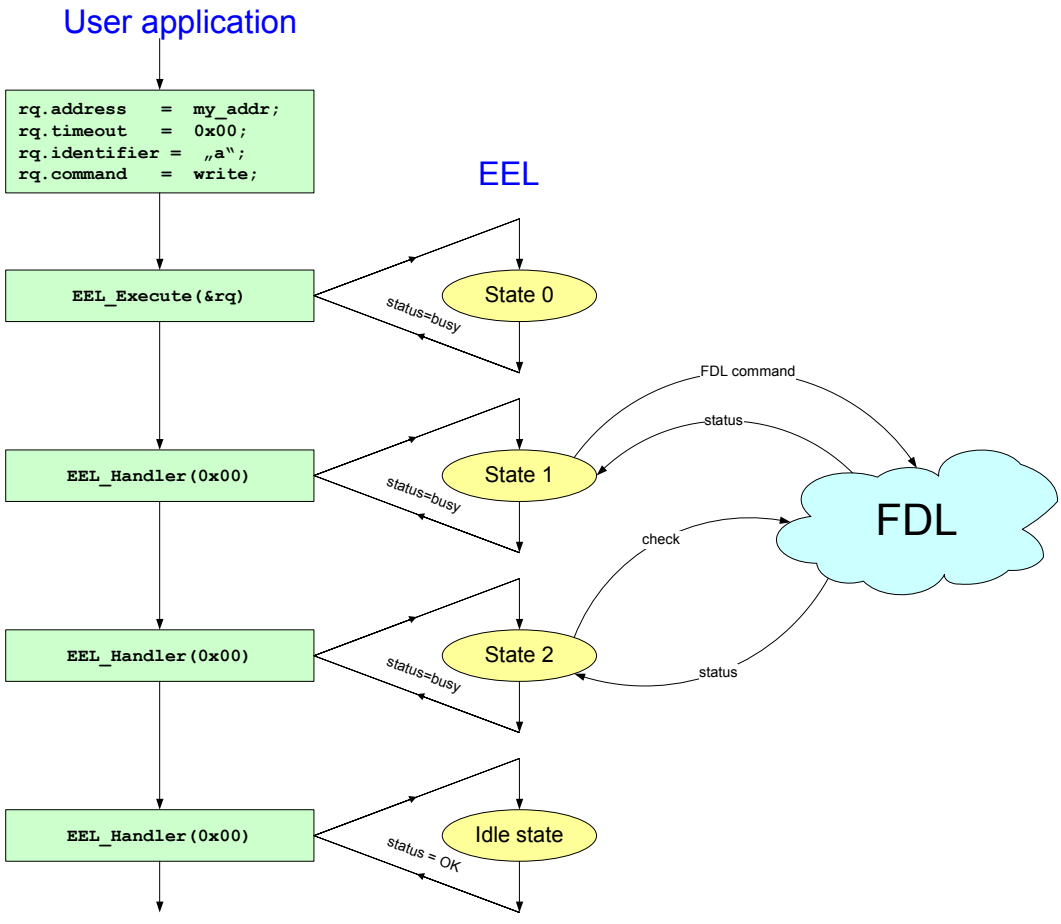
2.10.3 Polling execution mode

In the polling execution mode the function EEL_Execute(&my_eel_request) is just initiating the command execution and returns with the request-status “busy” after execution of the first internal state. The further command execution is performed in the EEL_Handler(t) that can operate with its own timeout period. If calling of EEL_Handler(0), the command execution or background maintenance will be executed state by state. In this operation mode the interaction frequency between the application and the EEL is the highest (fastest reaction). It is intended to be used in asynchronous systems where blocking of the CPU by any process must be minimized.

Note:

When pure polling mode is used in the system EEL_TimeOut_CountDown() function becomes mandatory.

Figure 2-20 Schematic illustration of the polling operation mode



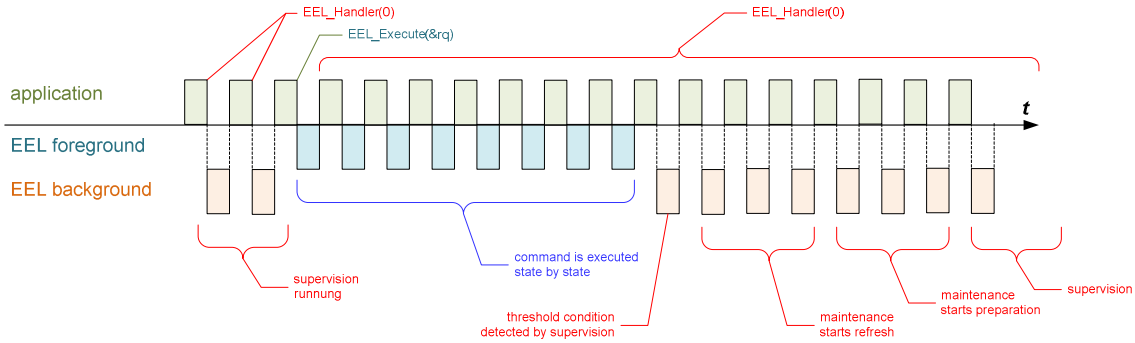
2.10.3.1 Full polling execution mode

The timeout parameter in the request variable as well the handler time-slice value are 0x00. The EEL commands, the supervision and maintenance process are executed very smooth, state by state.

Example conditions:

rq.address_pu08	- no meaning for the timing
rq.identifier_u08	- EEL variable
rq.timeout_u08	- 0x00, polling mode
rq.command_enu	- always EEL_CMD_WRITE command used
time-slice	- 0x00, no time-slice for the handler

Figure 2-21 Timing example of pure polling operation



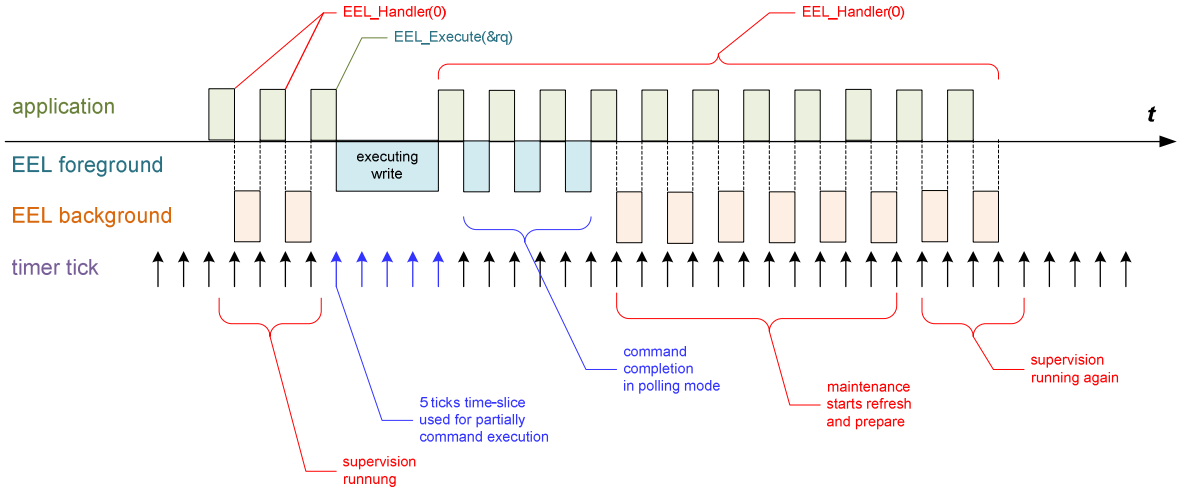
2.10.3.2 Mixed execution mode (timeout execution and polling maintenance)

The timeout parameter in the request variable as well the handler time-slice value are 0x00. The EEL commands, the supervision and maintenance process are executed very smooth, state by state.

Example conditions:

rq.address_pu08	- no meaning for the timing
rq.identifier_u08	- EEL variable
rq.timeout_u08	- 0x04, timeout execution
rq.command_enu	- always EEL_CMD_WRITE command used
time-slice	- 0x00, no time-slice for the handler (maintenance)

Figure 2-22 Timing in mixed operation mode (timeout and polling)



2.11 Supported command spectrum

There are two groups of commands supported by the EEL:
a) pool related commands influencing the whole pool status and structure.
b) variable related commands that control the access to the EEL data

Table 4 Command groups of the EEL

Command group	normal operation	exceptional operation
Pool related commands	EEL_CMD_STARTUP	EEL_CMD_CLEANUP
	EEL_CMD_SHUTDOWN	EEL_CMD_FORMAT
Variable related commands	EEL_CMD_READ	-
	EEL_CMD_WRITE	-

Note:
Refer to chapter “Operation” for command execution details

2.12 EEL execution planes

The EEL operates in so called two planes: background plane and foreground plane that dedicated to different purposes. The background plane is intended to perform maintenance and supervision work. The foreground plane is used exclusively to perform asynchronous commands requested by the user. Some of the commands require processes already implemented in the background plane. In such cases the foreground is able to activate background processes by swapping the activity focus into the background to perform necessary maintenance measures.

2.12.1 Foreground plane

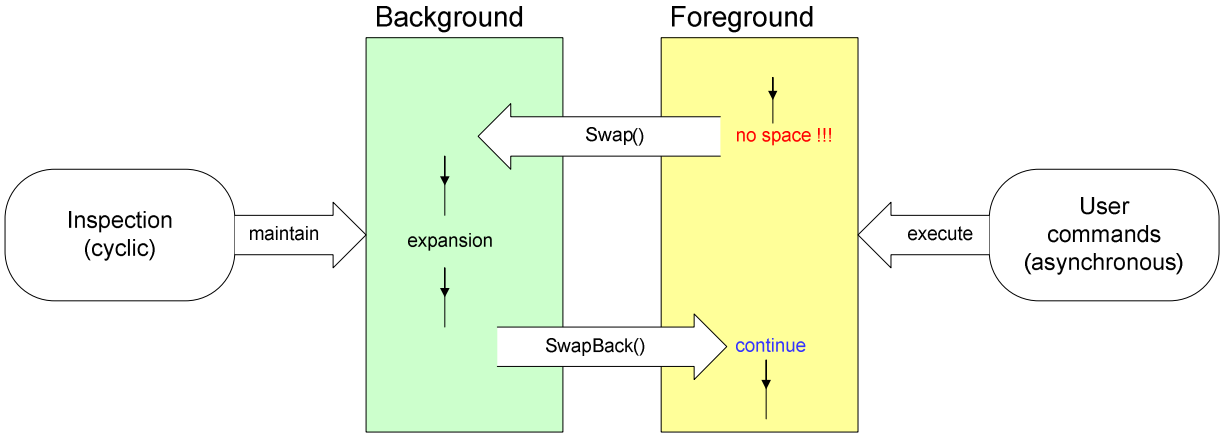
The foreground plane is receiving and executing user commands only. Any foreground command can always suspend the maintenance process running in the background. On the other hand a foreground command has to be finished before next command can be executed.

Variable oriented commands (read and write) are executed directly and completely in the foreground and are normally isolated from the maintenance running in background. Only when space-alert or checksum-error happens in the foreground the process focus is swapped temporary to the background.

Pool oriented commands (startup, shutdown, cleanup and format) are just passing the command-request to the background and waits for its completion. This allows re-usage of common FSM's used for background maintenance and foreground command execution.

In exceptional cases it can happen that due to very heavy write traffic the maintenance process running in background gets no chance to prepare enough space in time. In such a case the foreground write process can request "space expansion" at the background process before being able to continue writing. For that purpose the activity focus is swapped.

Figure 2-23 Swap mechanism scheme



This approach allows collision-free operation even the user do not use the EEL_Handler(t) and all commands are executed in “enforced” mode. It simplifies the handling at user side without loosing any flexibility in the operability. Swapping of execution focus plane does not change the command handling at user side. It is not visible at user side, just the command execution time increases for the time needed for the background processing.

As mentioned above, pool-oriented commands use the background processes for its execution. That means that all error-codes generated in the background must be transferred to the foreground (request variable). There could be errors like FAL_ERR_PROTECTION that never happens during normal operation. To simplify the error handling at user side unexpected error codes are transformed to one common error code EEL_ERR_INTERNAL. The original error code remains stored in the background and can be read by the function EEL_GetDriverStatus(&my_eel_driver_status).

2.12.2 Background plane

The background plane is dealing with background processes, normally executed when calling the function EEL_Handler(t) periodically. After EEL initialization the background process is passive (EEL-Handler does not have any effect and consumes, just few CPU cycles). After successful STARTUP the handler becomes active and starts the execution of the background process. There are several task the background process does manage, like:

a) background execution of pool related commands initiated by the foreground plane

b) background execution of exceptional handling initiated by the foreground:

- when less than 2 prepared blocks detected
- when checksum error during READ command

c) supervision of the refresh threshold and size of the invalid region

d) maintenance to eliminate problems detected by c)

Chapter 3 Application Programming Interface

The following chapters describe formally the user interface of the EEPROM Emulation Library.

3.1 Data types

This chapter describes all data definitions used and offered by the EEL.

3.1.1 Library specific simple type definitions

Simple numerical type used by the library:

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

Note: types are defined in EEL_types.h

3.1.2 Enumeration type “eel_command_t”

This type defines all codes of available commands:

```
/* EEL command set  */
typedef enum  {
                EEL_CMD_UNDEFINED        = (0x00),
                EEL_CMD_STARTUP          = (0x00 | 0x01),
                EEL_CMD_WRITE            = (0x00 | 0x02),
                EEL_CMD_READ             = (0x00 | 0x03),
                EEL_CMD_CLEANUP          = (0x00 | 0x04),
                EEL_CMD_FORMAT           = (0x00 | 0x05),
                EEL_CMD_SHUTDOWN         = (0x00 | 0x06)
              } eel_command_t;
```

Note: type is defined in EEL_types.h

Code value description:

EEL_CMD_UNDEFINED - undefined command (initial value)
EEL_CMD_STARTUP - plausibility check of the EEL data and driver
EEL_CMD_WRITE - creates new instance of specified EEL variable
EEL_CMD_READ - reads last instance of the specified EEL variable
EEL_CMD_CLEANUP - refresh of all variables (minimize active region)
EEL_CMD_FORMAT - format the EEL pool, all instances (data) are lost
EEL_CMD_SHUTDOWN - deactivates the EEL

3.1.3 Enumeration type “eel_operation_status_t”

This type defines all codes of available driver operation status:

```
/* type of the EEL driver operation status */
typedef enum  {
                EEL_OPERATION_PASSIVE     = (0x00),
                EEL_OPERATION_IDLE        = (0x30 | 0x01),
                EEL_OPERATION_BUSY        = (0x30 | 0x02)
              } eel_operation_status_t;
```

Note: type is defined in EEL_types.h

Code value description:

EEL_OPERATION_PASSIVE - when library is not yet started
EEL_OPERATION_IDLE - only background supervision process is active
EEL_OPERATION_BUSY - fore- or background process is active

3.1.4 Enumeration type “eel_access_status_t”

This type defines all codes of available driver access status:

```
/* type of the access status */
typedef enum  {
                EEL_ACCESS_LOCKED         = (0x00),
                EEL_ACCESS_UNLOCKED       = (0x40 | 0x01)
              } eel_access_status_t;
```

Note: type is defined in EEL_types.h

Code value description:

EEL_ACCESS_LOCKED - neither read nor write access possible
EEL_ACCESS_UNLOCKED - full access to the EEL is possible

3.1.5 Enumeration type “eel_status_t”

This type defines all codes of available request status and errors:

```
/* EEL status set  */
typedef enum  {
                EEL_OK                    = (0x00),
                EEL_BUSY                  = (0x00 | 0x01),
                EEL_ERR_CONFIGURATION     = (0x80 | 0x02),
                EEL_ERR_INITIALIZATION    = (0x80 | 0x03),
                EEL_ERR_ACCESS_LOCKED     = (0x80 | 0x04),
                EEL_ERR_COMMAND           = (0x80 | 0x05),
                EEL_ERR_PARAMETER         = (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;
```

Note: type is defined in EEL_types.h

Code value description:

EEL_OK	- no error occurred
EEL_BUSY	- request is under processing
EEL_ERR_CONFIGURATION	- bad FAL or EEL configuration
EEL_ERR_INITIALIZATION	- EEL_Init(), EEL_Open missed
EEL_ERR_ACCESS_LOCKED	- STARTUP missing or fatal operation error
EEL_ERR_COMMAND	- wrong command code
EEL_ERR_PARAMETER	- wrong parameter
EEL_ERR_REJECTED	- another request under processing
EEL_ERR_NO_INSTANCE	- no instance found (variable never written)
EEL_ERR_POOL_FULL	- no space for writing data
EEL_ERR_POOL_INCONSISTENT	- no active block found within EEL-pool
EEL_ERR_POOL_EXHAUSTED	- EEL pool to small for correct operation
EEL_ERR_INTERNAL	- internal error

3.1.6 Structured type “eel_request_t”

This type defines structure of the EEL request variables:

```
/* EEL request type */
typedef __near struct {
                  __near eel_u08*       address_pu08;
                  __near eel_u08        identifier_u08;
                  __near eel_u08        timeout_u08;
                  __near eel_command_t  command_enu;
                  __near eel_status_t   status_enu;
                } eel_request_t;
```

Note: type is defined in EEL_types.h

Structure member description:

address_pu08	- source/destination RAM-address
identifier_u08	- variable identifier
timeout_u08;	- number of timeout ticks for execution
command_enu;	- command has to be processed
status_enu;	- error code after command execution

3.1.7 Structured type “eel_driver_status_t”

This type defines structure of the EEL request variables:

```
/* type of the internal EEL driver status */
typedef struct {
                 eel_operation_status_t operationStatus_enu;
                 eel_access_status_t    accessStatus_enu;
                 eel_status_t           backgroundStatus_enu;
               } eel_driver_status_t;
```

Note: type defined in EEL_types.h

Structure member description:

operationStatus_enu	- operation status of the foreground process
accessStatus_enu	- access rights indicator
backgroundStatus_enu	- error status of the background process

3.2 Functions

Due to the request (data) oriented interface of the EEL the functional interface is very narrow. Beside the initialization function and some administrative function the whole EEPROM access is concentrated to two functions only: **EEL_Execute(**&my_eel_request**)** and **EEL_Handler(t).**

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

3.2.1 EEL_Init

Initialization of all internal data and variables.

C Language Interface (Renesas version)

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

C Language Interface (IAR version)

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

Pre-condition

The FDL must be initialized already

Post-condition

None

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
EEL_OK	eel_status_t	when EEL pool and descriptor OK
EEL_ERR_CONFIGURATION	eel_status_t	when EEL pool or EEL descriptor wrong

Code example:

```
eel_status_t    my_eel_status;

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

3.2.2 EEL_Open

This function can be used by the application to open the access to the EEL pool.

C Language Interface (Renesas version)

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

C Language Interface (IAR version)

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

Pre-condition

The FDL must be initialized already

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
none		

Code example:

```
EEL_Open();
```

3.2.3 EEL_Close

This function can be used by the application to close the access to the EEL pool.

C Language Interface (Renesas version)

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

C Language Interface (IAR version)

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

Pre-condition

None

Post-condition

In case that the USER part of the FDL-pool also “opened” too at that time, the Data Flash hardware remains active. To switch the Data Flash passive, both parts of the FAL-pool (EEL-part and USER-part) has to be closed.

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
none		

Code example:

```
EEL_Close();
```

3.2.4 EEL_Execute

This is one of the main function of the EEL the application can use to initiate execution of any command. Depending on the defined operation mode (time out value) this function returns:

a) immediately after execution of the first command state (timeout = 0)
b) after execution of the defined time-slice (0<timeout<255)
c) after execution of the complete command (timeout = 255)

C Language Interface (Renesas version)

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

C Language Interface (IAR version)

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

Pre-condition

EEL_Init() executed successfully
EEL_Open() must be executed before.

Post-condition

none

Argument

Argument	Type	Description
request_pstr	eel_request_t*	This argument defines user's request should be processed by the EEL. It is passing the request variable to the driver that is used for bi-directional information exchange before and during command execution between EEL and the application.

Return types/values

Argument	Type	Description
none		

Code example:

```
eel_request_t    my_eel_request_str;
eel_status_t     my_eel_status;

my_eel_status = EEL_Init();
EEL_Open();

/* enfoced mode  ---------------------------------------- */
my_eel_request_str.timeout_u08      = 0xFF;
my_eel_request_str.command_enu      = EEL_CMD_STARTUP;

EEL_Execute(&my_eel_request_str);
if(my_eel_request_str.status_enu != EEL_OK) MyErrorHandler();


/* timeout mode ----------------------------------------- */
my_eel_request_str.timeout_u08      = 5;
my_eel_request_str.command_enu      = EEL_CMD_FORMAT;


do {
     EEL_Execute(&my_eel_request_str);
     EEL_Handler(0);
}while(my_eel_request_str.status_enu == EEL_ERR_REJECTED);

do {
  EEL_Handler(5);
while(my_eel_request_str.status_enu == EEL_ERR_BUSY);

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

/* STARTUP after FORMAT mandatory (enfoced mode)----------- */
my_eel_request_str.timeout_u08      = 0xFF;
my_eel_request_str.command_enu      = EEL_CMD_STARTUP;

EEL_Execute(&my_eel_request_str);
if(my_eel_request_str.status_enu != EEL_OK) MyErrorHandler();


/* polling mode ----------------------------------------- */
my_eel_request_str.address_pu08     = (eel_u08)&A[0];
my_eel_request_str.identifier_u08   = 'A';
my_eel_request_str.timeout_u08      = 0;
my_eel_request_str.command_enu      = EEL_CMD_WRITE;

do {
     EEL_Execute(&my_eel_request_str);
     EEL_Handler(0);
}while(my_eel_request_str.status_enu == EEL_ERR_REJECTED);

do {
  EEL_Handler(0);
while(my_eel_request_str.status_enu == EEL_ERR_BUSY);

if(my_eel_status != EEL_OK) MyErrorHandler();
```

3.2.5 EEL_Handler

Depending on internal status of the EEL this function is managing different processes as follows:

a)
When no user command is processed in the foreground, the EEL_Handler(t) is executing the internal maintenance process. It is monitoring permanently the size of the “active region” to trigger the “refresh process” when exceeded the defined EEL_REFRESH_BLOCK_THRESHOLD. On the other side “preparation process” is triggered in the background whenever an invalid block is found in the EEL pool. Finally it checks if any requests from the foreground are pending in the meantime.

b)
If a foreground command is not finished in “timeout” or “polling” mode the EEL_Handler(t) takes care for continuation of the execution of not-finished commands in the next time-slices.

C Language Interface (Renesas version)

```
void  __far  EEL_Handler(eel_u08 timeout_u08);
```

C Language Interface (IAR version)

```
__far_func  void  EEL_Handler(eel_u08 timeout_u08);
```

Pre-condition

EEL initialized and opened

Post-condition

None

Argument

Argument	Type	Description
timeout_u08	eel_u08	Timeout value expressed in ticks. If timeout_u08=0 only one state of the internal FSM will be executed. If timeout_u08<>0 internal states are executed as long the timeout counter>0.

Return types/values

Argument	Type	Description
none		

Code example:

```
/* The best place for EEL_Handler is the scheduler loop */

eel_u08   my_time_slice;

my_time_slice = 0x00;
do {
      EEL_Handler(my_time_slice);
      User_Task_A();
      User_Task_B();
      User_Task_C();
      User_Task_D();
} while(true);
```

3.2.6 EEL_TimeOut_CountDown

This function counts the internal 8-bit timeout counter down to zero. When executing a command, the program counter remains inside the EEL_Execute(&my_eel_request) or EEL_Handler(t) as long this counter>0. The EEL_TimeOut_CountDown() function can be called at any place in the application. The preferable place is any periodical interrupt service routine, for example the timer ISR of the operating system. When the internal 8-bit timer achieve the value 0x00 the EEL_TimeOut_CountDown() function stops the counting. The counter starts counting again when a new “timeout” request was placed via EEL_Execute(&my_eel_request) or when EEL_Handler(t) was called with t>0.

C Language Interface (Renesas version)

```
void      __far  EEL_TimeOut_CountDown(void);
```

C Language Interface (IAR version)

```
__far_func  void  EEL_TimeOut_CountDown(void);
```

Pre-condition

none

Post-condition

Timeout counter decremented in case it was running.

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
none		

Code example:

```
#pragma interrupt INTTM00 isr_OS_timer

void isr_OS_timer(void)
{
  EEL_TimeOut_CountDown();
}
```

3.2.7 EEL_GetDriverStatus

This function opens a way to check the internal status of the EEL driver in advance, before placing a request.

C Language Interface (Renesas version)

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

C Language Interface (IAR version)

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

Pre-condition

EEL initialized and opened

Post-condition

none

Argument

Argument	Type	Description
driverStatus_pstr	eel_driver_status_t*	This argument is a placeholder for capturing the internal status of the driver. It indicates the operation status, the access status and the status of the background process of the EEL.
EEL_OPERATION_PASSIVE	driverStatus_pstr-> operationStatus_enu	EEL not initialized or not opened or not started-up successfully. Operation and access to the data is not possible.
EEL_OPERATION_IDLE		After successful STARTUP when neither foreground command nor background maintenance is active.
EEL_OPERATION_BUSY		EEL is processing an user command or when main-tenance process is active in background. Other commands are not possible at that time.
EEL_ACCESS_LOCKED	driverStatus_pstr-> accessStatus_enu	STARTUP not executed/successful or access to data-flash was locked by the EEL due to any internal problems.
EEL_ACCESS_UNLOCKED		STARTUP executed successfully, read/write access to the EEL-pool is possible
any	driverStatus_pstr-> backgroundStatus_enu	Any value of the eel_status_t related to background processes are possible. It will be actualized/overwritten by the background process only. The usage of it is quite limited.

Return types/values

Argument	Type	Description
none		

Code example:

```
eel_request_t           my_eel_request_str;
eel_status_t            my_eel_status_enu;
eel_driver_status_t     my_eel_driver_status_str;

my_eel_status_enu = EEL_Init();
EEL_Open();

/* execute STARTUP if not already done */
EEL_GetDriverStatus(&my_eel_driver_status_str);
if(my_eel_driver_status_str.operationStatus_enu==EEL_OPERATION_P
ASSIVE)
{
  my_eel_request_str.timeout_u08      = 0xFF;
  my_eel_request_str.command_enu      = EEL_CMD_STARTUP;

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

/* write data when access already possible */
EEL_GetDriverStatus(&my_eel_driver_status_str);
if(my_eel_driver_status_str.accessStatus_enu==EEL_ACCESS_UNLOCKE
D)
{
  my_eel_request_str.address_pu08     = (eel_u08)&A[0];
  my_eel_request_str.identifier_u08   = 'A';
  my_eel_request_str.timeout_u08      = 0;
  my_eel_request_str.command_enu      = EEL_CMD_WRITE;

  do {
    EEL_Execute(&my_eel_request_str);
    EEL_Handler(0);
  }while(my_eel_request_str.status_enu==EEL_ERR_REJECTED);

  do {
    EEL_Handler(0);
  while(my_eel_request_str.status_enu==EEL_ERR_BUSY);

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

3.2.8 EEL_GetSpace

This function provides the number of flash words inside the active-head and the prepared region that can still absorb new references and data.

C Language Interface (Renesas version)

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

C Language Interface (IAR version)

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

Pre-condition

EEL must be initialized, opened and STARTUP must be executed before space can be calculated

Post-condition

none

Argument

Argument	Type	Description
space_pu16	eel_u16*	Address of the space information variable

Return types/values

Argument	Type	Description
EEL_OK	eel_status_t	When space value is correct
EEL_ERR_INITIALIZATION	eel_status_t	When EEL_Init() or EEL_Open() is missing
EEL_ERR_ACCESS_LOCKED	eel_status_t	when STARTUP command missing
EEL_ERR_REJECTED	eel_status_t	when space not stable, just being modified.

Code example:

```
eel_request_t           my_eel_request_str;
eel_status_t            my_eel_status_enu;
eel_u16                 my_eel_space_u16;

my_eel_status = EEL_Init();
EEL_Open();

/* execute STARTUP if not already done */
EEL_GetDriverStatus(&my_eel_driver_status_str);
if(my_eel_driver_status_str.operationStatus_enu==EEL_OPERATION_P
ASSIVE)
{
  my_eel_request_str.timeout_u08      = 0xFF;
  my_eel_request_str.command_enu      = EEL_CMD_STARTUP;

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

/* read current space value */
my_eel_status_enu = EEL_GetSpace(&my_eel_space_u16);

if(my_eel_status_enu==EEL_OK)
{
  if(my_eel_space_u16<MY_SPACE_ALERT_THRESHOLD)
  {
    my_eel_request_str.timeout_u08      = 0xFF;
    my_eel_request_str.command_enu      = EEL_CMD_CLEANUP;

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

3.2.9 EEL_GetVersionString

This function can be used by the application to check and control the library version information at runtime. I provides the pointer to the zero-terminated library version-string in ASCII format.

Table 5 Format information of the library version string

Field:	Field meening	Field type	Field length	Comment
1	Library mark	fix	1	"S", "E", "A" ....
2	Device info	variable	1-6	free content between field 1 and field 3
3	Library type info	fix	3	starting with "T" followed by 2 char
4	Compiler info	variable	5-6	starting with "R"/"I" or "G" followed by 3 digits
5	Library version	fix	4	starting with "E" or "V" followed by 3 max. digits
6	compiler setting	fix	3	starting with "C" followed by 2 digits
7	Zero-Termination	fix	1	always 0x00
Total size:			18-24	

Examples:

Version string of Renesas version 1.10 of the EEL is: ERL78T01R110GV110

Version string of the IAR version 1.10 of the EEL is: ERL78T01I100GV110

C Language Interface (Renesas version)

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

C Language Interface (IAR version)

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

Pre-condition

none

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
	__far eel_u08*	pointer to the first character of the zero-terminated library version string.

Code example:

```
__far  eel_u08*   my_version_string_pu08;

my_version_string_pu08 = EEL_GetVersionString();

PrintMyVersion(&my_version_string_pu08);
```

Chapter 4 Operation

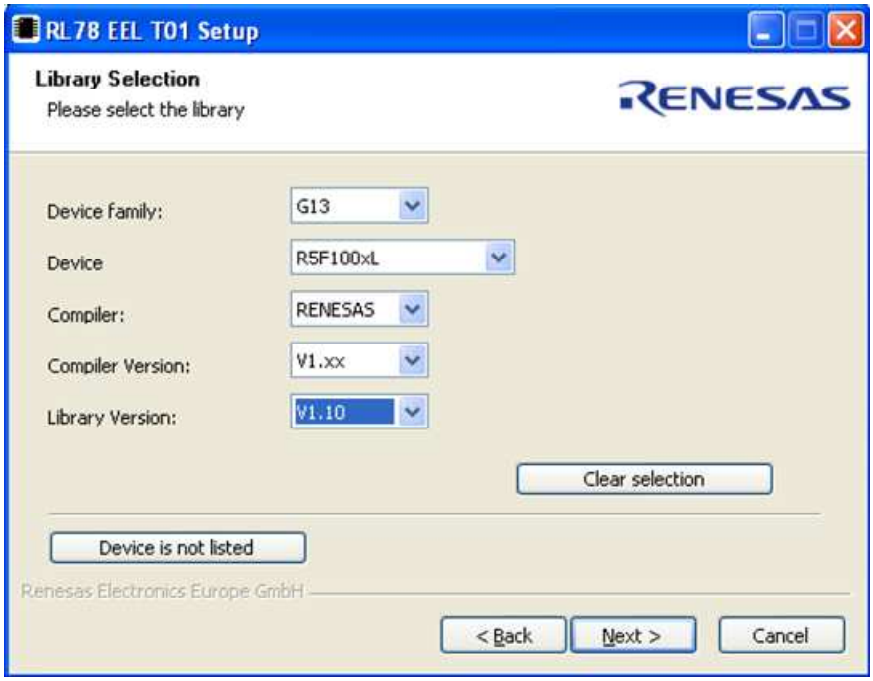
This chapter describes the installation, integration, configuration and of the EEPROM Emulation library.

4.1 Installation

All components of the EEPROM Library package are extracted by the self extracting installer file **RENESAS_EEL_RL78_T01E_*version*.exe**

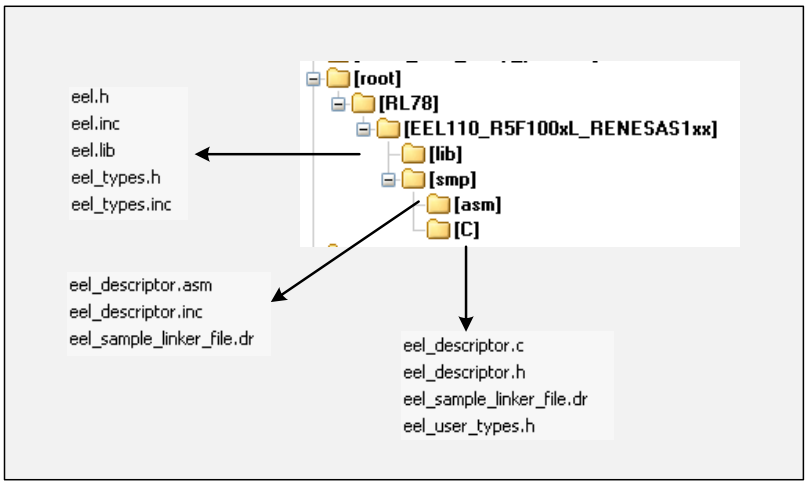
After acceptation of the license the library for the required device and compiler environment can be selected.

Figure 4-1 EEL installer mask



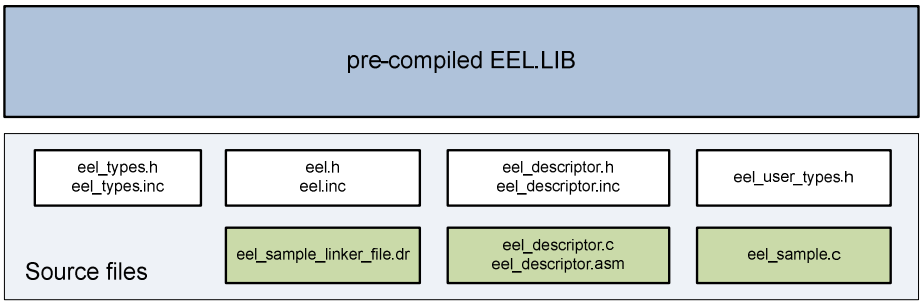
After successful installation all EEL related files are copied to the chosen root-directory

Figure 4-2 Subdirectory tree of the EEL after installation



The main file of the installed library package is the pre-compiled EEL. The header and include files defining the API as well the descriptor files are available in source form.

Figure 4-3 File structure of the EEL delivery package



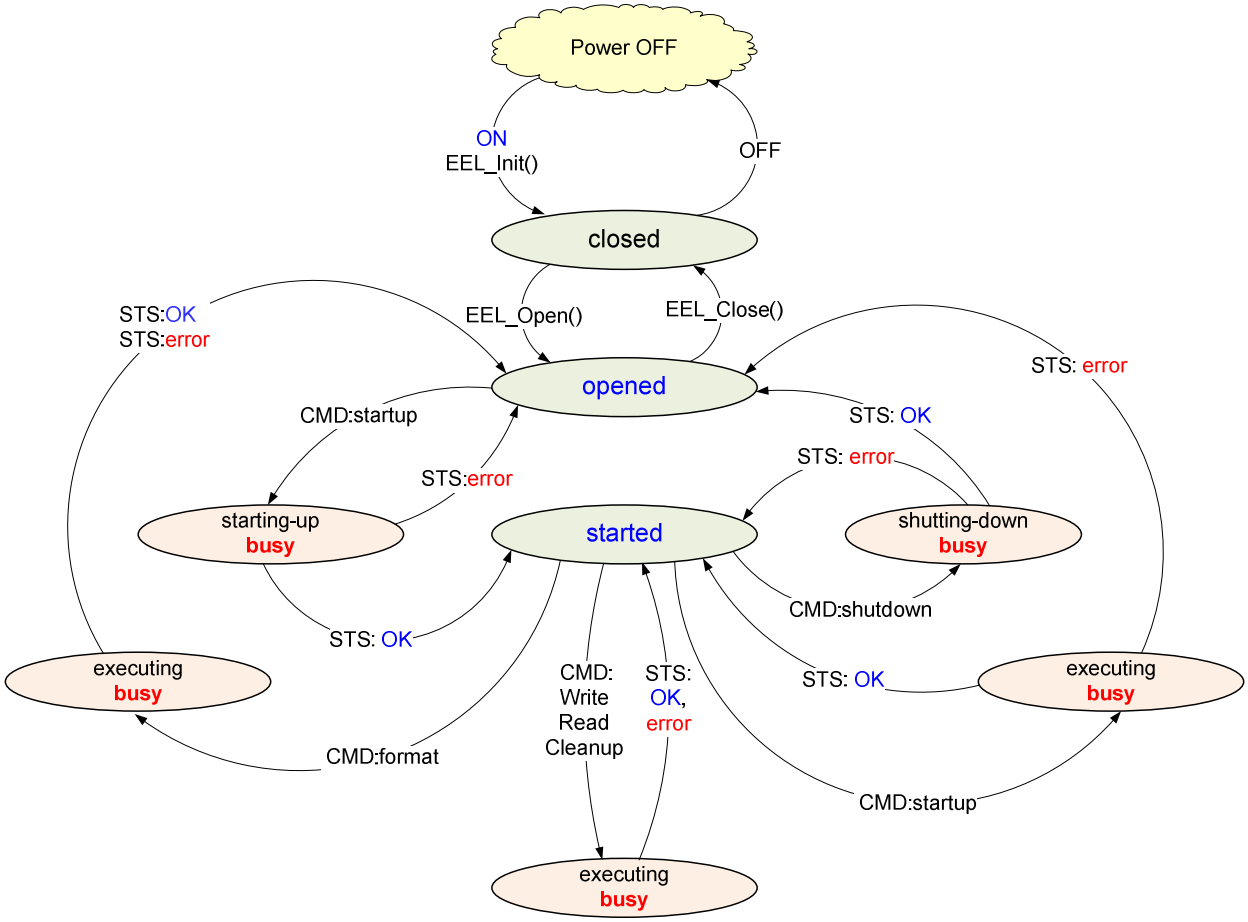
Note:

Assembler files (*.INC, *.ASM) are available for Renesas compiler environment only.

4.2 Basic workflow

To be able to use the EEL (execute commands) in a proper way the user has to follow a specific startup and shutdown procedure.

Figure 4-4 Basic workflow of the EEL



Notes:

1 - The FORMAT command can be executed without successful STARTUP

2 - After execution of the FORMAT command the EEL goes into state “opened”, so STARTUP command must be executed again .

4.3 Configuration

The EEL configuration can be divided into two stages:

- configuration of the EEL pool in the FAL-descriptor
- configuration of the EEL library in EEL-descriptor

4.3.1 Pool configuration

The size of the EEL pool is configured in the FAL_descriptor files. The minimum size of the EEL-pool is 4 blocks (1 active, 1 prepared, 1 being erased and one potentially excluded). This is the virgin condition. At runtime the EEL must be able to work with at least 1 excluded block.

File FDL_descriptor.h

```
EEL_POOL_SIZE 6      /* specify number of EEL blocks, min 4 */
```

Note:
EEL_POOL_SIZE should not exceed the FDL_POOL_SIZE

File EEL_descriptor.h

```
EEL_STORAGE_TYPE           'D'    /* determines flash medium */
```

'D' - Data Flash and FDL in use
other values - invalid

```
EEL_REFRESH_BLOCK_THRESHOLD 3 /* determines refresh threshold */
```

Note:
It is not easy to develop a precise and certain formula for the refresh-threshold because the order of written/refreshed instances in the active-region is a random process decided at runtime. Good results can be achieved when defining the threshold to (N + 1) where N is the number of blocks needed for coverage of all initial instances of all variables declared in EEL descriptor. Generally the bigger the prepared region the smoother is the run-time operation of the EEL Therefore the threshold should be minimized in relationship to the amount of data.

It is strongly recommend to check the runtime behavior of the EEL at a given configuration in the target system under worst case conditions (variable size, variable number, threshold, pool-size, block exclusion, writing speed...) before establishing and releasing the configuration.

4.3.2 Variable configuration

The number and size of variable managed by the EEL are configured in the EEL_descriptor files. The EEL driver/library can only read/write variable-ID's registered in the EEL-descriptor.

File EEL_descriptor.h

```
EEL_VAR_NO   8   /* number of variables handled by EEL, min 1 */
```

File EEL_descriptor.c

```
/* EEL variable size expressed in bytes */
#define   bsize_A   (sizeof(type_A))
#define   bsize_B   (sizeof(type_B))
#define   bsize_C   (sizeof(type_C))
#define   bsize_D   (sizeof(type_D))
#define   bsize_E   (sizeof(type_E))
#define   bsize_F   (sizeof(type_F))
#define   bsize_X   (sizeof(type_X))
#define   bsize_Z   (sizeof(type_Z))

/* EEL variable size expressed in words */
#define   wsize_A   (bsize_A+3)/4)
#define   wsize_B   (bsize_B+3)/4)
#define   wsize_C   (bsize_C+3)/4)
#define   wsize_D   (bsize_D+3)/4)
#define   wsize_E   (bsize_E+3)/4)
#define   wsize_F   (bsize_F+3)/4)
#define   wsize_X   (bsize_X+3)/4)
#define   wsize_Z   (bsize_Z+3)/4)


__far const eel_u08 eel_descriptor[EEL_VAR_NO+1][4] =
{
/*identifier     word-size (1...64)  byte-size (1..255)  RAM-Ref.  */
/*-------------------------------------------------------------------*/
  (eel_u08)'a',  (eel_u08)(wsize_A), (eel_u08)(bsize_A),  0x01, \
  (eel_u08)'b',  (eel_u08)(wsize_B), (eel_u08)(bsize_B),  0x01, \
  (eel_u08)'c',  (eel_u08)(wsize_C), (eel_u08)(bsize_C),  0x01, \
  (eel_u08)'d',  (eel_u08)(wsize_D), (eel_u08)(bsize_D),  0x01, \
  (eel_u08)'e',  (eel_u08)(wsize_E), (eel_u08)(bsize_E),  0x01, \
  (eel_u08)'f',  (eel_u08)(wsize_F), (eel_u08)(bsize_F),  0x01, \
  (eel_u08)'x',  (eel_u08)(wsize_X), (eel_u08)(bsize_X),  0x01, \
  (eel_u08)'z',  (eel_u08)(wsize_Z), (eel_u08)(bsize_Z),  0x01, \
   0x00,          0x00,               0x00,               0x00, \
};
```

The EEL descriptor is a [N+1] vector containing descriptor information of each EEL variable (N is the total number of EEL variables registered).

Each variable descriptor is an array of 4 bytes.
The EEL descriptor must be terminated by a descriptor terminator (4 bytes 0x00). This pattern is used internally by the EEL as descriptor-end-criteria in the variable searching process.

Identifier:
The 1'st byte of the variable descriptor is the "identifier" field that must be unique within the whole EEL-descriptor. Variables can be identified, read and written by using this identifier.

Word-size:
The 2'nd byte of the variable descriptor specifies the size of the variable expressed in words.

Byte-size:
The 3'rd byte of the variable descriptor specifies the size of the variable expressed in bytes.

RAM-ref:
The 4'th byte of the variable descriptor is the "RAM-reference" which should indicate EEL variables referenced by RAM-reference. This field is only relevant when EEL is using the FCL for flash access. When FDL is accessing the flash, the "RAM-reference" files doesn't have any meaning (in that case each variable is referenced by RAM automatically).

4.3.3 Pool configuration hints and tipps

During operation the situation in the EEL-pool changes whenever data are written into it. This is a high dynamic, unpredictable random process. On the other hand each application has different timing requirements when writing data. Some application need so called burst write (writing many data in relatively short time e.g. crash data in airbag applications). Other applications have to write data permanently in equidistant intervals like odometer in automotive applications. Moreover the size of variables and its individual write cycles and writing frequency may influence the real write-time.

When writing data into the EEL-pool three different cases are possible:

1) enough space for the instance and its reference exists in active head
2) not enough space in active head but more than 2 prepared blocks exist.
3) not enough space in active head but less than 3 prepared blocks exist.

In case 1) the execution time of the WRITE command consists of the pure writing-time only:
T_1(WRITE) = t(write).

In case 2) the execution time of the WRITE command consists of two components: the activation-time and writing-time:
T_2(WRITE) = t(activation) + t(write).

In case 3) the execution time of the WRITE command consists of three components: the expansion-time, activation-time and writing-time:
T_3(WRITE) = t(expand) + t(activation) + t(write).

Where: T_1(WRITE) < T_2(WRITE) <<< T_3(WRITE)

The difference between T_1(WRITE) and T_2(WRITE) is very small and cannot/mustn't be avoided by the user (system architecture related behavior).

The T_3(WRITE) is much longer than T_1/T_2 (WRITE) because it incorporates block erase time. Consequently to keep writing-time constant during EEL operation the user should avoid situation described in case 3) by keeping the background maintenance alive. When calling the EEL_Handler(t) permanently in the application idle loop the EEL will automatically remove conditions described in case 3) according to the EEL-pool configuration.

There are some general dependencies that should be taken into account when configuring the EEL and its pool.

1) the bigger the prepared area S(P) the better the real time performance
2) the bigger the S(F) the better (more efficient) the usage of erase cycles
3) the refresh threshold should be max. 1 block bigger than S(D)

In below examples following abbreviations were used:

B(P) – number of prepared blocks in initially programmed EEL-pool
B(D) – number of blocks containing initial data
S(H) – size of block header expressed in flash words
S(R) – size of the initial reference area in the active heading block in words
S(F) – size of the free space in active heading block in words (active space)
S(D) – size of the initial data area in the active head expressed in word
S(B) – size of the block expressed in words
SEP – size of the min. separator between reference and data area in words
N – number of variables registered in the EEL_descriptor.
wsize - size of the given variable expressed in words (see EEL_descriptor).
TH - refresh threshold defined in eel_descriptor.h

Where:

S(B) = 512, S(H) = 8, SEP = 3, S(R) = 2*N + SEP

$$S(D) = \sum_{i=1}^{N} wsize(data_i)$$

After initial programming following situation in EEL pool is assumed:

1) the active region of the EEL-pool contains only one instance (the initial one) of each variable registered in EEL descriptor.

2) the remaining EEL (none-exluded and data-less) blocks are prepared.

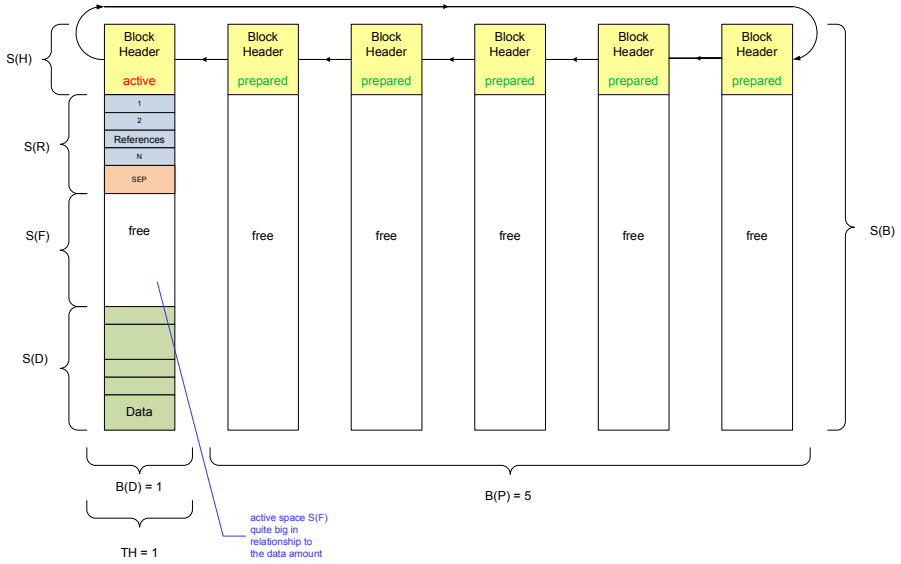
In such situation the remaining active space S(F) in the active heading block and the number of prepared blocks S(P) could be one criteria for proper configuration of the refresh threshold.

S(F) = S(B) – S(H) – S(R) – S(D)

CAUTION:
Before releasing the EEL configuration have to be ensured by tests under worst case conditions (write frequency, write duration, block exclusion and so on) required by the application.

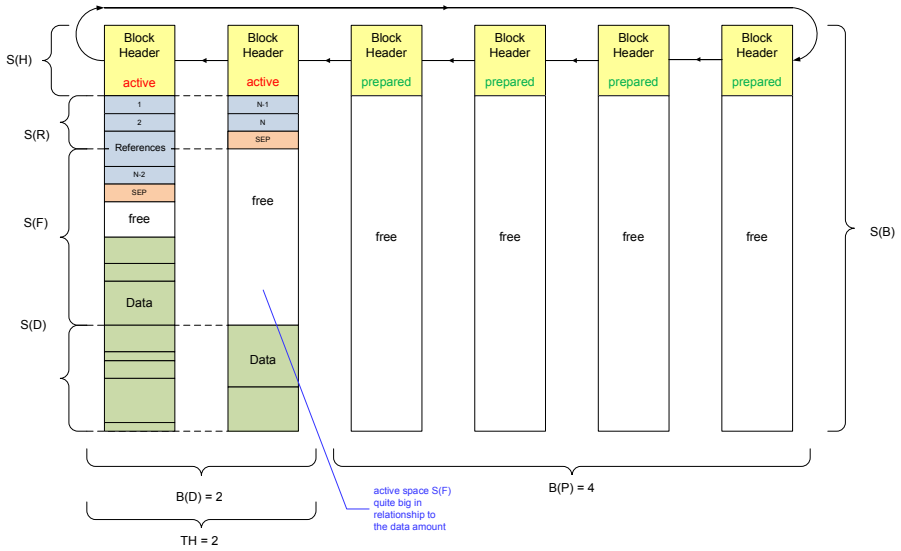
The following practical examples of EEL pool configuration should illustrate the dependancies.

Figure 4-5 Configuration for small data amount where S(F) is sufficient



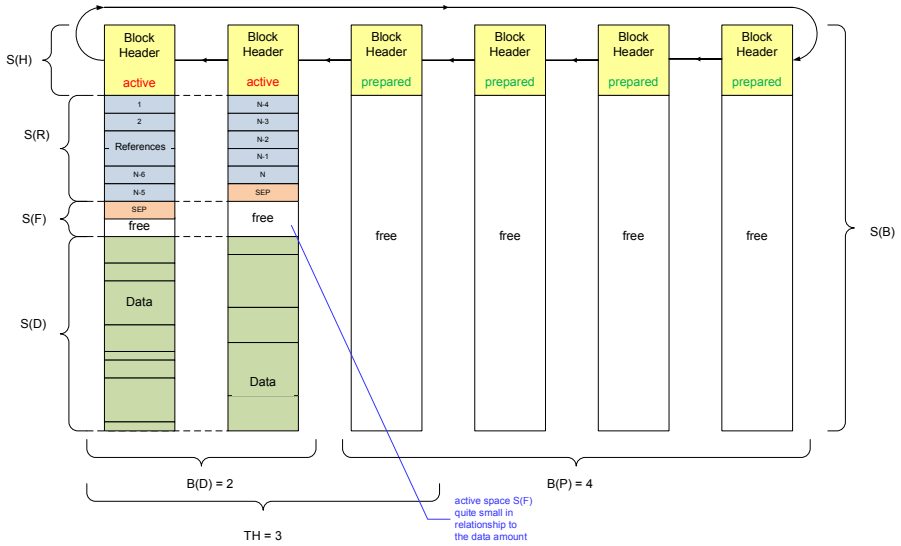
In the above scenario the active space S(F) is quite big, so that many instances of relatively small variables can be written into it before activation of the next block becomes necessary. When setting TH=1 the B(P) will be maximized automatically by the background process (EEL_Handler(t)). The relatively big buffer of prepared blocks allows intensive, continuous writing process for a long time before “space expansion” will be enforced by a pool-full situation.

Figure 4-6 Configuration for larger data amount where S(F) is sufficient



This example is similar to the previous one, but the total amount of initial data excides the space in one block. The active space is big enough, consequently the refresh threshold TH can be set to 2 to keep B(P) at maximal possible level.

Figure 4-7 Configuration for larger data amount where S(F) is not sufficient



In that example like in the previous one, the initial data occupies 2 blocks (B(D)=2), but in that case the remaining space S(F) in the active head is very small. To avoid that each write access would immediately cause a refresh and afterwards an erase cycle, the refresh threshold TH must be set to TH = B(D) + 1 = 3 in that case.

4.4 Initialisation

After power-on RESET the EEL has to be initialized by using the EEL_Init function. After this the plausibility of the configuration is checked and all internal variables are initialized. The driver remains passive and access to the flash medium is disabled.

```
my_eel_status_u08 = EEL_Init();

if (my_eel_status_u08 == EEL_OK)
{
   /* EEL is initialized */
}
else  My_ErrorHandler();
```

4.5 EEL activation and deactivation

After power-ON reset the Data Flash hardware is passive. Before using the EEL commands the access to the Data Flash has to be opened and the clock of the Data Flash hardware has to be switched on.

The physical resource data-flash is divided in the FAL into two virtual parts: the EEL-pool and the USER-pool. Both of them can be opened and closed independently. To open access to the EEL-pool the EEL_Open() function has to be called. To avoid unexpected side-effects the FAL is managing the Data Flash clock status (ON/OFF) internally.

The sequencer clock:

- is OFF after FAL_Init(...)
- goes ON when any part of the FAL-pool is beeing opened.
- remains ON when any part of the FAL-pool is still open
- goes OFF when both parts of the FAL-pool were closed.

The EEL-pool can be opened and clocsed by using the interface function EEL_Open()/EEL_Close().

```
....
<POWER-ON RESET>              /* sequencer clock is OFF */
....
my_fal_status = FAL_Init(&my_fal_descriptor);
if (my_fal_status <> FAL_OK) My_ErrorHandler();
....
....
my_eel_status = EEL_Init();
if (my_eel_status <> EEL_OK) My_ErrorHandler();
....
....
EEL_Open(); /* data flash clock starts here controlled */
....
....
EEL-commands can be executed here
....
....
....
FAL_Open();          /* data flash clock remains ON here  */
....
....
FAL commands can be used for access to the USER-pool
....
....
EEL_Close();    /* data flash clock remains ON because   */
....            /* FAL is still accessing the USER-pool  */
....
EEL-commands cannot be executed anymore
FAL-commands can be used for access to the USER-pool

....
....
FAL_Close();    /* data flash clock is switched OFF here  */
....
....
```

4.6 Foreground and background process

The backgound process is not visible directly to the user. It should take care for keeping conditions defined by the user in the configuration. Especially minimation of the invalide region and maximation of the space (according to the predefined refresh threshold).

4.6.1 Controlling background process

When automatical maintenance is required, the EEL_Handler(t) has to be called periodically in any loop (for example in the idle-loop or in the scheduler-loop).

When the application want to know if background maintenance is surelly finished the operation status provided by EEL_GetDriverStatus(...) must be stable EEL_OPERATION_IDLE for at least 4 EEL_Handler(0) calls.

In other words, min. 4 internal states of the EEL must be executed in EEL_OPERATION_IDLE operation-status to be sure that the background maintenance is definitively finished.

Figure 4-8 Example flow to ensure background passivity (enforced mode only)

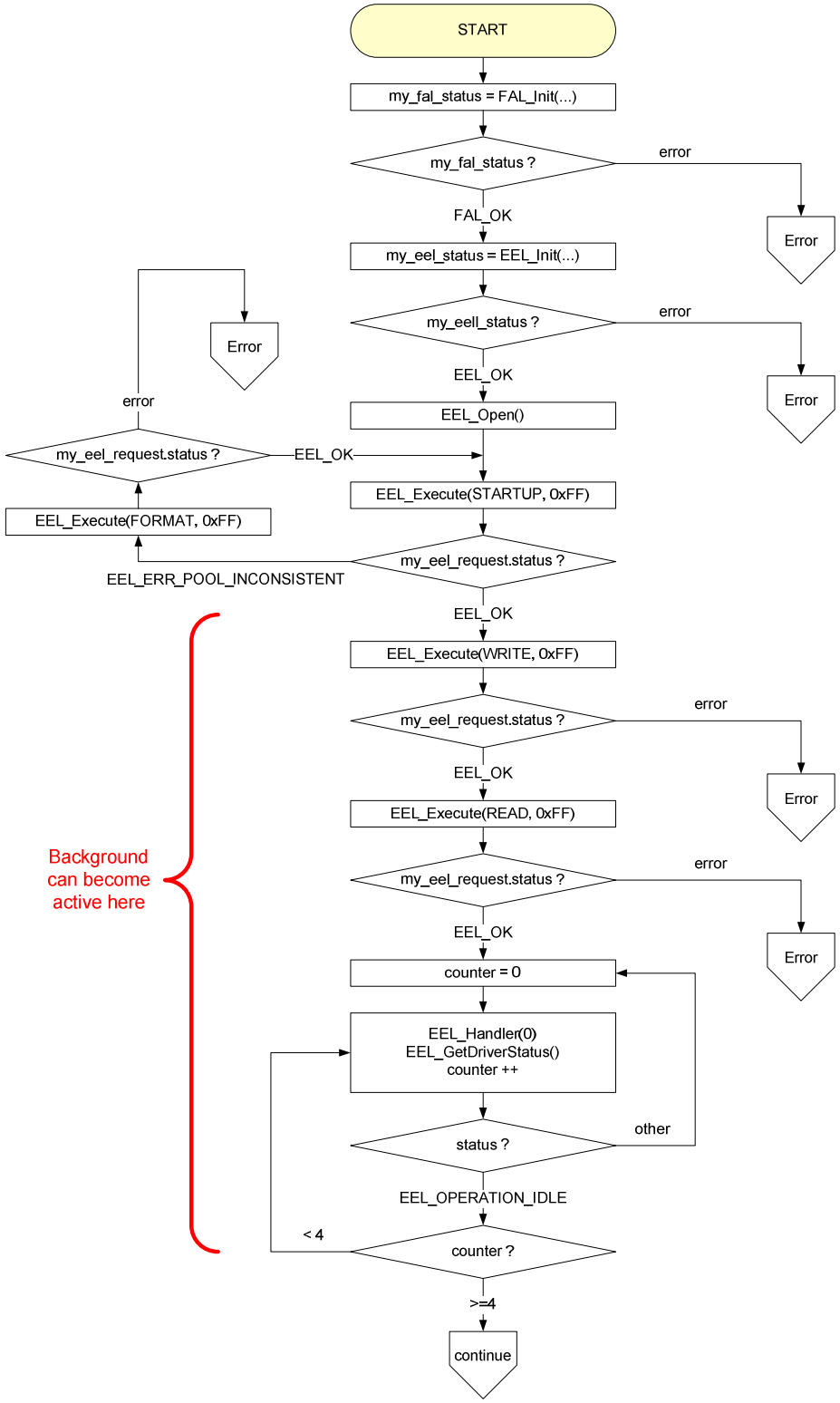
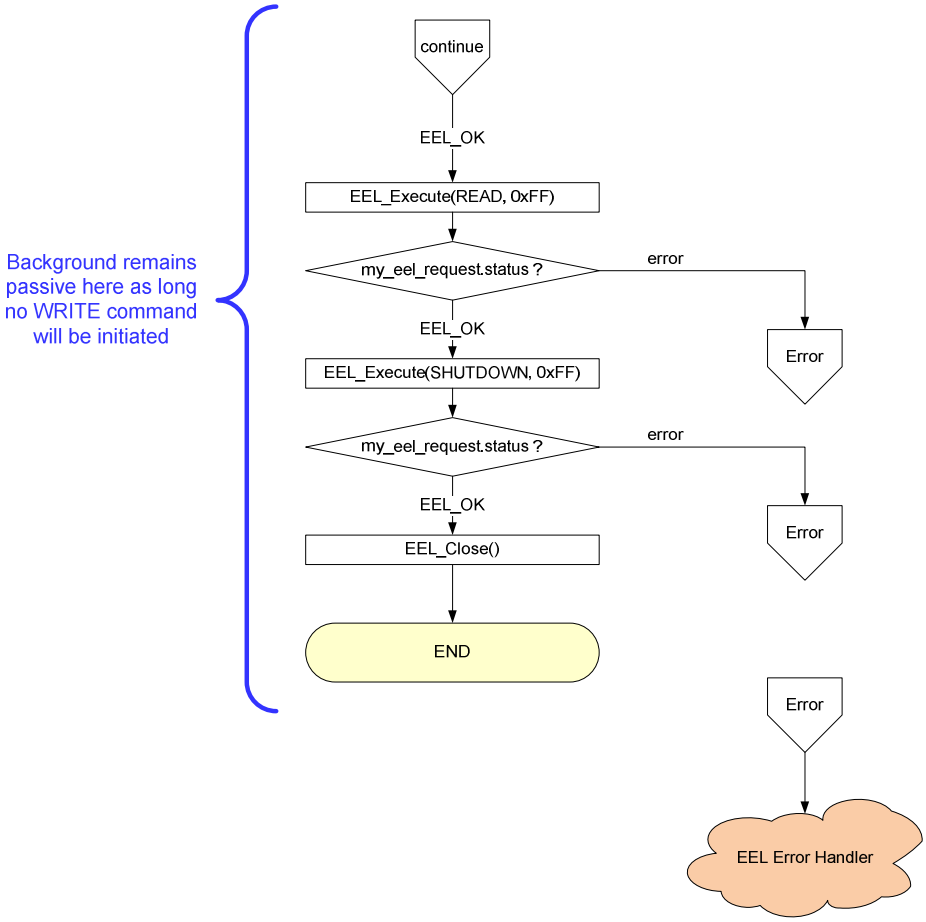


Figure 4-9 Example flow to ensure background passivity (cont.)

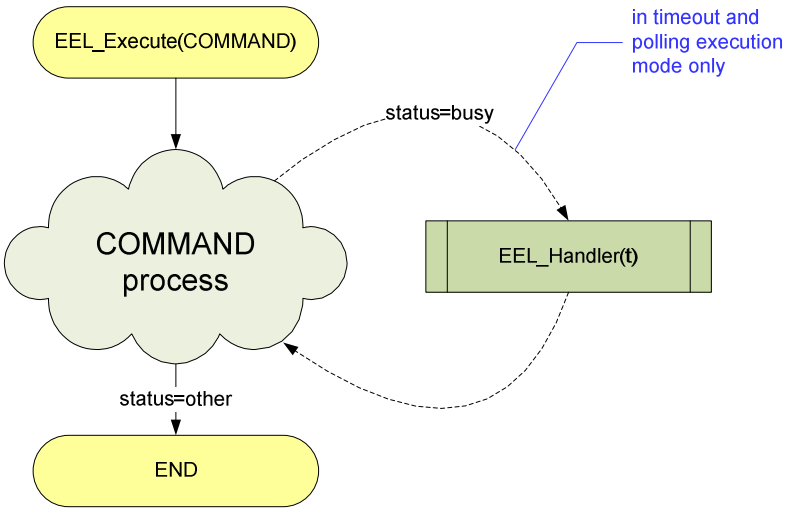


4.7 Commands

EEL commands has to be initiated by passing completed EEL-request using the function EEL_Execute(&my_eel_request). To simplify the handling of the EEL the command spectrum was reduced to the esential only. Depending on the affected object there are two groups of commands supported by the EEL. Some of them influences the operation and status of the wohle EEL-pool and some other the instance data only.

All EEL commands are executed/handled in the same wise and can be executed in individual execution mode.

Figure 4-10 General command execution flow



4.7.1 Pool oriented commands

EEL pool oriented command influences the blocks or data in the wohl EEL pool.

4.7.1.1 Command STARTUP

The startup command interpretes the actual status of the EEL-pool, especially the region parameters, block status flags and instance references. Successful STARTUP command opens the access to the EEL data for the variable oriented commands.

Table 6 Status of EEL_CMD_STARTUP command

Status	Class	Background and Handling	
EEL_ERR_INITIALIZATION	heavy	meaning	EEL not initialized or not opened
		reason	wrong handling on user side
		remedy	Initialize and open EEL before using it
EEL_ERR_COMMAND	light	meaning	invalid command code
		reason	unknown code used in request
		remedy	use eel_command_t type only
EEL_ERR_POOL_INCONSISTENT	heavy	meaning	pool structure not usable
		reason	inconsistent EEL pool detected *)
		remedy	FORMAT the EEL pool
EEL_ERR_POOL_EXHAUSTED	fatal	meaning	EEL pool size smaller < 3 blocks
		reason	to much blocks excluded
		remedy	no remedy, EEL dead
EEL_ERR_REJECTED	normal	meaning	EEL cannot accept the request
		reason	EEL busy with any other request
		remedy	wait until status changes or call EEL_Handler() until request accepted.
EEL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	wait until status changes call EEL_Handler() until request accepted.
EEL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution
		remedy	nothing

Supported execution modes:
enforcing, timeout, polling

Note 1):
EEL pool inconsistency can be caused by various reasons, for example:
- FIP flag is <> 0xFFFFFFFF
- RWP or DWP not found
- no active region detected or active-head missing
- active region not homogenous (discontinued by invalid block)
- all blocks excluded

Code example (enforced mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited STARTUP request */
my_eel_request.command_enu  = EEL_CMD_STARTUP;
my_eel_request.timeout_u08  = 255;
EEL_Execute(&my_eel_request);

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

Code example (timeout mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited STARTUP request */
my_eel_request.command_enu  = EEL_CMD_STARTUP;
my_eel_request.timeout_u08  = 20;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(20);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........

/* periodical counting timeout tick */
void isr_tm01(void)
{
  EEL_TimeOut_CountDown();
}
```

Code example (polling mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;

.........
.........
.........

/* specification of a time limited STARTUP request */
my_eel_request.command_enu  = EEL_CMD_STARTUP;
my_eel_request.timeout_u08  = 0;
EEL_Execute(&my_eel_request);

.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(0);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

.

4.7.1.2 Command SHUTDOWN

There is no real functionality behind the SHUTDOWN command. It is just used for synchronization between the background processes and the application. Practically it is just waiting until all running background processes (REFRESH, EXPANSION,...) are finished correctly. The access to the EEL pool is closed and the access status provided by EEL_GetDriverStatus(&my_driver_status) is EEL_ERR_ACCESS_LOCKED. Also the EEL_Handler(t) becomes passive and does not consume CPU time anymore (just few clocks).

Table 7 Status of EEL_CMD_SHUTDOWN command

Status	Class	Background and Handling	
EEL_ERR_INITIALIZATION	heavy	meaning	EEL not initialized
		reason	wrong handling on user side
		remedy	Initialize EEL before using it
EEL_ERR_COMMAND	light	meaning	invalid command code
		reason	unknown code used in request
		remedy	use eel_command_t type only
EEL_ERR_INTERNAL	heavy	meaning	unexpected/unknown error code generated in background
		reason	SW bug, EMI, unexpected problems
		remedy	no standard remedy possible. Next STARTUP should manage the problem
EEL_ERR_REJECTED	normal	meaning	EEL cannot accept the request
		reason	EEL busy with other request
		remedy	Call EEL_Handler() and retry later
EEL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	Call EEL_Handler() until status have changed.
EEL_OK	normal	meaning	request was finished regular
		reason	no problems did happen during command execution
		remedy	nothing

Supported execution modes:
enforcing, timeout, polling

Code example (enforced mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
/* specification of a time limited SHUTDOWN request */
my_eel_request.command_enu  = EEL_CMD_SHUTDOWN;
my_eel_request.timeout_u08  = 255;
EEL_Execute(&my_eel_request);

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();
.........
.........
```

Code example (timeout mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
/* specification of a time limited SHUTDOWN request */
my_eel_request.command_enu  = EEL_CMD_SHUTDOWN;
my_eel_request.timeout_u08  = 20;
EEL_Execute(&my_eel_request);
.........
.........
/* execute a state as long not finished */
do{
  EEL_Handler(20);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();
.........
.........
.........
/* periodical timeout count tick */
void isr_tm01(void)
{
  EEL_TimeOut_CountDown();
}
```

Code example (polling mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
/* specification of a time limited SHUTDOWN request */
my_eel_request.command_enu  = EEL_CMD_SHUTDOWN;
my_eel_request.timeout_u08  = 0;
EEL_Execute(&my_eel_request);
.........
.........
/* execute a state as long not finished */
do{
  EEL_Handler(0);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();
.........
.........
```

4.7.1.3 Command FORMAT

The format command destroys all data and creates an “empty” EEL pool consists of one active block. All remaining “not excluded” blocks are “prepared” by this command. After format the STARTUP command must be executed after FORMAT to identify the new EEL-pool status.

Table 8 Status of EEL_CMD_FORMAT command

Status	Class	Background and Handling	
EEL_ERR_INITIALIZATION	heavy	meaning	EEL not initialized
		reason	wrong handling on user side
		remedy	Initialize EEL before using it
EEL_ERR_COMMAND	light	meaning	invalid command code
		reason	unknown code used in request
		remedy	use eel_command_t type only
EEL_ERR_POOL_EXHAUSTED	fatal	meaning	EEL pool size smaller < 3 blocks
		reason	to much blocks excluded
		remedy	no remedy, EEL dead
EEL_ERR_INTERNAL	heavy	meaning	unexpected/unknown error code generated in background
		reason	SW bug, EMI, unexpected problems
		remedy	No standard remedy possible, analyze background status for details.
EEL_ERR_REJECTED	normal	meaning	EEL cannot accept the request
		reason	EEL busy with other request
		remedy	Call EEL_Handler or retry later
EEL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	Call EEL_Handler
EEL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	nothing

Supported execution modes:
enforcing, timeout, polling

CAUTION:
Once started, the FORMAT command must be completed successfully. When RESET discontinues a running FORMAT, the following STARTUP command will fail with status EEL_ERR_POOL_INCONSISTENT. This should enforce the user to re-start the broken FORMAT just to create a consistent and empty EEL-pool in any case.

Code example (enforced mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited FORMAT request */
my_eel_request.command_enu  = EEL_CMD_FORMAT;
my_eel_request.timeout_u08  = 0xFF;
EEL_Execute(&my_eel_request);

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

Code example (timeout mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited FORMAT request */
my_eel_request.command_enu  = EEL_CMD_FORMAT;
my_eel_request.timeout_u08  = 20;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(20);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........

/* periodical timeout count tick */
void isr_tm01(void)
{
  EEL_TimeOut_CountDown();
}
```

Code example (polling mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited FORMAT request */
my_eel_request.command_enu  = EEL_CMD_FORMAT;
my_eel_request.timeout_u08  = 0;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long command not finished */
do{
  EEL_Handler(0);
  CheckCommunicationInterface();
  DoSomethingElse();
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

Figure 4-11 EEL pool after FORMAT (pool complete)

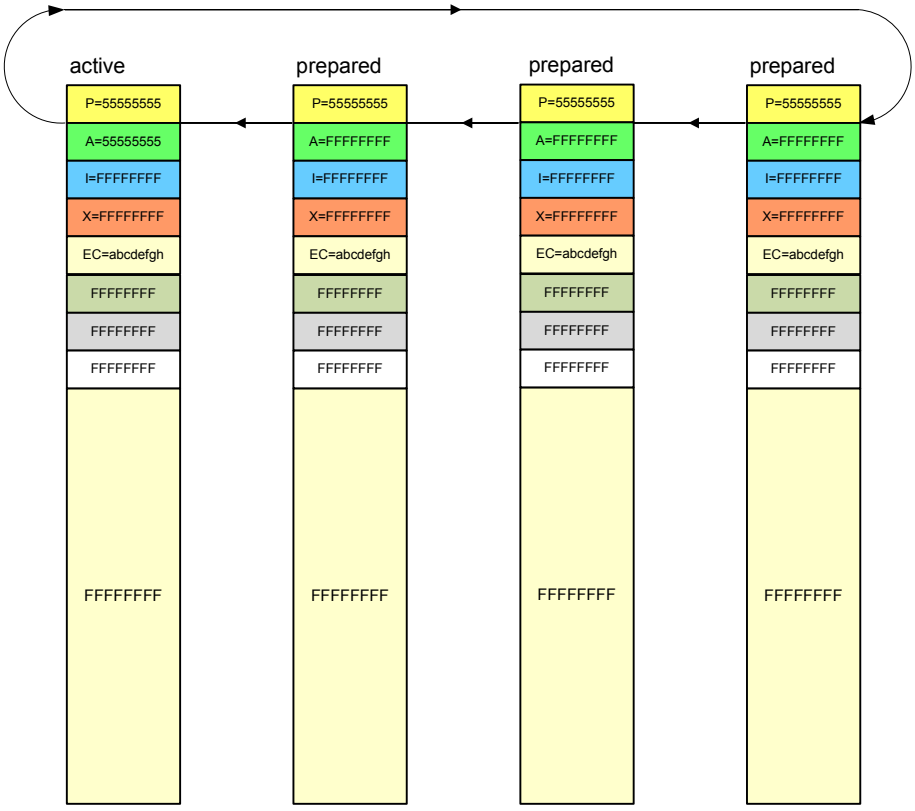
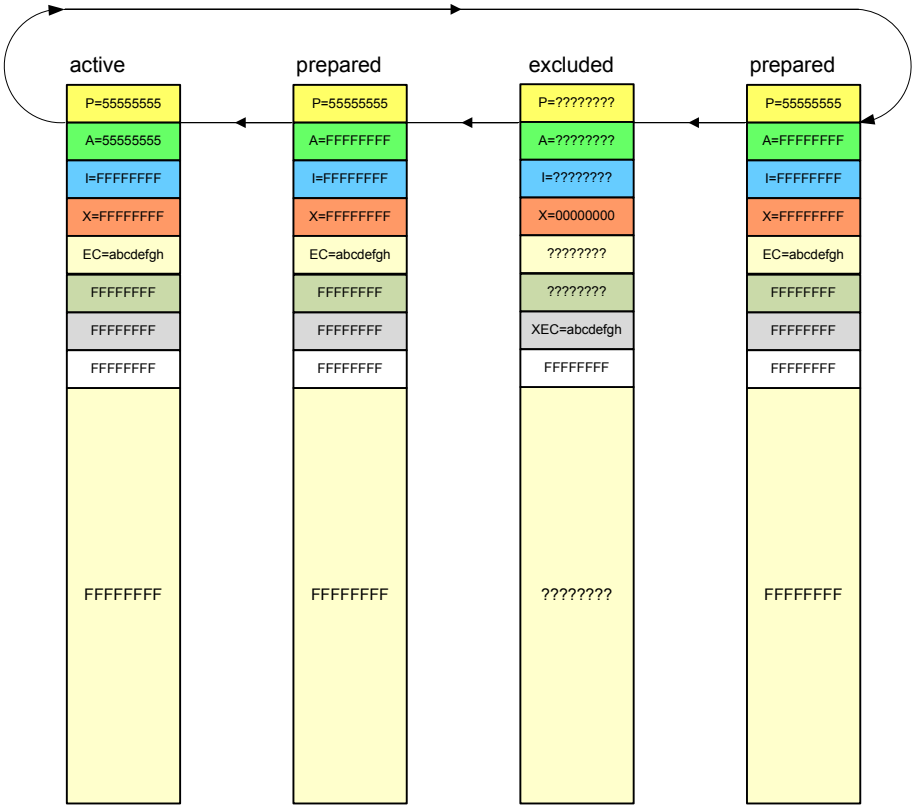


Figure 4-12 4-13 EEL pool after FORMAT (1 block excluded)



Note:

If the third block was already excluded before starting formatting its status remains untouched by the FORMAT command.

4.7.1.4 Command CLEANUP

The cleanup command compresses the acive reqion occupied by data to minimum. The “prepared” region is maximized. Data are not lost in that case. STARTUP is not necessary after CLEANUP for further operation.

Table 9 Status of EEL_CMD_CLEANUP command

Status	Class	Background and Handling	
EEL_ERR_INITIALIZATION	heavy	meaning	EEL not initialized
		reason	wrong handling on user side
		remedy	Initialize EEL before using it
EEL_ERR_COMMAND	light	meaning	invalid command code
		reason	unknown code used in request
		remedy	use eel_command_t type only
EEL_ERR_ACCESS_LOCKED	light	meaning	no access to EEL pool
		reason	STARTUP missing
		remedy	Execute STARTUP
EEL_ERR_POOL_EXHAUSTED	fatal	meaning	EEL pool size smaller < 3 blocks
		reason	to much blocks excluded
		remedy	no remedy, EEL dead
EEL_ERR_INTERNAL	heavy	meaning	unexpected/unknown error code generated in background
		reason	SW bug, EMI, unexpected problems
		remedy	Execute STARTUP. Background status can be analyzed for details.
EEL_ERR_REJECTED	normal	meaning	EEL cannot accept the request
		reason	EEL busy with other request
		remedy	Call EEL_Handler or retry later
EEL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	Call EEL_Handler
EEL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	nothing

Supported execution modes:
enforcing, timeout, polling

Code example (enforced mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
/* specification of a time limited CLEANUP request */
my_eel_request.command_enu  = EEL_CMD_ CLEANUP;
my_eel_request.timeout_u08  = 255;
EEL_Execute(&my_eel_request);

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();
.........
.........
```

Code example (timeout mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited CLEANUP request */
my_eel_request.command_enu  = EEL_CMD_ CLEANUP;
my_eel_request.timeout_u08  = 20;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(20);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu! = EEL_OK)  My_Error_Handler();

.........
.........
.........

/* periodical timeout count tick */
void isr_tm01(void)
{
  EEL_TimeOut_CountDown();
}
```

Code example (polling mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited CLEANUP request */
my_eel_request.command_enu  = EEL_CMD_CLEANUP;
my_eel_request.timeout_u08  = 0;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(0);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

Figure 4-14 EEL pool before CLEANUP command (example)

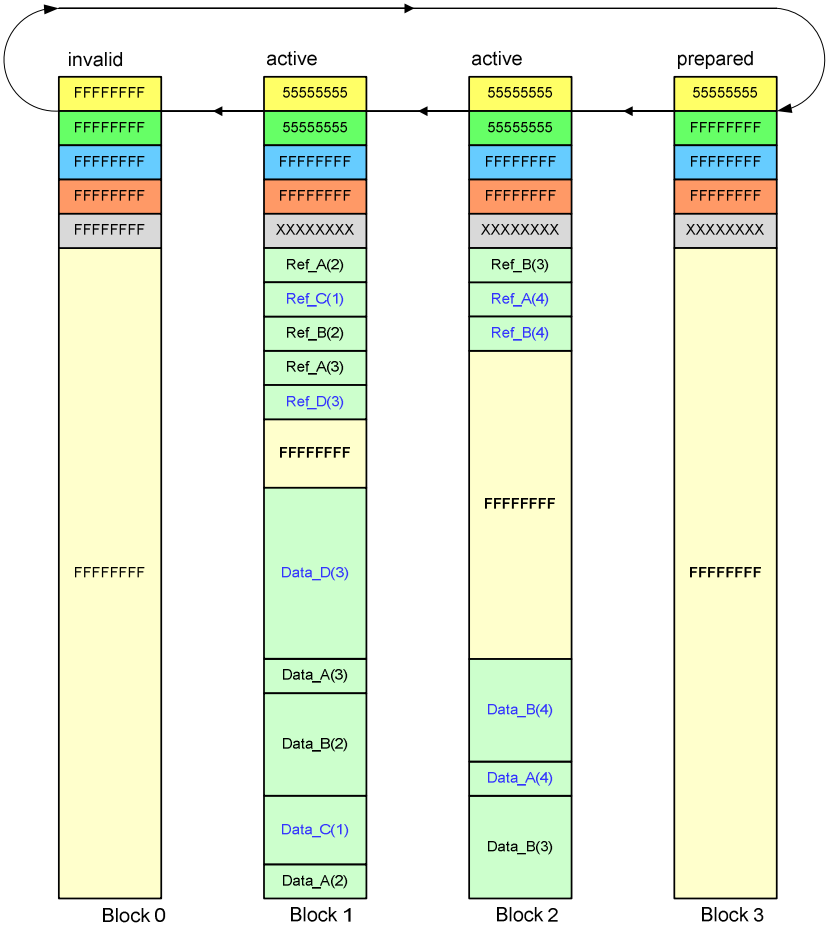
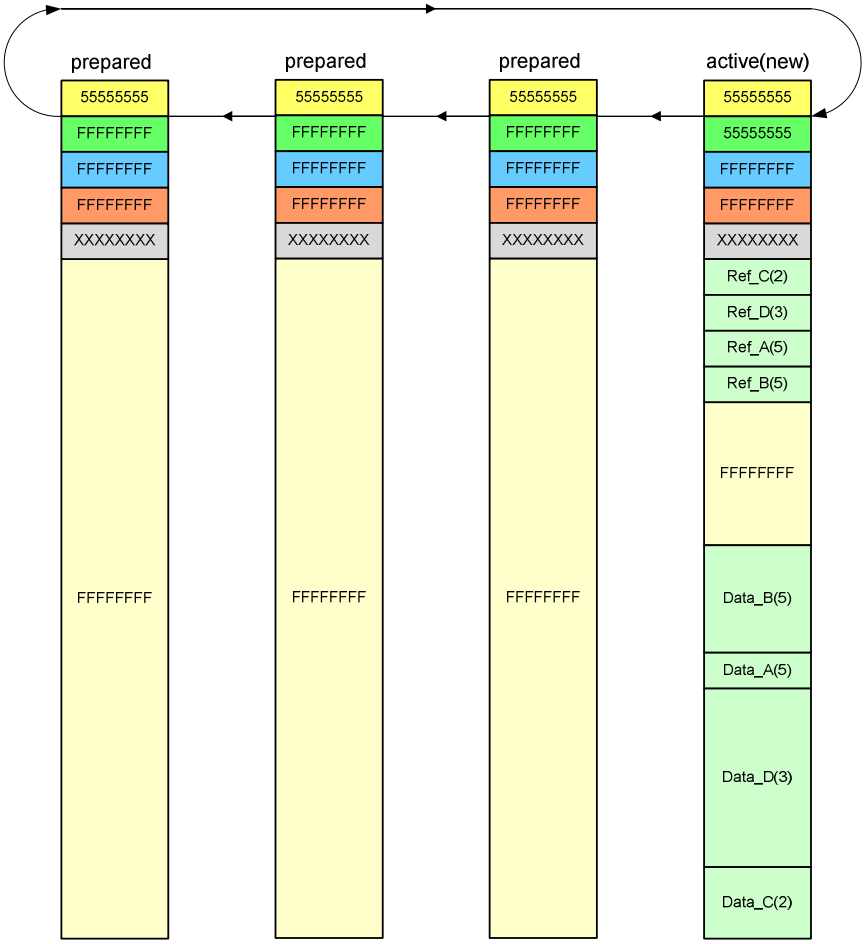


Figure 4-15 EEL pool after CLEANUP command (example)



Note:

Header word marked as XXXXXXXX contain EC, RWPprev, XEC...

4.7.2 Variable oriented commands

EEL variable oriented command can be used by the application to read/write new instnces (values) of the variables registered in the EEL-descriptor.

4.7.2.1 Command WRITE

The write command writes new value of the EEL-variable specified by the identifier.

Table 10 Status of EEL_CMD_WRITE command

Status	Class	Background and Handling	
EEL_ERR_INITIALIZATION	heavy	meaning	EEL not initialized
		reason	wrong handling on user side
		remedy	Initialize EEL before using it
EEL_ERR_ACCESS_LOCKED	light	meaning	no access to EEL pool
		reason	STARTUP missing
		remedy	Execute STARTUP
EEL_ERR_PARAMETER	heavy	meaning	Unknown variable identifier
		reason	Not registered variable ID used
		remedy	Correct or register the variable in the EEL descriptor
EEL_ERR_POOL_EXHAUSTED	fatal	meaning	EEL pool size smaller < 3 blocks
		reason	to much blocks excluded
		remedy	no remedy, EEL dead
EEL_ERR_POOL_FULL	heavy	meaning	no space in pool
		reason	Due to block exclusion not enough space is to cover all variables
		remedy	Execute CLEANUP
EEL_ERR_REJECTED	normal	meaning	EEL cannot accept the request
		reason	EEL busy with other request
		remedy	Call EEL_Handler or retry later
EEL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	Call EEL_Handler
EEL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	none

Supported execution modes:
enforcing, timeout, polling

Code example (enforced mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited WRITE request */
my_eel_request.address_pu08   = (eel_u08*)&my_A_mirror;
my_eel_request.identifier_u08 = 'A';
my_eel_request.command_enu    = EEL_CMD_WRITE;
my_eel_request.timeout_u08    = 255;
EEL_Execute(&my_eel_request);

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

Code example (timeout mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited WRITE request */
my_eel_request.address_pu08   = (eel_u08*)&my_A_mirror;
my_eel_request.identifier_u08 = 'A';
my_eel_request.command_enu    = EEL_CMD_WRITE;
my_eel_request.timeout_u08    = 20;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(20);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........

/* periodical timeout count tick */
void isr_tm01(void)
{
  EEL_TimeOut_CountDown();
}
```

Code example (polling mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited WRITE request */
my_eel_request.address_pu08   = (eel_u08*)&my_A_mirror;
my_eel_request.identifier_u08 = 'A';
my_eel_request.command_enu    = EEL_CMD_WRITE;
my_eel_request.timeout_u08    = 0;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do {
  EEL_Handler(0);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK) My_Error_Handler();

.........
.........
.........
```

Note:

Whenever the application writes data into the EEL-pool the space available in active head may not be sufficient to cover the reference and data of the new instance. To guaranty proper operation in any situation the EEL takes care for sufficient space conditions before writing the instance. This may cause different execution time for writing same portion of data. The user can avoid that situation by offering enough CPU time for the background process that can prepare space in advance.

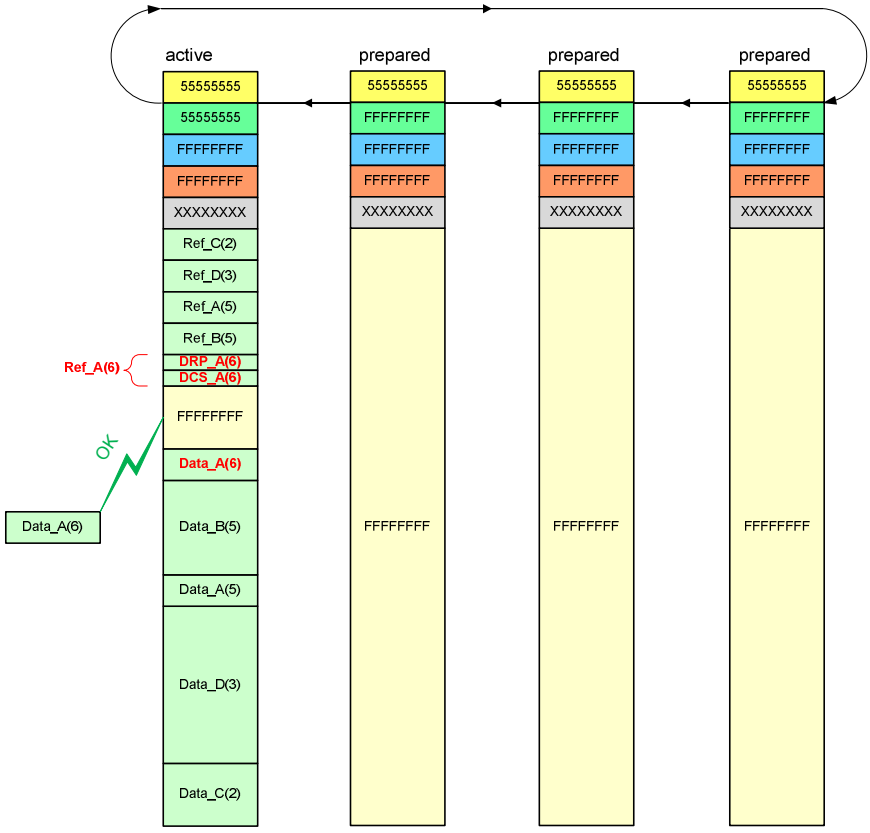
Depending on space precondition in different behavior is possible when writing new instance into the EEL pool. Please have a look to the below examples.

Example 1:
Best case conditions.

Conditions:
a) Enough space available in heading active block to cover the complete instance (reference and data)
b) EEL_REFRESH_BLOCK_THRESHOLD > 1

Sequence:
1) DRP_A(6) is written into flash word addressed by RWP
(allocates space for the new instance in reference- and data-area)
2) Data_A(6) are written word by word into the allocated space in data area.
3) DCS_A(6) is written into the flash word addressed by (RWP+1)
4) RWP, DWP, RAM-reference, and region parameter are updated

Figure 4-16 EEL pool after WRITE command (normal example).



Note:
Data_A(6) means 6’ts instance of the variable “A”

Example 2:

Best case conditions.

Conditions:

a) Not enough space available in heading active block to cover the complete instance (reference and data).

b) more than two blocks are prepared and ready for activation

c) EEL_REFRESH_BLOCK_THRESHOLD > 2

Sequence:

1) After negative space check next block will be activated before write

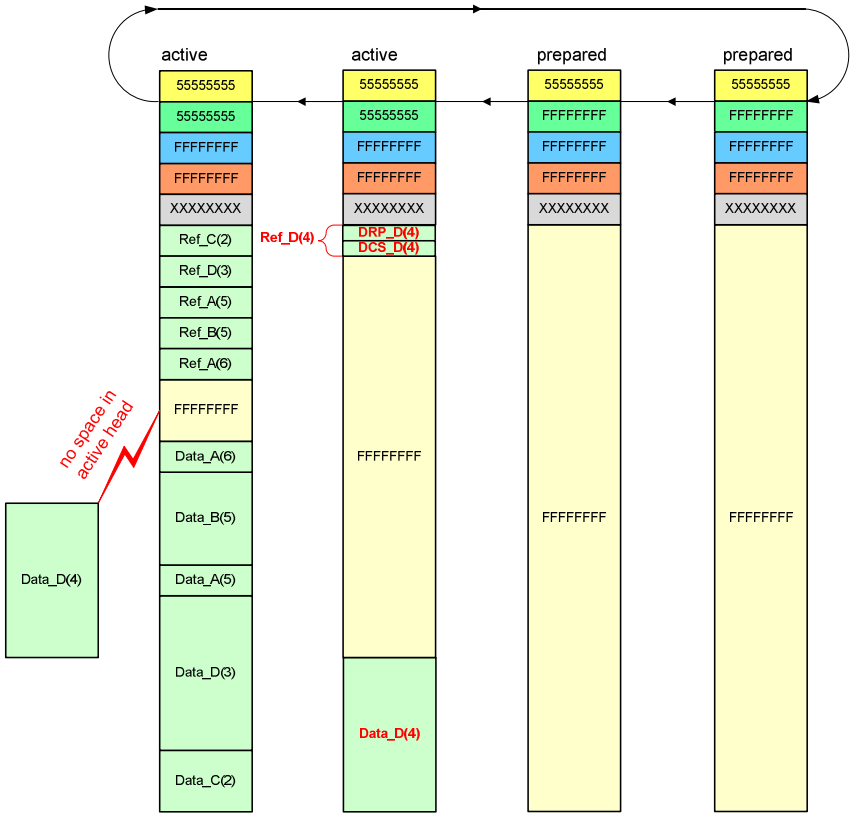
2) DRP_D(4) is written into flash word addressed by RWP
(allocates space for the new instance in reference- and data-area)

3) Data_D(4) are written word by word into the allocated space in data area.

4) DCS_D(4) is written into the flash word addressed by (RWP+1)

5) RWP, DWP, RAM-reference, and region parameter are updated

Figure 4-17 EEL pool after WRITE command (activation example)



Example 3:

Best case conditions.

Conditions:

a) Not enough space available in heading active block to cover the complete instance (reference and data).

b) Not enough prepared for activation

c) EEL_REFRESH_BLOCK_THRESHOLD > 2

Sequence:

1) After negative space check next block should be activated before write
2) Activation not possible (prepared region to small)
3) Execution focus swapped to background for space expansion
4) The background refreshes the last active block C(2) -> C(3)
5) After refresh completion of block 0 will be invalidated and prepared
6) Completed space expansion swaps the execution focus back to foreground
7) DRP_D(5) is written into flash word addressed by RWP
   (allocates space for the new instance in reference- and data-area)
8) Data_D(5) are written word by word into the allocated space in data-area.
9) DCS_D(5) is written into the flash word addressed by (RWP+1)
10) RWP, DWP, RAM-reference, and region parameter are updated

Figure 4-18 EEL pool before WRITE command (expansion example)

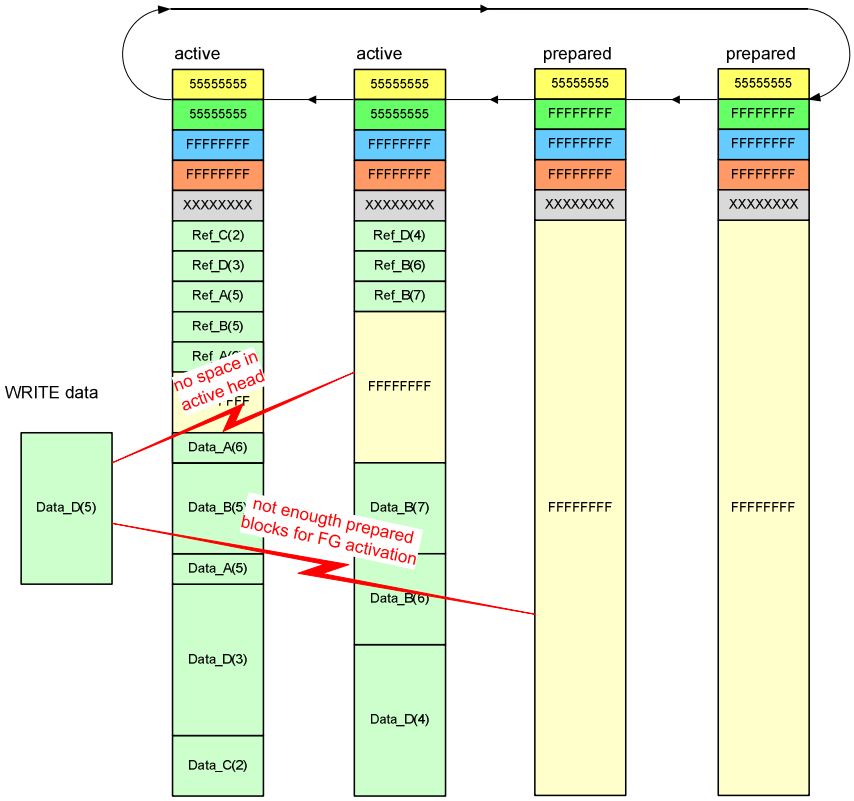
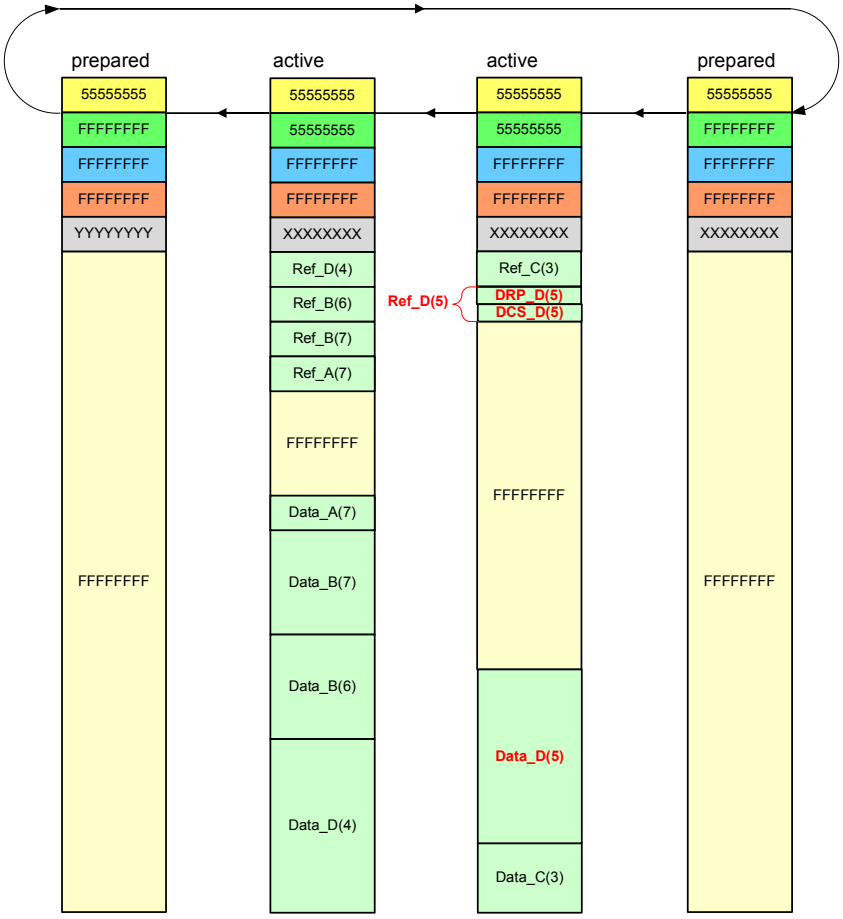


Figure 4-19 EEL pool after WRITE command (expansion example)



The final scenario after completion WRITE(D) is:
- block 0 is prepared after refreshing instance C(2) -> C(3)
- the newest (5'th) instance of D is written into block 2

4.7.2.2 Command READ

The read command copies the actual value of the EEL-variable specified by the identifier into its RAM mirror variable.

When checksum error (DCS) is detected internally during READ execution, the EEL will enforce re-filling the reference table and before reading the next older instance of the specified variable automatically. When no older instance exists, the READ command signalizes EEL_ERR_NO_INSTANCE.

Table 11 Status of EEL_CMD_READ command

Status	Class	Background and Handling	
EEL_ERR_INITIALIZATION	heavy	meaning	EEL not initialized
		reason	wrong handling on user side
		remedy	Initialize EEL before using it
EEL_ERR_ACCESS_LOCKED	light	meaning	no access to EEL pool
		reason	STARTUP missing
		remedy	Execute STARTUP
EEL_ERR_PARAMETER	heavy	meaning	Unknown variable identifier
		reason	Not registered variable ID used
		remedy	Correct or register the variable in the EEL descriptor
EEL_ERR_NO_INSTANCE	light	meaning	no instance of the identifier found
		reason	no initial value written
		remedy	write initial value of the variable
EEL_ERR_REJECTED	normal	meaning	EEL cannot accept the request
		reason	EEL busy with other request
		remedy	Call EEL_Handler or retry later
EEL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	Call EEL_Handler
EEL_OK	normal	meaning	request was finished regular
		reason	no problems during command execution happens
		remedy	none

Supported execution modes:
enforcing, timeout, polling

Code example (enforced mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited READ request */
my_eel_request.address_pu08   = (eel_u08*)&my_A_mirror;
my_eel_request.identifier_u08 = 'A';
my_eel_request.command_enu    = EEL_CMD_READ;
my_eel_request.timeout_u08    = 255;
EEL_Execute(&my_eel_request);

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

Code example (timeout mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* specification of a time limited READ request */
my_eel_request.address_pu08   = (eel_u08*)&my_A_mirror;
my_eel_request.identifier_u08 = 'A';
my_eel_request.command_enu    = EEL_CMD_READ;
my_eel_request.timeout_u08    = 20;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(20);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........

/* periodical timeout count tick */
void isr_tm01(void)
{
  EEL_TimeOut_CountDown();
}
```

Code example (polling mode):

```
/* declaration of the request variable */
eel_request_t my_eel_request;
.........
.........
.........
/* initiation of a READ request */
my_eel_request.address_pu08   = (eel_u08*)&my_A_mirror;
my_eel_request.identifier_u08 = 'A';
my_eel_request.command_enu    = EEL_CMD_READ;
my_eel_request.timeout_u08    = 0;
EEL_Execute(&my_eel_request);
.........
.........
.........

/* execute a state as long not finished */
do{
  EEL_Handler(0);
} while (my_eel_request.status_enu == EEL_BUSY)

if (my_eel_request.status_enu != EEL_OK)  My_Error_Handler();

.........
.........
.........
```

Chapter 5 Characteristics

5.1 Resource consumption

RAM consumption at user side:

High speed RAM:	1 byte
Short address RAM:	9 bytes

ROM consumption:

EEL code size:	6,6 kByte
EEL constant size:	4+(N+1)*4, N = number of EEL variables

Final stack consumption:

FDL and EEL stack:	<120 bytes

EEPROM Emulation Library