NEC

Application Note

EEPROM Emulation

32-/16-bit Single-Chip Microcontroller

EEPROM Emulation Library for embedded Single Voltage Flash

Document No. U18005EE1V2AN00
Date Published May 2008

NOTES FOR CMOS DEVICES

① **VOLTAGE APPLICATION WAVEFORM AT INPUT PIN**
Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (MAX) and V_{IH} (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (MAX) and V_{IH} (MIN).

② **HANDLING OF UNUSED INPUT PINS**
Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

③ **PRECAUTION AGAINST ESD**
A strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

④ **STATUS BEFORE INITIALIZATION**
Power-on does not necessarily define the initial status of a MOS device. Immediately after the power source is turned ON, devices with reset functions have not yet been initialized. Hence, power-on does not guarantee output pin levels, I/O settings or contents of registers. A device is not initialized until the reset signal is received. A reset operation must be executed immediately after power-on for devices with reset functions.

⑤ **POWER ON/OFF SEQUENCE**
In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current.
The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

⑥ **INPUT OF SIGNAL DURING POWER OFF STATE**
Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements.
Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

Introduction

Readers

This application note is intended for users who want to understand the functionality of the EEPROM emulation for devices of the V850 core family.

Purpose

This application note explains the background and handling of the EEPROM emulation on data and on code Flash.

Organization

This application note contains the major sections:

- Emulation strategy
- Emulation library structure
- Emulation library usage
- Emulation library APIs

Legend

Symbols and notation are used as follows:

Weight in data notation	:	Left is high-order column, right is low order column
Active low notation	:	$\overline{\text{xxx}}$ (pin or signal name is over-scored) or /xxx (slash before signal name)
Memory map address:	:	High order at high stage and low order at low stage
Note	:	Explanation of (Note) in the text
Caution	:	Information requiring particular attention
Remark	:	Supplementary explanation to the text
Numeric notation	:	Binary... xxxx or xxxB Decimal... xxxx Hexadecimal... xxxxH or 0x xxxx

[MEMO]

Table of Contents

Chapter 1 Overview

This document describes the usage of the NEC library (*EEELib*) for the EEPROM emulation layer on NEC's microcontrollers featuring EEPROM emulation.
The device features differ depending on the used Flash implementation and basic technology node. Therefore, compile options allow adaptation of the library to the device features and to the application needs.
The libraries are delivered in source code. However it has to be considered carefully to do any changes, as not intended behaviour and programming faults might be the result.
The development environments of the companies Green Hills (GHS), IAR and NEC are supported. Due to the different compiler and assembler features, especially the assembler files differ between the environments. So, the library and application programs are distributed using an installer tool allowing to select the appropriate environment.
For support of other development environments, additional development effort may be necessary. Especially, but maybe not only, the calling conventions to the assembler code and compiler dependent section defines differ significantly.
The libraries are delivered together with device dependent application programs, showing the implementation of the libraries and the usage of the library functions.
The different options of setup and usage of the libraries are explained in detail in this document.

Caution: Please read all chapters of the application note carefully and follow exactly the given sequences and recommendations. This is required in order to make full use of the high level of security and safety provided by the devices

The EEPROM emulation libraries together with demo programs, this application note and device dependent information can be downloaded from the following URL:

http://www.eu.necel.com/updates

1.1 Naming Conventions

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.

EEELib

EEELib is the short form of “EEPROM emulation library”.

DFALib

DFALib is the short form for “Data Flash access library”.

Flash Environment

This is device internal hardware (charge pumps, sequencer,...), required to do any Flash programming. It is not continuously activated, only during Flash modification operations. Activation and deactivation is done implicitly by the *DFALib*.

Chapter 2 Emulation Strategy

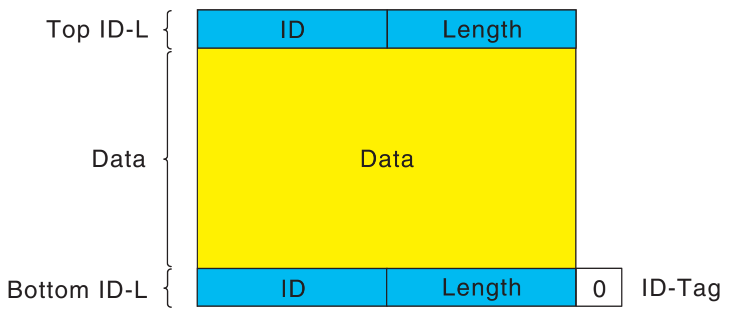
In opposite to classical EEPROMS, where the data is stored on a fixed address and so can always be found on the same location, EEPROM emulation need to store data on changing locations as a kind of data "stack". From the different options to find the data during *Read* operations, the NEC EEPROM emulation uses the way to bundle a certain set of data with one ID, that is then searched in memory on data *Read*.
Writing new data sets is easily done, by appending the data to the data pool.

Furthermore, the write granularity differs between real EEPROM and the data Flash used for EEPROM emulation. While the EEPROM can usually be written in 8-bit or 16-bit units, the data Flash must be written in 32-bit units (+1Bit for the ID-Tag, see below). In case of using code Flash for the emulation, even higher granularity must be considered (64 bits / 128 bits). Please refer to the devices users manual for the write granularity.

2.1 Data Set Representation in the Flash

Due to security reasons that are explained later on, a data set is always embraced by identical ID and length information on the top and on the bottom.
The approach of using IDs to identify a set of data is in-line with AUTOSAR, as well as the ID size of 16 bits and length information of 16 bits.

Figure 2-1: Data Set Representation



Differing from Code Flash with 32-bit width, the currently implemented NEC Data Flash is 33-bit wide. The 33-rd bit is called the ID-Tag, indicating, the a word is information and not part of the data itself. The current implementation in the EEPROM emulation layer sets this on the bottom ID-L, while keeping it cleared on the data bits and on the top ID-L.
Dedicated hardware on the data Flash macro can search for a certain ID by using this ID-Tag.

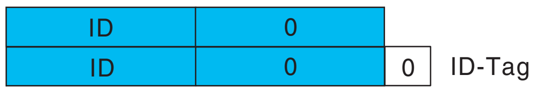
Notes: 1. When doing EEPROM emulation on code Flash, the 33-rd bit is not available and the data set search need to be done inside the *EEELib* by software (100% CPU load)

2. AUTOSAR does not allow the usage of the IDs 0xFFFF and 0x0000 for normal data sets, so that these IDs can be used for special emulation strategy internal purposes.
In the NEC EEPROM emulation strategy the ID 0x0000 indicates invalid data sets (not obsolete data, see below). Invalid data sets are treated library internal and are invisible for the user application.

2.1.1 Invalidation of data (Obsolete data)

Data related to a certain ID can be explicitely invalidated before writing a data set new.
In case of invalidated data a *Read* operation on the data set does not return the latest values, but returns with an error, that the data is obsolete.
The user application can e.g. react on this by taking default values for this data set. Invalidating data is done by writing the concerning data set with the length information 0.
The data set is then represented in Flash as the following:

Figure 2-2: Invalidation Data Set



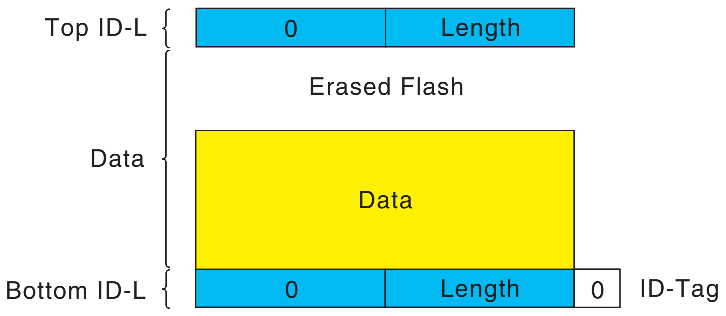
Please refer also to 5.1.3 "Operation initiation" on page 50(*Write* operation).

2.1.2 Data set *Write* operation abort

Data sets are written during a *EEELib Write* or *Refresh* operation (See 2.3 "Emulation Flows - Normal Operation" on page 17). Writing the data sets requires quite some time, depending on the data size. In case of an emergency the time for finishing the data write is not given, but there might be the requirement to write some emergency data (See 2.5.3 "Aborting library operations" on page 33)

The normal *Write* sequence is: Write data --> Write Top ID-L --> Write bottom ID-L.
The data set *Write* operation can be aborted until writing the Top ID-L has started. After that point of time, the operation is finished, as this is faster than the abort procedure.
Abort is done by finishing writing the current data word, then writing an invalid top ID and an invalid bottom ID. This data set is then invalid, but the overall data set structure is still consistent.
This kind of abort is called data encapsulation, as invalid data is encapsulated by ID-L's indicating invalid data.

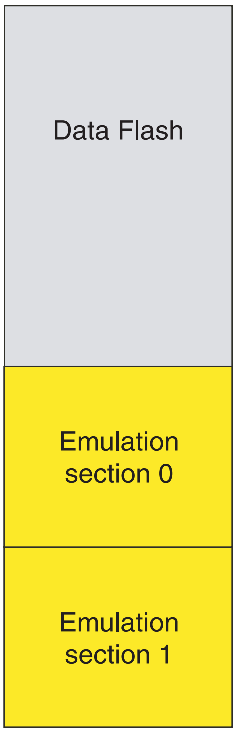
Figure 2-3: Encapsulated Data



2.2 EEROM Emulation Sections

The Flash used for EEPROM emulation contains two emulation sections of an equal size.

Figure 2-4: EEPROM Emulation Sections in the Flash



Notes: 1. An emulation section must always consist of a number of Flash blocks following the rule **NoBlocks = 2^n** with n=0, 1, 2, 3,... ==> NoBlocks = 1, 2, 4, 8,...

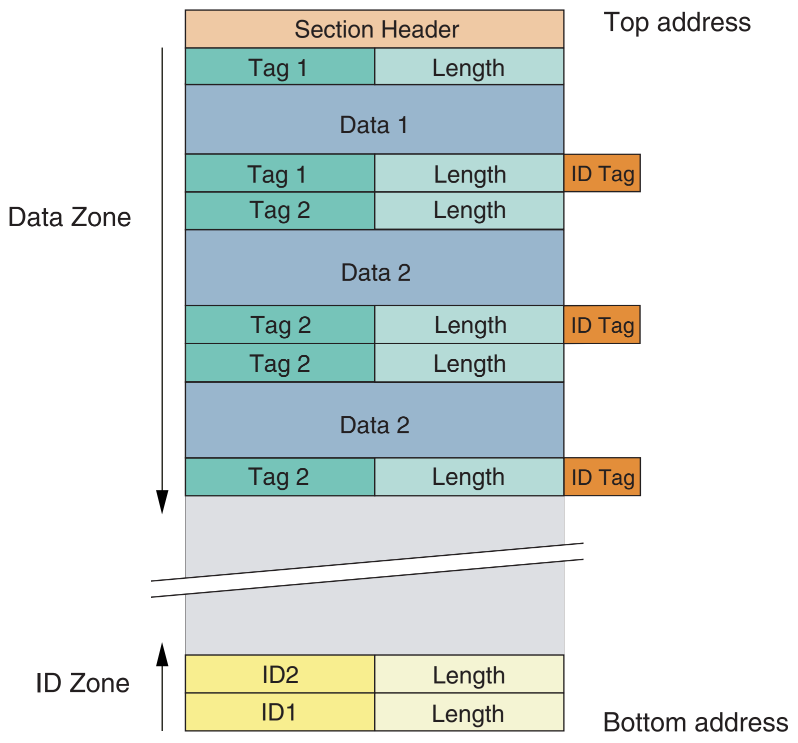
2. Furthermore, the section must start on a block number following the rule: **StartBlock = NoBlocks * n** with n=0, 1, 2, 3,...

2.2.1 Overview

Each emulation section consists of 3 parts:

- Section Header
- Data Zone
- ID Zone

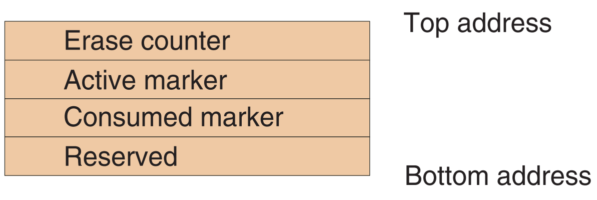
Figure 2-5: EEPROM Emulation Section Overview



2.2.2 Section header

Section header size is 16 Bytes (4 Words), where currently 3 words are used to define the current section status.

Figure 2-6: Section Header



The erase counter word consists of the 16-bit counter and a 16-bit inverse value for protection. It describes the number of erase cycles on the data section.

When set, the active marker or the consumed marker have the value 0x55555555. The value of a marker which is cleared is 0xFFFFFFFF (Blank Word)

The following section states are possible:

- Prepared:
  The section has been erased
  In the section header the erase counter is written. Active marker and consumed marker are cleared
  This section is ready for activation by a *Refresh* operation.
- Active:
  The section contains the latest data sets. The EEPROM emulation can read and write data to this section.
  In the section header additionally to the erase counter also the active marker is set. Consumed marker is cleared.
- Consumed:
  The section is full, no more space is available to *Write* new data sets. After activating the next section and copying the latest data sets there, the *Refresh* operation markes the full section consumed
  In the section header additionally to the erase counter and active marker, also the consumed marker is set.
  Next step is to *Prepare* the section.
- Invalid:
  This is no valid state. During operation it can only occur in case of interruption of a Flash operation like erase to this section.
  The section header may contain any other data except as described in the three states before.
  Also a completely erased Flash (default factory delivery state) is invalid from EEPROM emulation point of view.

The normal section state transitions are:

Prepared --> Active --> Consumed --> Prepared -->...

2.2.3 ID zone

The ID zone is a list of IDs of the data sets written to the section so far. The ID-list is dynamically generated during writing new data sets into the section (See 2.3.2 “Write data set” on page 18).
An ID-List is required in order to simplify and to speed up *Refresh* operations (copying data sets from a full data section to a prepared one). The solution to keep the ID-list dynamically in the EEPROM Emulation Flash space instead of keeping it statically in Code Flash area has been chosen in order to allow new/added applications to store data sets with new IDs.
The ID zone grows by time in case of writing new data sets with new IDs.

2.2.4 Data zone

The data zone contains the data sets. New data sets are simply appended after the last written data set. So the data zone grows down in the address space.
The section is full and a *Refresh* is required, when there is no more enough erased space between ID zone and data zone for the data set to be written.
In order to do a *Refresh*, when the free Data Zone space is under a certain limit, a service function of the library allows to check the remaining space (See 5.2.1 “Check free section space” on page 55).

2.3 Emulation Flows - Normal Operation

For details regarding the operation execution and calling parameters please refer to Chapter 5 “Emulation Library - API” on page 47.

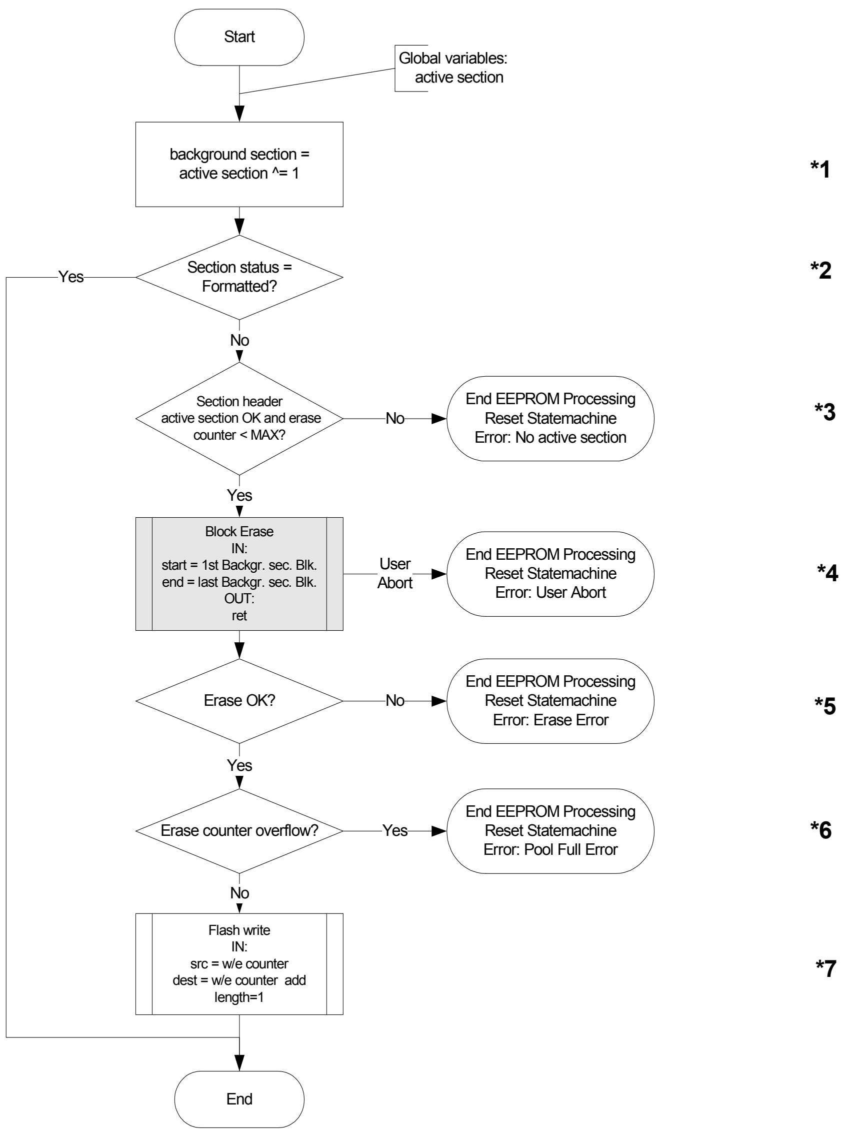
2.3.1 Prepare

The *Prepare* operation checks, whether the background section (not active section) is already prepared. If yes it returns immediately, if not it does a section erase and write the erase counter as indication, that the section is now prepared.

Note: As the *Prepare* operation returns immediately without Flash operation, when the section is already prepared, the *Prepare* operation can be called frequently, e.g. time driven.

Note: The erase counter is increased on each preparation of section 0. The number of erase cycles is specified as max. 10.000 for Data Flash. This may not be exceeded and should ensured by the user function. Independant from that, the PREPARE operation returns the error EEE_ERR_POOLFULL on erase counter overflow (Exceeding 0xFFF1) or if the counter is invalid.

Figure 2-7: Prepare Operation Flow

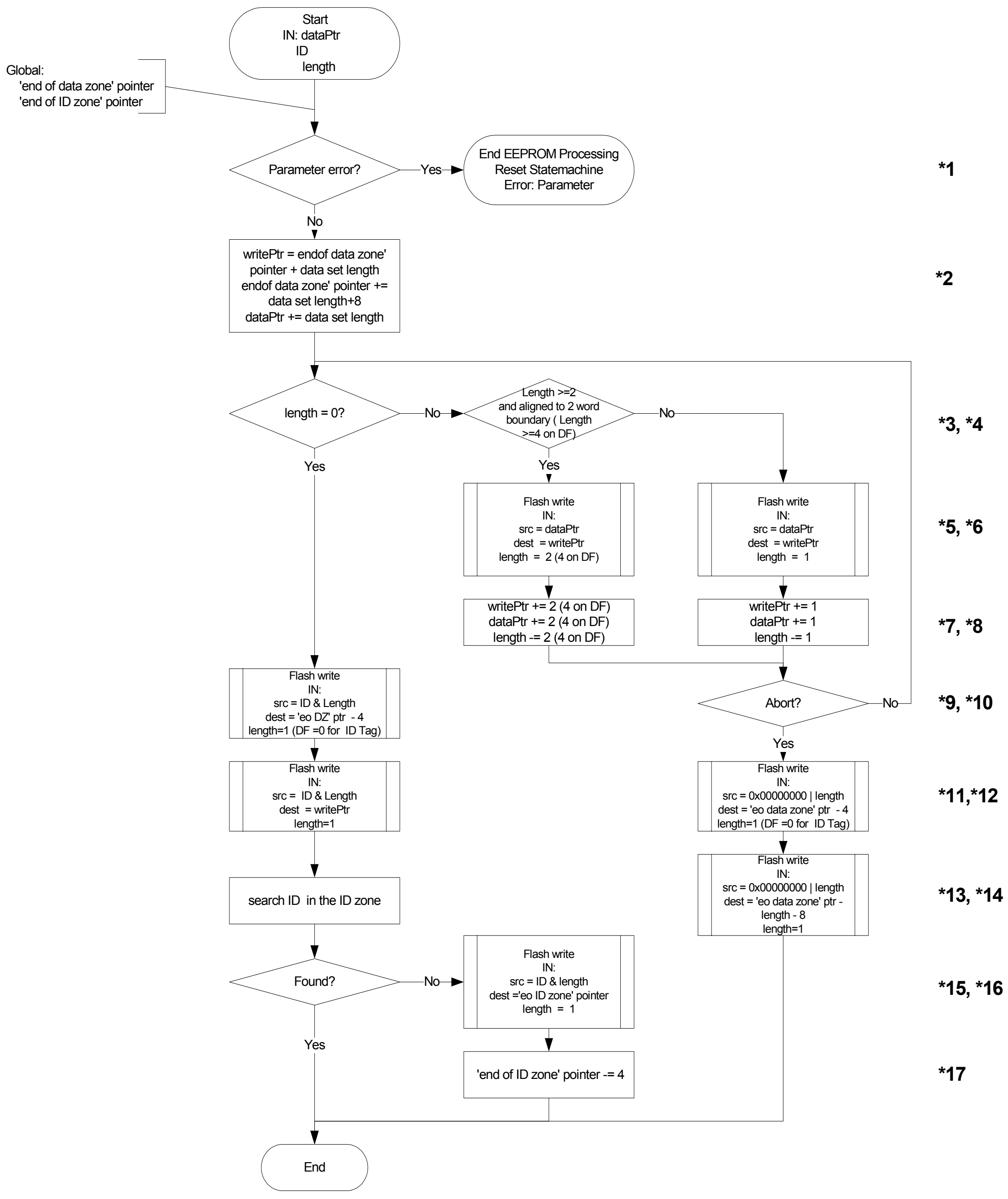

Start
Global variables: active section
background section = active section ^= 1
*1
Section status = Formatted?
Yes
*2
No
Section header active section OK and erase counter < MAX?
No
End EEPROM Processing Reset Statemachine Error: No active section
*3
Yes
Block Erase IN: start = 1st Backgr. sec. Blk. end = last Backgr. sec. Blk. OUT: ret
User Abort
End EEPROM Processing Reset Statemachine Error: User Abort
*4
Erase OK?
No
End EEPROM Processing Reset Statemachine Error: Erase Error
*5
Yes
Erase counter overflow?
Yes
End EEPROM Processing Reset Statemachine Error: Pool Full Error
*6
No
Flash write IN: src = w/e counter dest = w/e counter add length=1
*7
End

2.3.2 Write data set

The data set *Write* operation writes a complete data set to the active section.
The data is written first, followed by the top ID-L and the bottom ID-L. Last the ID zone is checked whether the ID-L is already contained. If not, the ID-L is added to the ID zone.

Note: It is not possible to directly write a data set with a length differing from the initial data set length (Exception lenght 0 for obsolete data). If due to an application update the data set length shall be modified, remove the data set ID from the emulation by the operation *Refresh_EDS* and then write the data set with the new length.

Figure 2-8: Write Data Set Operation Flow

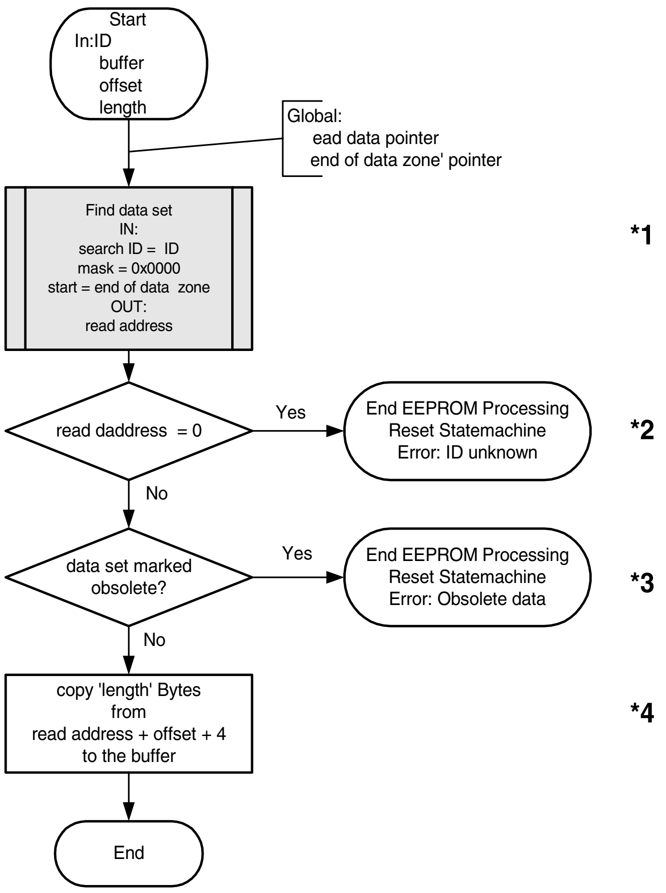


Special options are the write abort and invalidation data set.

2.3.3 Read

The *Read* operation searches the current data set according to the ID in the function call parameter and copies the data set to the destination location.
In some cases it may make sense to read only parts of bigger data sets in order to save read buffer space. Therefore, the *Read* operation has a parameter to define an offset within the data set and a length parameter to define the number of bytes to be read.

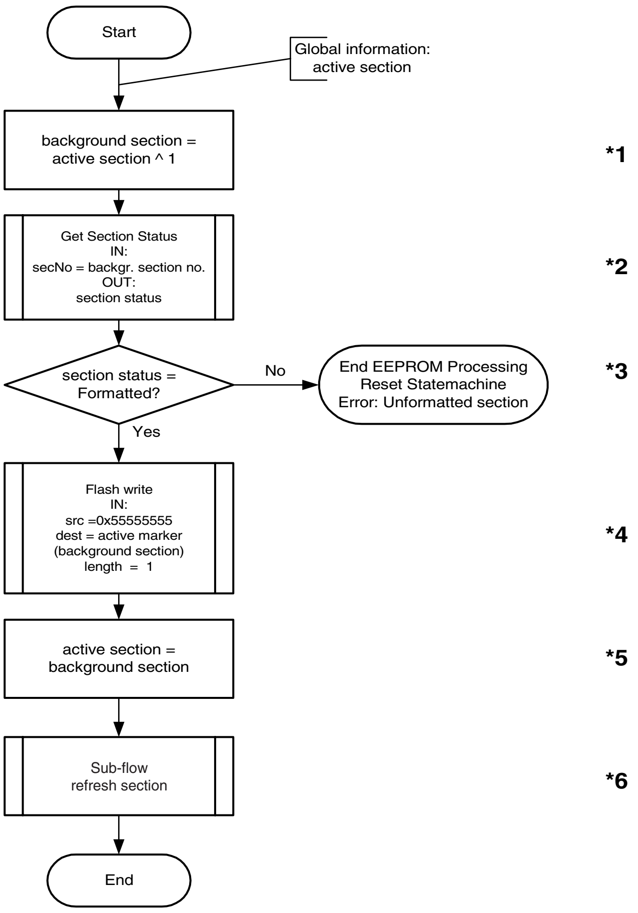
Figure 2-9: Read Operation Flow



2.3.4 Refresh

The *Refresh* operation is used to copy the latest data sets of all valid IDs from a full data section to a prepared data section and to activate this section for subsequent *Read* and *Write* operations.

Figure 2-10: Refresh Operation Flow



The *Refresh* flow activates the new section first before copying the data sets from the full section to the new section. Background to first activate the section is, that the *Refresh* operation can be interrupted for writing emergency data. The emergency *Write* operation is a normal *Write* operation, that writes to the active section. As the old section is full, the data need to be written to the new section.
In case of an interruption of the *Refresh*, it is completed after reset by the *Startup* procedure.

Note: In case of aborting a *Refresh* for writing the emergency data, also a complete restart of the Emulation is required (starting with the function *EEELib_Init)* so that the *Startup* procedure can resume the *Refresh.*

The *Refresh* section flow (next figure) is a part of the *Refresh* operation. It parses the list of IDs in the old sections ID zone.
If an ID is already contained in the new section ID-list, the concerning latest data set of the old section is not copied to the new section as a newer data set is already contained in the new section. This case may happen, on resuming a *Refresh* operation, that has been interrupted for emergency data *Write* or by power fail.
If an ID is not contained in the ID zone of the new section, the latest data set instance belonging to this ID is searched and then copied to the new section.

Robustness features in *Refresh* section flow:

- If the consistency of the latest data set instance's IDL is not given (E.g. ID zone entry differs from data set ID-L), the Refresh skips this data set and returns the error EEE_ERR_SECTION_FIXED (This error cannot occure on normal program flow incl. power fails, but only on application faults e.g. stack overflow, ...)

- Robustness against section overflow. If the REFRESH cannot be finished due to section overflow, the error EEE_ERR_SECTION_OVERFLOW is returned and the new section is marked consumed This problem might occure in two cases:
  - After a *Refresh* abort too many data sets are written by the user application, so that the Refresh resume during *Startup-2* cannot complete.
  - *Startup-2* is interrupted too often when trying to resume the *Refresh*. Then the section is filled up with partially written data sets, that are overwritten on next *Startup-2*. Finally *Refresh* during this *Startup-2* cannot resume

  So, *Refresh* during normal operation will not return this error, but only *Startup-2* can do, based on this check during *Refresh.*

  Note: If after *Refresh* abort new emergency data was written to the new active section, this data is lost as the elder section without this data remains active

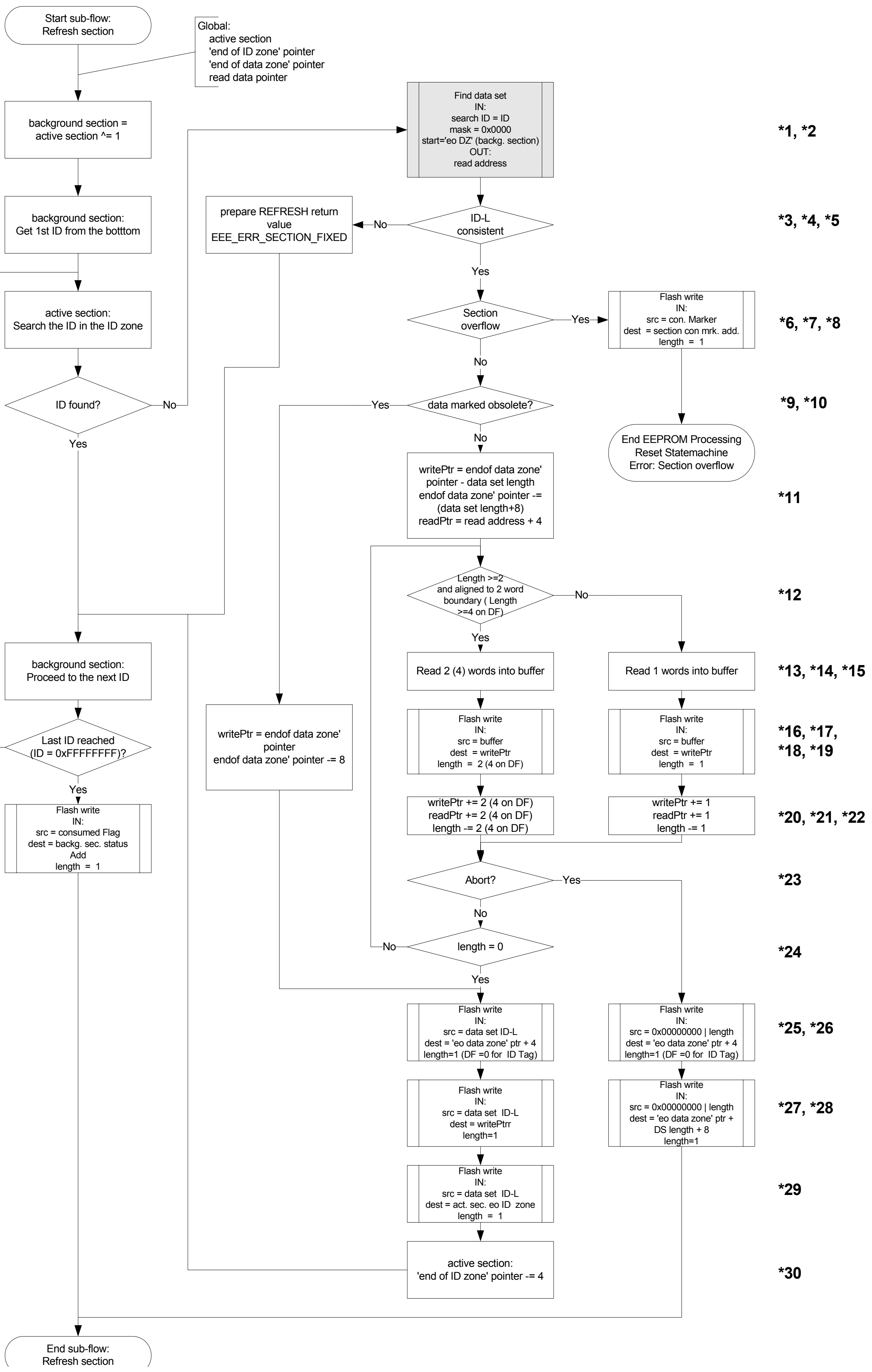
2.3.5 Refresh-EDS

Refresh-EDS allows to exclude data set IDs from the emulation. To do so, a zero terminated list is handed over to the operation which contains the data set IDs which shall not be copied into the new active section. After finishing the operation, these IDs are no longer part of the emulation.

Background of the operation is, that the length of data sets cannot be changed if set once by initially writing the data set. If in case of an application update the data set length shall change, it must be removed from the emuation before it can be written new with the new length.

This operation is a special kind of *Refresh*. In fact the same flow/code is executed, so no separate flow chart is required.
For *Refresh-EDS* the same conditions apply as for *Refresh*.

Figure 2-11: Refresh Section Sub-Flow


Start sub-flow: Refresh section
Global: active section 'end of ID zone' pointer 'end of data zone' pointer read data pointer
background section = active section ^= 1
background section: Get 1st ID from the botttom
active section: Search the ID in the ID zone
ID found?
No
Yes
Find data set IN: search ID = ID mask = 0x0000 start='eo DZ' (backg. section) OUT: read address
*1, *2
ID-L consistent
No
prepare REFRESH return value EEE_ERR_SECTION_FIXED
*3, *4, *5
Yes
Section overflow
Yes
Flash write IN: src = con. Marker dest = section con mrk. add. length = 1
*6, *7, *8
No
data marked obsolete?
Yes
*9, *10
End EEPROM Processing Reset Statemachine Error: Section overflow
No
writePtr = endof data zone' pointer - data set length endof data zone' pointer -= (data set length+8) readPtr = read address + 4
*11
Length >=2 and aligned to 2 word boundary (Length >=4 on DF)
No
*12
Yes
Read 2 (4) words into buffer
Read 1 words into buffer
*13, *14, *15
background section: Proceed to the next ID
writePtr = endof data zone' pointer endof data zone' pointer -= 8
Last ID reached (ID = 0xFFFFFFFF)?
No
Yes
Flash write IN: src = buffer dest = writePtr length = 2 (4 on DF)
Flash write IN: src = buffer dest = writePtr length = 1
*16, *17, *18, *19
Flash write IN: src = consumed Flag dest = backg. sec. status Add length = 1
writePtr += 2 (4 on DF) readPtr += 2 (4 on DF) length -= 2 (4 on DF)
writePtr += 1 readPtr += 1 length -= 1
*20, *21, *22
Abort?
Yes
*23
No
length = 0
No
*24
Yes
Flash write IN: src = data set ID-L dest = 'eo data zone' ptr + 4 length=1 (DF =0 for ID Tag)
Flash write IN: src = 0x00000000 | length dest = 'eo data zone' ptr + 4 length=1 (DF =0 for ID Tag)
*25, *26
Flash write IN: src = data set ID-L dest = writePtrr length=1
Flash write IN: src = 0x00000000 | length dest = 'eo data zone' ptr + DS length + 8 length=1
*27, *28
Flash write IN: src = data set ID-L dest = act. sec. eo ID zone length = 1
*29
active section: 'end of ID zone' pointer -= 4
*30
End sub-flow: Refresh section

2.3.6 Format

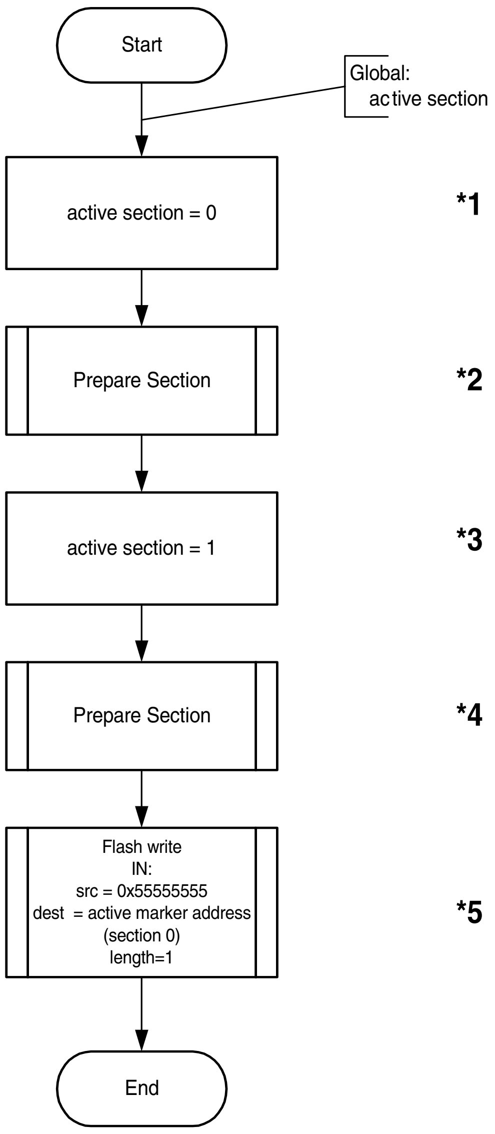
The *Format* operation prepares both sections and activates one of them for data set *Read* and *Write* operations. This operation can be used for completely erased data as well as already used Flash. After the *Format* operation all data sets are deleted. This operation can be used, if the application itself prepares an erased Data Flash for usage.

Alternatives to using the *Format* operation in the application are using a Debugger (e.g. GHS) or a Flash programmer (e.g. the NEC PG-FP4) to format the data Flash and to copy initial data sets into the sections.

Note: The *Format* operation is not needed in the normal application operation.

Note: The erase counter is increased on each *Format* operation. The number of erase cycles is specified as max. 10.000 for Data Flash. This may not be exceeded and should ensured by the user application. Independant from that, the FORMAT operation resets the erase counter to 1 in case of an overflow (>0xFFF1) or if the erase counter is inconsistent (E.g. on completely erased Flash).

Figure 2-12: Format Operation Flow



2.4 Emulation Flows - Initialization Phase

The initialization operations *Startup-1* and *Startup-2* initialize the EEPROM emulation library and ensure the data sections consistency.

Major failure mode in the EEPROM emulation is the interruption of an EEPROM emulation operation, mainly by a power fail. Depending on the point of interruption various possibilities of inconsistent data sections, data sets or ID zone entries exist.
According to a FMEA (Failure Modes and Effects Analysis) done on the EEPROM emulation flows, the *Startup* operations check the identified possible failures and fix the sections and data set consistency.

As the *Startup* requires quite some time, but EEPROM data may be required before, the operation has been split into the phases *Startup-1* and *Startup-2*. *Startup-1* run-time is relatively short (usually <10ms) but require 100% CPU load. It can be executed in the device startup phase. After *Startup-1* data sets can be *Read* (See below)
Startup-2 requires longer time, but less CPU load. It may be executed in the device startup phase, or when the device is in full operation. Due to the shorter handler run-time, execution in a task may be possible.

2.4.1 Startup-1

The *Startup-1* flow is the first part of the *EEELib* initialisation. It executes the Ensure Consistency - Step1 flow.

The flow checks, which section is active. In case of two active sections it checks, which section contains valid data sets. Furthermore, library internal pointers are initialized.
After *Startup-1* reading data sets is possible

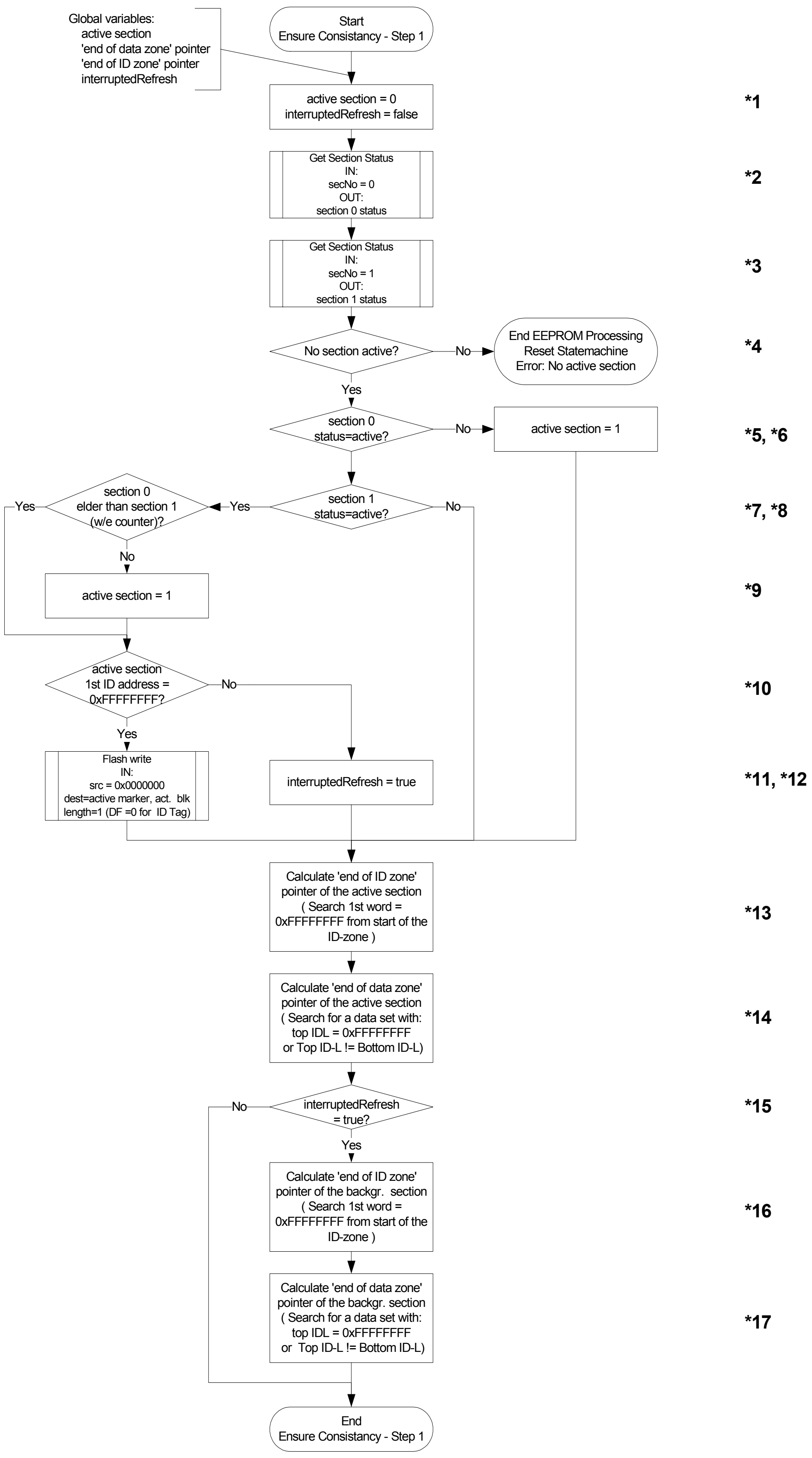
Notes: **1.** Differing from the state of a fully initialized *EEELib*, in this phase searching data sets in the *Read* operation is done by the CPU, not by the Flash hardware. Following that, the CPU is loaded with 100% during the search operation

2. It cannot be ensured, that emergency data (See 2.5.3 “Aborting library operations” on page 33) are found in this phase. For emergency data *Read* the *EEELib* must be initialized completely (See 2.5.1 “Data Flash initialisation” on page 30)

The *Startup-1* executes the ensure consistancy - step 1 flow. It first checks, which section is the active section (Flow steps *1 ~ *9).

In case of a previously interrupted *Refresh* operation, most probably both sections are active, as at *Refresh* startup the new section is activated and at *Refresh* end the old section is deactivated (marked consumed).
In that case of *Refresh* interruption before completion of writing the 1st data set into the new section, the new section contains no relevant data and is marked invalid (Flow steps *10, *11). So empty sections need not be considered on the later *Startup* flow.
The “interruptedRefresh” flag is set for the later flow, if the new section is not empty.
Steps *13~*17 initialize important variables pointing to the end of the ID zone and the end of the data zone of the active section and if later on required also the pointers of the background section.

Figure 2-13: Ensure Consistency - Step 1 Flow


Global variables:
active section
'end of data zone' pointer
'end of ID zone' pointer
interruptedRefresh
Start Ensure Consistancy - Step 1
active section = 0 interruptedRefresh = false
Get Section Status IN: secNo = 0 OUT: section 0 status
Get Section Status IN: secNo = 1 OUT: section 1 status
No section active?
No
End EEPROM Processing Reset Statemachine Error: No active section
Yes
section 0 status=active?
No
active section = 1
section 1 status=active?
Yes
No
section 0 elder than section 1 (w/e counter)?
Yes
No
active section = 1
active section 1st ID address = 0xFFFFFFFF?
No
Yes
Flash write IN: src = 0x0000000 dest=active marker, act. blk length=1 (DF =0 for ID Tag)
interruptedRefresh = true
Calculate 'end of ID zone' pointer of the active section (Search 1st word = 0xFFFFFFFF from start of the ID-zone)
Calculate 'end of data zone' pointer of the active section (Search for a data set with: top IDL = 0xFFFFFFFF or Top ID-L != Bottom ID-L)
interruptedRefresh = true?
No
Yes
Calculate 'end of ID zone' pointer of the backgr. section (Search 1st word = 0xFFFFFFFF from start of the ID-zone)
Calculate 'end of data zone' pointer of the backgr. section (Search for a data set with: top IDL = 0xFFFFFFFF or Top ID-L != Bottom ID-L)
End Ensure Consistancy - Step 1
*1
*2
*3
*4
*5, *6
*7, *8
*9
*10
*11, *12
*13
*14
*15
*16
*17

2.4.2 Startup-2

The *Startup-2* flow is the second part of the *EEELib* initialisation. It executes the Ensure Consistency - Step 2 flow, which contains the sub-flows Ensure Section Consistency and Ensure ID Consistency.

Ensure Consistency - Step 2

If Ensure Consistency - Step 1 detected an interrupted *Refresh*, Ensure Consistency - Step 2 first switches the active section and adapts the internal variables accordingly. This is done in the steps *2~*3.

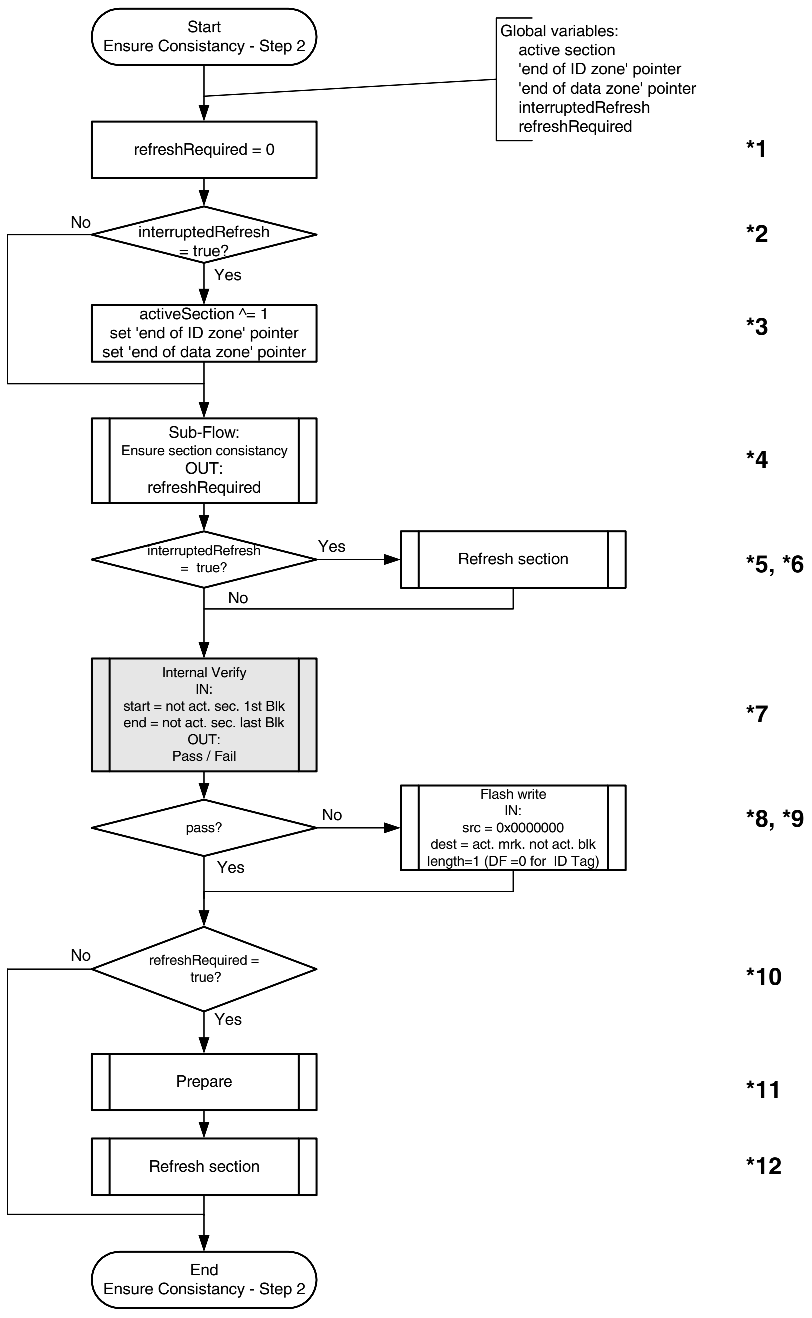
Next is to ensure the consistency of the active section. This is done in the later explained sub-flow Ensure Section Consistency. If the section consistency has been repaired, depending on the necessary measures, a *Refresh* of the active section may be necessary. In this case the flag “refreshRequired” is set within that sub-flow.

In case of two active sections next the interrupted *Refresh* need to be completed (steps *5, *6).

The steps *7~*9 check the read margin of the background section and mark the background section invalid in case of not sufficient margin. Then no activation of the background section is possible without a *Prepare*, that erases and prepares the section for activation.

If the “refreshRequired” flag is set, a *Refresh* operation or a *Prepare* and *Refresh* operation completes the *Startup* flow in order to ensure the read margin of the active section (steps *10 ~ *12).

Figure 2-14: Ensure Consistency - Step 2 Flow



Ensure Section Consistency

At the beginning of the section consistency check, the read margin of the section is checked. Insufficient read margin may be caused by an interrupted *Write* operation due to power fail. Then cells are written over the read margin, but below the margin required for the specified data retention.
In case of insufficient read margin, the “refreshRequired” flag is set (Steps *1 ~ *4).

In case of interrupted data set writing in a preceding *Write* or *Refresh* operation, it is possible, that invalid data is written between the end of the valid data zone and the end of the ID zone.
To check this, the area is checked backwards by blank check backwards. This is done in the steps
*5 ~ *14.

In case of invalid data, the complete range from the end of the data zone up to the last non blank word is over written with the value 0x00000000, which is interpreted by a later *Refresh* operation as data to be ignored. Over-writing is done in the steps *15, *17, *18, *20.

In case of invalid data in the data zone, an interruption while writing the data has to be assumed. In this case a write operation into the ID zone cannot have been interrupted and so the ID zone need not be checked.
Otherwise the ID zone need to be checked for consistency. This is done in the steps *16, *19.

Figure 2-15: Ensure Section Consistency Sub-Flow

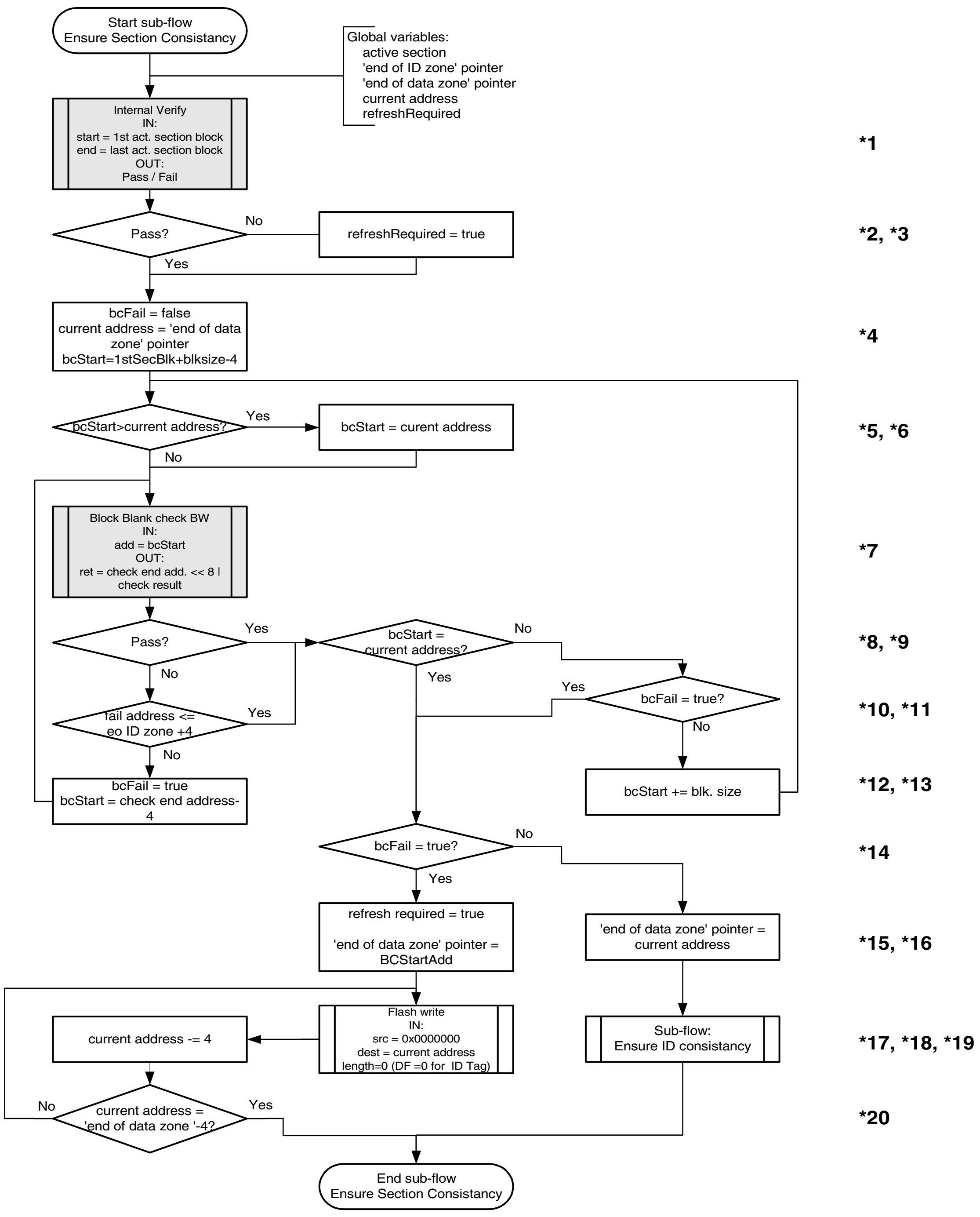
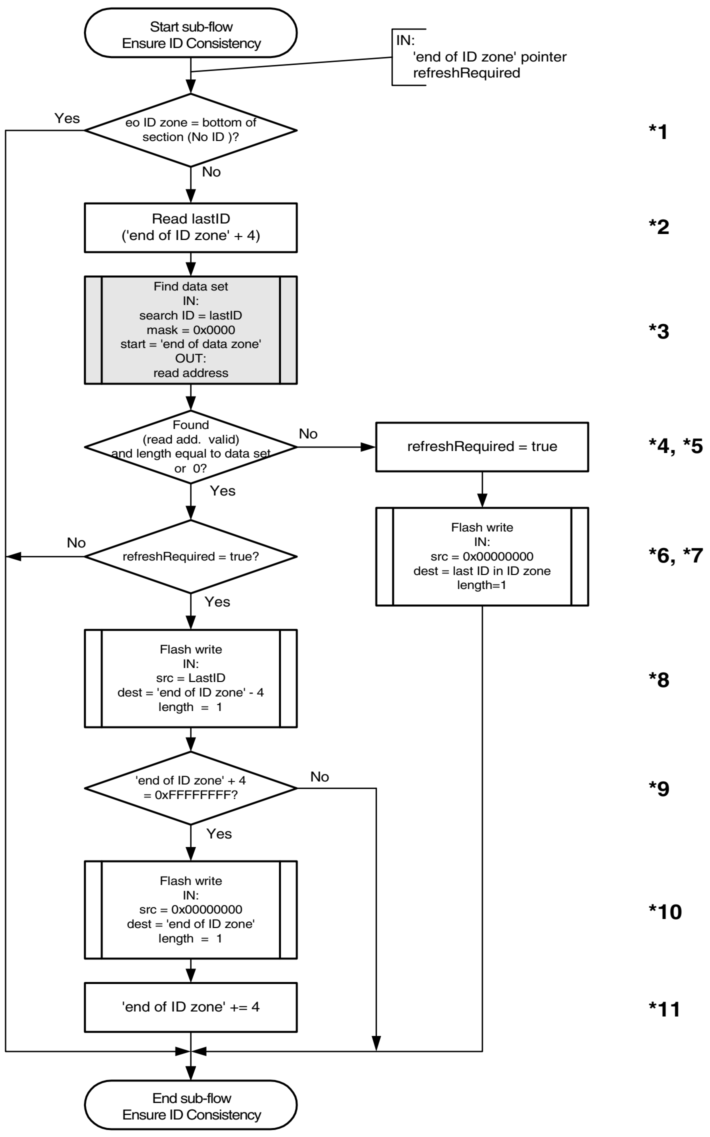


Figure 2-16: Ensure ID Consistency Sub-Flow



Ensure ID Consistency

In case of no ID entry, the flow ends immediately, otherwise the data set for the last entry in the ID zone is searched. If not found, a previous ID-L write on this address has been interrupted and the ID-L entry is not valid. It is over written with 0x00000000. This is done in the steps *2 ~ *5, *7.

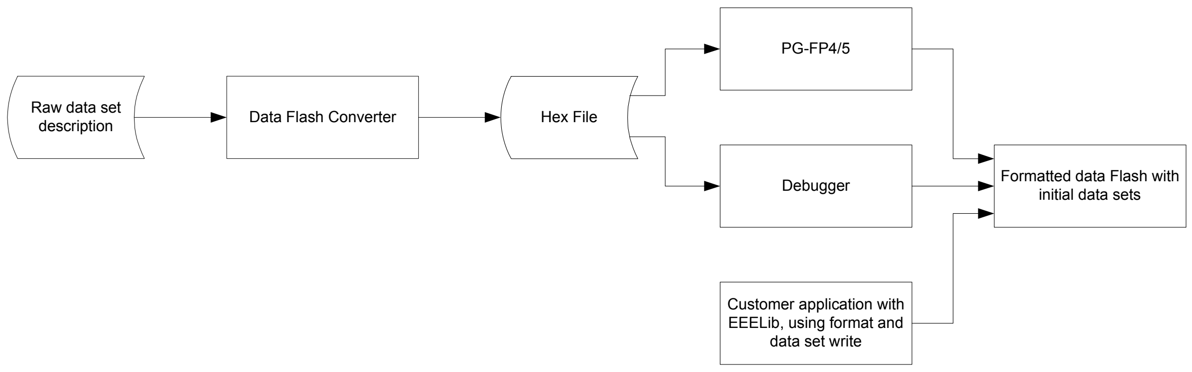
If a data set has been found, the last entry is valid. If the flag “refreshRequired” is not set, the section has no data without sufficient read margin and the flow ends.
If the flag is set, there is a chance, that either the last entry has not sufficient read margin or the word after this ID-L entry (Should be erased) is not completely blank but below the read level due to interrupted write operations. So the last entry is rewritten with its own value to increase the read margin and the word after is over written with 0x00000000.

2.5 EEPROM Emulation Library Usage

2.5.1 Data Flash initialisation

Before being able to use the data Flash for EEPROM emulation, the Flash must be formatted and (depending on the application) be filled with initial data sets. Formatting can be done by the *Format* operation of the *EEELib* (See 2.3.6 "Format" on page 23) or by a debugger (e.g. GHS) or a Flash programmer (e.g. NEC PG-FP4).
A tool called "Data Flash Converter" to convert raw data set information into a data format usable for the debugger or Flash programmer is available and can be loaded from the interrnet on the same page, where the *EEELib* can be requested.

Figure 2-17: Options for Data Flash Initialisation


Raw data set description
Data Flash Converter
Hex File
PG-FP4/5
Debugger
Customer application with EEELib, using format and data set write
Formatted data Flash with initial data sets

2.5.2 EEPROM emulation in the application

(1) Initialisation phase

Before being in normal operation, the *EEELib* has to sequentially pass dedicated initialisation phases with different conditions and possible operations.

In the explanation below two different device operational phases are distinguished. This is application dependent. However, most applications can be separated into these phases:
- Device initialisation phase
  The device hardware and application software is beeing initialized. An operating system is not yet started, so the code is executed sequential. No restrictions regarding the *EEELib* handler run-time (See 5.1.4 "State machine handler" on page 54), just the *EEELib* operation overall run-time is important.
- Device operational phase
  An operating system is started. If the *EEELib* is executed in a task (e.g. timed or interrupt driven), beside the overall *EEELib* operation run-time, also the function run-time (E.g. *EEELib* handler) must be limited.

The following *EEELib* phases have to be passed sequentially and need to be implemented into the application phases:

<1> --> <2> --> <3> --> <4> --> <5>:

<1> Library is not initialized
This phase should be handled during the device initialisation phase

Status:
EEELib status is undefined. No EEPROM Emulation operation is allowed.

User application actions:
Call the function EEELib_Init (See 5.1.2 "State machine initialization" on page 50)

<2> Library is initialized, Section status is undefined
This phase should be handled during the device initialisation phase

Status:
EEELib status is defined, but the section status is undefined. All EEPROM Emulation operations except *Startup-1* or *Format* are locked and result in a Flow error, if trying to execute them (See 5.1.3 "Operation initiation" on page 50)

User application actions:
In the standard case of sections contain valid data sets, even if previous program execution was interrupted by a power fail. In this case, execute the *Startup-1* operation.
In case of blank *EEELib* sections (Completely blank Flash), execute *Format* operation first and after operation end, execute *Startup-1*

Conditions:
The *Startup-1* operation parses all data sets in the active section and if required also in the background section in order to calculate library internal pointers (See 2.4.1 "Startup-1" on page 24). Depending on how many data sets have to be parsed, the execution time of *Startup-1* is in the range of several ms. During that time, the CPU load is 100%.

<3> Library passed the *Startup-1* Phase
This phase should be handled during the device initialisation phase

Status:
The data section status is defined, but inconsistencies are not fixed. All EEPROM Emulation operations except *Startup-2*, *Read* or *Format* are locked and result in a Flow error, if trying to execute them (See 5.1.3 “Operation initiation” on page 50)

User application actions:
In case of early required EEPROM data sets, these data sets can be read using the *Read* operation. If no EEPROM Emulation data is required in this phase, it can be skipped

Conditions
In this phase the *Read* operation differs from the normal operation:
- The data set search uses a different mechanism. While in normal operation on data Flash a background hardware does the search operation, here the CPU jumps from data set to data set to find the latest data. So, the timing is different and the CPU is loaded to 100% during the *Read* operation
- In case of a preceding interrupted *Refresh* for writing emergency data followed by a RESET and *Startup-1*, the written emergency data cannot be read, as the *Read* operates on the elder section in case of detection of the interrupted *Refresh*. So, it cannot be assured, that emergency data can be read in this phase. Possibility to read these is given after *Startup-2* phase

<4> Early required data sets are read
This phase should be handled during the device initialisation phase and/or the device operational phase (See below)

Status:
as <3>

User application actions:
The *Startup-2* operation is executed

Conditions:
The *Startup-2* operation requires a longer execution time. As the long lasting operations are executed by the Flash hardware in background, the CPU load is low and *EEELib* functions return soon.
This phase may be started and handled (Call the Handler function, see 5.1.4 “State machine handler” on page 54) during the device initialisation phase if enough time is given
Another option is to start the *Startup-2* operation in the device initialisation phase (See 5.1.3 “Operation initiation” on page 50; one long lasting operation is executed in background) and to call the handler in the device operational phase. So, the *EEELib* is operational relatively early and the device startup time is not extended more than necessary

<5> Start of normal operation
This phase should be handled during the device operational phase (See below)

Status:
The *EEELib* is up and running, all operations are possible.
However, due to eventually necessary operations during *Startup-2*, the active section might be full and/or the background section might be not prepared.
In order to avoid *Write* operation errors, it should be considered that a *Prepare* and a *Refresh* operation are executed first

Conditions:
Normal operation. After starting a *EEELib* operation, the *EEELib* Handler must be called fre-

quently in order to finish the operation timely.

(2) Normal operation

Normal operation consists of data set *Read* and *Write* operations.

Furthermore, when the active section is (nearly) full, a *Refresh* must be executed to copy the latest data sets into a Prepared (Formatted) section. *Prepare* and *Refresh* are relatively long lasting operations.

Having a *Write* failure due to the full active section, followed by a *Refresh* operation, that is then followed by the repeated *Write* operation is most probably not the preferred solution in the user application. To avoid this, the function EEELib_GetSectionSize (See 5.2.1 "Check free section space" on page 55) returns the remaining space in the active section. This helps the user application schedule the *Refresh*, that it is executed right on time, e.g. during a device idle time.

Also the *Prepare* operation can be timed. It must be executed between two *Refresh* operations, while it is not important if done immediately after a *Refresh*, before a *Refresh* or at any other time in between. Furthermore, the *Prepare* operation can be called continuously, as it returns without Flash operation, if the background section is already prepared.

2.5.3 Aborting library operations

The *EEELib* allows to abort longer lasting library operations, like *Refresh*, *Prepare*, *Read* and *Write* in order to do emergency data *Write*, that cannot wait for the end of the operation (e.g. due to power supply problems, writing crash data,...). When an operation shall be aborted, the *EEELib* just sets an indication flag internally and the library internal software state machine determines where and how to abort the operation.
E.g. on *Prepare* the internal Flash erase is stopped immediately, while in case of data set *Write* a started Flash write operation (on 1/2/4 words) is finished and additionally also invalidation top and bottom IDs are written (See 2.1.2 "Data set Write operation abort" on page 12). So, some time passes by from calling the abort function till the operation is really aborted.
A latency time in the same range as writing a 4-Word data set need to be considered (Finish writing 4 Word data + Top IDL + Bottom IDL)

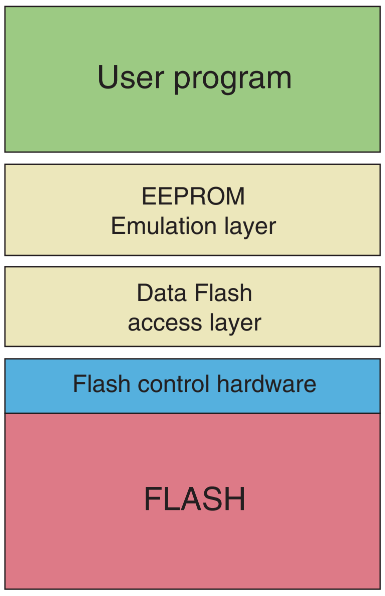
Aborting Flash operations may result in inconsistent segment structures (e.g. aborted *Refresh* leads to 2 active sections, where the newer section does not contain all data but the emergency data set). This need to be fixed before any other operation except *Write* is possible again. This is done in the *Startup-1*/2 procedures.
Following that, the *EEELib* need to be reinitialized after writing emergency data. Resuming the preceding operation is not considered.

[MEMO]

Chapter 3 Emulation Library - Structure

3.1 Layer Model

Figure 3-1: EEPROM Emulation Layer Model



The EEPROM emulation is implemented in a strictly layered manner. So that it is in principle possible, to setup a user dedicated EEPROM emulation strategy, differing from the NEC EEPROM emulation layer, but basing on the Data Flash access layer.

3.1.1 Flash control hardware

The Flash hardware can do simple Flash operations, like Flash block erase, word write, data search (Data Flash only) etc. in background after initial configuration. The upper layers and the user program can then continue operation, while the Flash hardware works. Operation end can either be polled or signalled by an interrupt.
The operation and the configuration of the hardware is not open to the user, so that the usage of the data Flash access layer is mandatory.

3.1.2 Data Flash access layer

The data Flash access layer (*DFALib*) configures the Flash hardware to do the basic Flash operations, This layer is transparent for the user, if the NEC EEPROM emulation layer is used.
The *DFALib* is explained in a separate document.

3.1.3 EEPROM emulation layer

The EEPROM emulation layer is represented by the *EEELib*. This library calls the *DFALib* for all Flash related operations.
The library contains a software state machine, controlling all EEPROM emulation related operations. State machine operations are initiated by a central function. An input structure defines the kind of operation and the parameters that are required.
Depending on the library configuration, the function returns after operation completion or after operation initiation. In the later case the user program need to frequently call a handler routine which then handles the state machine.

Chapter 4 Emulation Library - Usage

4.1 User Function Execution During EEPROM Emulation

Due to the dual operation feature, the code Flash is available all the time during the EEPROM emulation flow. So any user function from the below list can be executed from code Flash.
The time consuming task of searching the latest data set for a certain ID is realised as a background operation, done by hardware. During the search, user code can be executed.

4.1.1 EEELib handler function execution options

Basically all EEPROM emulation operations consist of a set Flash operations that are executed in a sequence, defined by the EEELib state machine. The Flash operations execution time is long compared to the software execution time to control the operations. The EEELib (basing on a DFALib define) provides the following basic options to call the handler function *EEELib_Handler* that controls the state machine (See See 5.1.4 "State machine handler" on page 54 and 4.2.3 "Configuration" on page 40):

- The *EEELib_handler* call is done by the user application.
  The *EEELib* operation is initiated by the function *EEELib_Execute*. Then, the handler function is called frequently by the user application until the operation is finished. The *EEELib* operation initialization as well as the handler calls require only short time. The user function can execute any code in between the handler function calls.

  This option will be the preferred option in case of EEPROM emulation on Data Flash.
  The operation initiation may be done from within the application functions and tasks or in a central non volatile memory manager.
  The handler may be called in different locations. See 4.5.4 "EEELib operation performance" on page 45

- The *EEELib* handler is called inside the *EEELib*.
  The user application does not have to take care of the handler function. In case of this configuration the user code gets back program flow control, after the EEPROM emulation operation is finished. This might require up to some hundreds of milliseconds. In the meantime, a user function can be called by callback from the *DFALib* internal Flash operation status check loop. See 4.2.3 "Configuration" on page 40.

4.1.2 Interrupt function execution

Any interrupt function can be executed all the time. As the code Flash is continuously accessible, also the interrupt vector table is available.

4.2 Library Software

4.2.1 Supported compilers

In general GHS, IAR and NEC Compilers are supported.

Even when tested on certain compiler versions, later versions of these compilers should be able to directly compile the projects or to convert them to a newer project type. However, this is not ensured by NEC.

4.2.2 Library delivery packages

The *DFALib* and *EEELib* are strictly separated, so that the *DFAlib* can be used without the *EEELib*. However, using *EEELib* without *DFALib* is not possible.

The delivery package contains dedicated directories for both libraries and in addition 2 header files for each one in the root directory. Furthermore, a general header file is required for the NEC types convention.

EEELib delivery package

[root]		
EEELib.h		Header file containing function prototypes, calling structures and error definitions
EEELibSetup.h		Header file with definitions for library setup at compile time
[*EEELib*]		
	EEELibGlobal.h	Library internal defines, function prototypes and variables
	EEELibUser.c	Source code for the *EEELib* internal state machine, service routines and initialization
	EEELibBasicFct.c	Source code of functions called by the state machine
	(EEELib.gpj)	Only for GHS compiler! Sub-Project file. This file can be added to the user application as a sub-project instead of adding the all source files.

DFALib delivery package

[root]		
nec_types.h		NEC types convention (Also required for the EEELib)
DFALib.h		Header file containing function prototypes and error definitions
DFALibSetup.h		Header file with definitions for library setup at compile time
[*DFALib*]		
	DFALibGlobal.h	Library internal defines, function prototypes and variables
	DFALibEnvironment.h	Library internal Flash environment related defines
	DFALibCommands.c	Source code of functions for the user interface of the *DFALib*
	DFALibHWAccess.c	Source code for the HW interface of the *DFALib*
	(DFALib.gpj)	Only for GHS compiler! Sub-Project file. This file can be added to the user application as a sub-project instead of adding all source files.

In addition to the library deliveries also an application sample is available to show the library implementation and usage in the target application.
The application sample initializes the *EEELib* and does some dummy data set *Write* and *Read* operations.

Furthermore, the *Refresh* and *Prepare* operations are executed, if necessary.

Application sample delivery package

[root]

main.c	Main source code
EEEAppControl	Source code of the control program for EEEPROM emulation
EEEApp.h	Application sample header with function prototypes and collecting all includes
nec_types.h	NEC's type definitions
target.h	target device and application related definitions
(df3xxx.h)	Only for GHS compiler: Device dependent header file
(io_macros.h)	Only for GHS compiler: Macro definitions for the device header file
(DF3xxx_irq.h)	Only for GHS compiler: IRQ definitions for the device header file
(EEEApp.gpj)	Only for GHS compiler: Project file
(DF3xxx.ld)	Only for GHS compiler: Linker directive file
(3xxx.mbs)	Only for GHS compiler: Debugger directive file
(startup_DF3xxx.s)	Only for GHS compiler: Startup assembler file
io_70f3xxx.h	Only for IAR compiler: Device dependent header file
(EEEApp.ewx)	Only for IAR compiler: Project files
(lnk70f3xxx.xcl)	Only for IAR compiler: Linker directive file
(DF3xxx_HWInit.s85)	Only for IAR compiler: Startup assembler file
(EEEApp.prw)	Only for NEC compiler: Project files
(EEEApp.prj)	Only for NEC compiler: Project files
(EEEApp.pri)	Only for NEC compiler: Project files
(EEEApp.dir)	Only for NEC compiler: Linker directive file
(crte.s)	Only for NEC compiler: Startup assembler file
<*DFALib* delivery package>	
<*EEELib* delivery package>	

4.2.3 Configuration

The libraries are configured in the DFALibSetup.h and EEELibSetup.h files.
The following configuration options are to be considered:

Flash type

The EEPROM emulation can operate on Data Flash and on Code Flash. Anyhow, as Emulation on Code Flash is not yet allowed, please use only the definition for Data Flash

File:	DFALibSetup.h	
Define:	DFALIB_FLASH_TYPE	
Definition options:	DFALIB_DATA_FLASH	Flash type is Data Flash
	DFALIB_CODE_FLASH	Flash type is Code Flash

***DFALib* Status check / *EEELib* state machine handler**

Please also refer to 4.1.1 "EEELib handler function execution options" on page 37.
The Flash background operations (e.g. Flash erase) need some time. Before continuing with any Flash related operation, like stepping forward the *EEELib* internal state machine, the SW need to check the end of the Flash operation. The *EEELib* handler responsible for these checks can either be called by the user application until the EEPROM emulation operation is finished or it can be called inside the *EEELib*.

File:	DFALibSetup.h	
Define:	DFALIB_STATUS_CHECK	
Definition options:	DFALIB_STATUS_CHECK_USER	The handler must be called by the user application
	DFALIB_STATUS_CHECK_INTERNAL	The *EEELib* calls the user handler internally

Flash Block Size

Here the Flash block size of the used device shall be set. Current implementations can be 2 kB (0x800) for data Flash and 2 kB, 4 kB or 8 kB for code Flash, depending on the device. Please refer to the device user's manual.

File:	DFALibSetup.h
Define:	DFALIB_FLASH_BLOCK_SIZE
Definition options:	0x800 (2 kB blocks), 0x1000 (4 kB blocks), 0x2000 (8 kB blocks)

User call-back functions

In case of DFALib internal status check, the library does a status check loop to wait for the end of a Flash operation. In that loop a user callback function can be called.
This function is additionally called in the EEELib loop where the handler function is called until the EEELib operation is finished. By this double call minimum latency of the callback function call is reached.

File: DFALibSetup.h
Define: DFALIB_USER_CALLBACK_FUNCTION
E.g.

```
extern void userFunc(void);
#define DFALIB_USER_CALLBACK_FUNCTION userFunc()
```

Note: Most probably the callback functionality would be used in EEPROM emulation on Code Flash. As on Code Flash the Flash is not available during Flash operations, the user function would need to be located outside the Flash, e.g. in RAM.

Special Flash dependent defines

Please refer to the application sample to check, which defines to set.

Flash related information for the EEPROM emulation

The Flash base address depends on the used Flash type.
On V850 devices the code Flash always starts on 0x00000000. Data Flash start address depends on the device. Please refer to the device documentation.
E.g. on V850ES/FJ3 the Flash start address can be configured according to the chip selects. Default chip select is CS3 and the resulting Flash base address is 0x00ff8000.

File:	EEELibSetup.h
Define:	EEELIB_FLASH_BASEADDRESS
Definition options:	See above. e.g. 0xff8000

EEPROM emulation sections related information

Note: The number of Flash blocks (n) used for one EEPROM emulation data section must follow the rule:
NoBlocks = 2 power m with m=0, 1, 2,... So NoBlocks can be 1, 2, 4,...
The EEELIB_SECTION_SIZE must then be set to
NoBlocks * Flash block size
In case of Flash block size = 0x800, the section size may be set to 0x800 (2 kB sections), 0x1000 (4 kB sections), 0x2000 (8 kB sections), 0x4000 (16 kB sections) and more on bigger Flash

File:	EEELibSetup.h
Define:	EEELIB_SECTION_SIZE
Definition options:	See above. e.g. 0x4000

Note: The emulation start address EEELIB_START_ADDRESS must be set to
EEELIB_FLASH_BASEADDRESS + x * EEELIB_SECTION_SIZE
with x=0, 1, 2,...

File:	EEELibSetup.h
Define:	EEELIB_START_ADDRESS
Definition options:	See above. e.g. 0xff8000

Note: The EEPROM emulation needs 2 EEPROM emulation sections.
So, EEELIB_START_ADDRESS + 2 * EEELIB_SECTION_SIZE
must be set to not exceed the Flash boundaries.

Handler execution in Flash Interrupt context

The library provides support for the Flash interrupt. Whenever a Flash operation (means a state machine state of the EEELib) is finished, the Flash interrupt is requested. Executing the handler function in this interrupt (And only there!), the handler would be called exactly and only when required, resulting in optimum performance and low CPU load. However, the execution time of <= 100us must be considered for the interrupt function

File:	EEELibSetup.h
Define:	EEELIB_INTREQ_SUPPORT
Definition options:	define or don’t define (e.g. comment out the define)

Note: When the define is set, also the FLash interrupt must be unmasked and the handler function must be called in the interrupt context. This setup need to be done within the user application

4.3 Section Handling

The following sections are EEPROM emulation library related:

- DFALib_Text
  This is *DFALib* code.
  In case of EEPROM emulation in data Flash, this section can be executed from the Code Flash. This is also possible in case of EEPROM emulation on Code Flash and *EEELib* internal handler execution (See 4.1.1 “EEELib handler function execution options” on page 37).

- DFALib_TextRam
  This is also *DFALib* code.
  In case of EEPROM emulation in Data Flash, this section can be executed from the code Flash.
  In case of EEPROM emulation on Code Flash, this section would need to be executed from internal RAM, as during execution of this code the Flash environment is activated and so the Flash is not accessible.

- EEELib_Text
  This is *EEELib* code.
  In case of EEPROM emulation in data Flash, this section can be executed from the code Flash. This is also possible in case of EEPROM emulation on Code Flash and *EEELib* internal handler execution.

- EEELib_Data
  This is the section containing *EEELib* internal data. It must be located to (internal) RAM.

4.4 MISRA Compliance

The EEELib and DFALib have been tested regarding MISRA compliance.

The used tool is the GHS V4.0.7 MULTI development environment, which tests on the predecessor of the MISRA 2004 standard.

The following rules had to be disabled, as enabling would result in significant disadvantages in terms of code size, performance or code readability:

- Rule 23: Module local functions shall be defined as static
  The checker did not recognize static functions as local
- Rule 45: Casting from and to pointers not allowed
  This is intensively used for all sfr accesses due to performance and code size reasons
- Rule 90: c-macros shall be used for constants, function-like or specifier/qualifier only
  This rule prohibits defines, assigning former defines. This is used in the libraries for better code readability and reduction of possible user misconfiguration
- Rule 96: Use of parenthesis in function like macros
  This rule prohibits macros with parameters defined as empty macros. This is used in the libraries for all debug output definitions.

4.5 Miscellaneous Forethoughts

The following sub-chapters describe important issues that must be considered when implementing the EEPROM emulation into the user application

4.5.1 Power safe modes

The usage of any device power safe modes including the HALT instruction is prohibited during a EEELib operation. Entering a power safe mode stops any Flash operation without notification of the libraries and so without notification of the user application. This may result in inconsistent data set contents or inconsistent section structure resulting in data loss or wrong read values

4.5.2 DMA transfers

DMA transfers during EEELib operations are permitted, but may have an impact on the EEELib performance and function execution time.
Background is that DFALib functions, called by the EEELib, need to modify protected registers. If the required protection sequence is interrupted by a DMA transfer, it is repeated in a loop until the register modification is successful. A long lasting DMA transfer may so result in long function execution latency for completion of this loops.

4.5.3 Data set length change on later application update

If later on an application shall be updated and with this update a change of a data set length is intended, it is not possible to directly write the data set new with the new length. Background is library robustness and power fail recovery considerations.

The data set ID must first be removed from the emulation by the operation *Refresh-EDS* (See also 2.3.5 “Refresh-EDS” on page 21 and 5.1.3 “Operation initiation” on page 50...).
Afterwards, the data set can be written new with the changed length.

The *Refresh-EDS* operation is a variant of the *Refresh* operation. It copies the latest data set instance of each ID to a prepared section except the IDs to exclude. As this is time consuming, the *Refresh-EDS* allows to remove a set of IDs at once according to a list passed to the operation as parameter.
The list has to be set up by application. The service function *EEELib_GetDSLength* can help to set up the list by checking the availability IDs and length of data sets in the current Emulation.

4.5.4 EEELib operation performance

When the EEELib handler is called by the user application (standard configuration for devices with Data Flash), the operation performance strongly depends on the frequency of the handler calls. This especially affects operations which require many Flash write operations until the operation is finished, such as *Refresh, Write*, *Startup-2* (when after a power fail the section consistancy need to be repaired).
As the typically Flash write operation needs between 200 and 500us, a slower handler call frequency significanly reduces the operation performance.
Currently the following user application implementations are considered (also mixtures are possible if the synchronization of the function calls is ensured):

- Call in a timed task
  In order to achive a reasonable emulation operation performance, the time slice should not be selected too big. A 500us interval would not significantly reduce the EEELib performance and so seems to be a reasonable compromise between library performance and CPU load
- Call in the idle task
  If it is ensured that the idle task is called often enough, that method might result in the good performance as the handler can be called continuously. However, as this method is not deterministic in case of higher CPU load by the application, it might be combined with calls in a timed task
- Call in the Flash interrupt context (See 4.2.3 “Configuration” on page 40)
  The library provides support for the Flash interrupt. The library would be called exactly and only when required, resulting in optimum performance and low CPU load. However, the execution time of <= 100us must be considered for the interrupt function

The best solution depends on the user application and need to be evaluated regarding performance (frequent handler calls), CPU load, interrupt latency etc.

4.5.5 Function reentrancy

All functions are not reentrant. So, reentrant calls of any EEELib or DFALib functions must be avoided

[MEMO]

Chapter 5 Emulation Library - API

The EEPROM Emulation library provides two types of user functions:

- Operational functions
  Operational functions control the standard EEPROM emulation operations like *Read*, *Write*, *Prepare*, etc.

- Service functions
  Providing service information to the user.

5.1 Operational Functions

EEPROM operation is controlled by an *EEELib* internal state machine. The operations are initiated by the function EEELib_Execute.
The function EEELib_Handler controls the state machine after operation initiation and forwards the states. Depending on the definition for the handler execution (See 4.1.1 "EEELib handler function execution options" on page 37), the handler function is either called *EEELib* internally or by the user application.

5.1.1 Command structure

The following structure is passed to the function EEELib_Execute to control the state machine and to read the status and operation results.

```
typedef struct
        {
        u32*            address;
        u16             identifier;
        u16             length;
        u16             offset;
        u16*            addList;
        tEEE_COMMAND    command;
        tEEE_ERROR      error;
        } tEEE_REQUEST;
```

command	Distinguishes between the different operations *Startup-1, Startup-2*, *Read*, *Write*, *Refresh*, *Format* and *Prepare.* See 5.1.3 "Operation initiation" on page 50
address identifier length offset addList	These parameters are individually used by the different operations.
error	various errors are possible, depending on the operation. Depending on the configured status check type (Internal or by the user, see 4.2.3 "Configuration" on page 40), the errors may be set by the initiating function (EEELib_Execute) or by the handler function (EEELib_Handler). See error describtion below.

Errors

EEE_OK	The operation finished successfully
EEE_DRIVER_BUSY	The state machine is busy This value is set, when a new operation has been successfully started by EEELib_Execute (Only applicable in case of user status check)
EEE_ERR_PARAMETER	Wrong parameters have been passed to the *EEELib* (See the different commands) --> Please check within the application why the error return occurred
EEE_ERR_SECTION_FULL	A data set *Write* attempt failed, as the section is full. Then, a *Refresh* operation is required
EEE_ERR_POOL_FULL	A *Refresh* operation failed, as the background section is not prepared
EEE_ERR_COMMAND_UNKNOWN	An unknown command has been passed to the *EEELib* --> Please check within the application why the error return occurred
EEE_ERR_FLASH_ERROR	A Flash operation of the *DFALib* (called by *EEELib*) failed due to a Flash problem. --> The Flash should be considered as defect

EEE_ERR_USERABORT	The recent operation has been aborted (Only applicable in case of user status check)
EEE_ERR_READ_UNKNOWNID	A *Read* operation did not find a data set with the requested ID
EEE_ERR_READ_OBSOLETE	The data set with the ID is marked invalid. The data is not read, instead the error is returned
EEE_ERR_NOACTIVESECTION	On Startup-1: This error is returned in case of unformatted Flash On Prepare or Startup-2: The Prepare (Or Prepare issued by Startup-2) could not be executed, as the erase counter of the active section got an overflow or was inconsistent This error should not occure, as the erase counter (max 0xFFF1) exceeds the Flash specification (max 10000 erase cycles) by far Anyhow, a Format operation can reset the erase counter --> Please check within the application why the error return occurred
EEE_ERR_FLOW	An operation, that is not allowed according to the *EEELib* initialisation status was called --> Please check within the application why the error return occurred
EEE_ERR_OPERATION_REJECTED	A new operation should be initiated although the state machine is still busy (Only applicable in case of user status check) --> Please check within the application why the error return occurred
EEE_ERR_SECTION_OVERFLOW	A Startup-2 operation failed in the Refresh operation, called internally (See 2.3.4 "Refresh" on page 20) The EEPROM emulation can continue normally, but the one (elder) active section is full and the other section with the incomplete Refresh is marked consumed. --> Please check within the application why the error return occurred **Note:** If after *Refresh* abort new emergency data was written to the new active section, this data is lost as the elder section without this data remains active
EEE_ERR_SECTION_FIXED	During Refresh: An inconsistancy in one of the data sets resulted in loss of one data set (See 2.3.4 "Refresh" on page 20) --> Please check within the application why the error return occurred During Startup-2: Startup-2 has done a consistancy fix-operation. This is a normal return value on Startup-2 after power fail. The sections are assumed to be consistent and emulation can normally continue. On Startup-2, the error can be considered as a notification only

5.1.2 State machine initialization

Function prototype
void EEELib_Init(void);

Required parameters
-

Return value
-

Function Description
This function initializes the *EEELib* state machine. The function must be called once before the 1st EEPROM emulation operation.

5.1.3 Operation initiation

Function prototype
void EEELib_Execute(tEEE_REQUEST* request);

Required parameters

request	Command structure as described below

Return value

request.error	Values as described in 5.1.1 “Command structure” on page 48.

Operation Description
Initiates a EEPROM emulation operation.

- Startup-1
  The *Startup-1* operation is explained in 2.4.1 “Startup-1” on page 24

  required parameters:

request.command	EEE_CMD_STARTUP1

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK
	EEE_DRIVER_BUSY
	EEE_ERR_NOACTIVESECTION
	EEE_ERR_OPERATION_REJECTED
	EEE_ERR_FLASH_ERROR

- Startup-2
  The *Startup-2* operation is explained in 2.4.2 “Startup-2” on page 26

  required parameters:

request.command	EEE_CMD_STARTUP2

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK
	EEE_DRIVER_BUSY
	EEE_ERR_FLASH_ERROR
	EEE_ERR_FLOW
	EEE_ERR_OPERATION_REJECTED
	EEE_ERR_SECTION_OVERFLOW
	EEE_ERR_SECTION_FIXED
	EEE_ERR_NOACTIVESECTION

- *Prepare*
  The *Prepare* operation is explained in 2.3.1 “Prepare” on page 17

  required parameters:

request.command	EEE_CMD_PREPARE

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK
	EEE_DRIVER_BUSY
	EEE_ERR_FLASH_ERROR
	EEE_ERR_USERABORT
	EEE_ERR_FLOW
	EEE_ERR_OPERATION_REJECTED
	EEE_ERR_NOACTIVESECTION

- *Refresh*
  The *Refresh* operation is explained in 2.3.4 “Refresh” on page 20

  required parameters:

request.command	EEE_CMD_REFRESH

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK
	EEE_DRIVER_BUSY
	EEE_ERR_POOL_FULL
	EEE_ERR_FLASH_ERROR
	EEE_ERR_USERABORT
	EEE_ERR_FLOW
	EEE_ERR_OPERATION_REJECTED
	EEE_ERR_SECTION_OVERFLOW
	EEE_ERR_SECTION_FIXED

- *Refresh-EDS*
  The *Refresh-EDS* operation is explained in 2.3.5 “Refresh-EDS” on page 21

  required parameters:

request.command	EEE_CMD_REFRESH_EDS

request.addList	pointer to a zero terminated list of IDs, that shall be excluded from the REFRESH process and so will be removed from the emulation.
	E.g.: u16 removeList[]={ 5, 19, 86, 97, 99, 0 };
	request.addList = &(removeList[0]);

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK EEE_DRIVER_BUSY EEE_ERR_POOL_FULL EEE_ERR_FLASH_ERROR EEE_ERR_USERABORT EEE_ERR_FLOW EEE_ERR_OPERATION_REJECTED EEE_ERR_SECTION_OVERFLOW EEE_ERR_SECTION_FIXED

- *Write*
  The *Write* operation is explained in 2.3.2 “Write data set” on page 18

  required parameters:

request.command	EEE_CMD_WRITE
request.address	Source address of the *Write* operation
request.identifier	Identifier of the data set to be written
request.length	Length of the data set to be written. The write granularity is 4 Bytes for Data Flash and 8 bytes for code Flash Length=0 defines an invalidation data set. (2.1.1 “Invalidation of data (Obsolete data)” on page 12)

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK EEE_ERR_PARAMETER Result from: - The write granularity of 4 Bytes on Data Flash respectively 8 Bytes for Code Flash is not met - ID is 0x0000 or > 0xFFFE) - If already a data set instance is in the emulation, length is != length of 1st instance and is != 0 - If no data set instance is in the emulation, length information is 0 or > section size EEE_DRIVER_BUSY EEE_ERR_SECTION_FULL EEE_ERR_FLASH_ERROR EEE_ERR_USERABORT EEE_ERR_FLOW EEE_ERR_OPERATION_REJECTED

  Note:
  If no instance of a data set with the ID is available in the active section, the length information must be != 0. Writing the first data set instance of an ID as obsolete data (length = 0) is prohibited

- *Read*
  The *Read* operation is explained in 2.3.3 “Read” on page 19

  required parameters:

request.command	EEE_CMD_READ
request.address	Destination address of the *Read* operation. The data set (part) is copied here
request.identifier	Identifier of the data set to be read
request.length	Number of bytes to read
request.offset	Offset within the data set to read. Together with length information this is used to read data set fragments

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK
	EEE_DRIVER_BUSY
	EEE_ERR_PARAMETER Resulting from: data set length < (request.length+ request.offset)
	EEE_ERR_READ_UNKNOWNID
	EEE_ERR_READ_OBSOLETE
	EEE_ERR_USERABORT
	EEE_ERR_FLOW
	EEE_ERR_OPERATION_REJECTED

- *Format*
  The *Format* operation is explained in 2.3.6 “Format” on page 23
  required parameters:

request.command	EEE_CMD_FORMAT

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_OK
	EEE_DRIVER_BUSY
	EEE_ERR_FLASH_ERROR
	EEE_ERR_OPERATION_REJECTED

  Notes:
  - After execution of the *Format* operation, the EEELib must be "restarted" with the *Startup-1*
  - The operation does not reset the erase counter, if it is valid. Only in case of invalid erase counter or erase counter overflow (>0xFFF1) it is reset

- Other commands
  Other commands are not available, an error is returned

request.command	Any other unknown command number

  return (See 5.1.1 “Command structure” on page 48):

request.error	EEE_ERR_COMMAND_UNKNOWN

5.1.4 State machine handler

Function prototype
void EEELib_Handler(void);

Required parameters
The request structure, passed to the function EEELib_Execute, is used.

Return value
The request structure, passed to the function EEELib_Execute, is modified.

request.error	Values as described in 5.1.1 "Command structure" on page 48

Function Description
Handles the *EEELib* state machine and forwards the states.

Notes: 1. Depending on the definition for the handler execution (See 4.1.1 "EEELib handler function execution options" on page 37), this handler function is either called *EEELib* internally or by the user application. In case of library internal calls, this function is not required and may not be called in the user application.

2. Do not change the command structure information until the *EEELib* operation is finished (request.error != EEE_DRIVER_BUSY)

5.1.5 State machine abort

Function prototype
void EEELib_Abort(void);

Required parameters
-

Return value
-

Function Description
This function indicates the user abort request to the state machine. The state machine aborts an EEP-ROM emulation operation as soon as possible.

Notes: 1. After the abort the handler function must still be called until the state machine has completed the abort procedure. This is indicated by the request.state and request.error accordingly.

2. Aborts before finish of the *Startup-2* operation are not possible, so ignored.

5.2 Service Functions

5.2.1 Check free section space

Function prototype
tEEE_ERROR EEELib_GetSpace(u32 *space);

Required parameters

space	pointer to the variable, that shall receive the section space information

Return value

EEE_OK	The operation finished successfully
EEE_DRIVER_BUSY	The *EEELib* state machine is busy, the size information could not be calculated
EEE_ERR_FLOW	The function can not be executed in the library initialization phase. Execute the function after Startup-2

Function Description
This function calculates the free space in the active section, usable for new data sets.

Note: Please consider, that the number of words written to the active section on a *Write* operation is 2 words (8 Bytes) more than the length, passed to the function EEELib_Execute. These additional words are required for the top and bottom ID-L.
When comparing the data set length with the return value "space", these 2 words must be considered
A data set fits into the section under the condition:
space >= DS length (Bytes) + 8

5.2.2 Get the erase counter

Function prototype
tEEE_ERROR EEELib_GetEraseCounter(u32 *eCnt);

Required parameters

eCnt	Pointer to the variable, that shall receive the erase counter information

Return value

EEE_OK	The operation finished successfully
EEE_DRIVER_BUSY	The *EEELib* state machine is busy, the size information could not be calculated
EEE_ERR_FLOW	The function can not be executed in the library initialization phase. Execute the function after Startup-2

Function Description
This function reads the erase counter of the EEPROM emulation sections.

5.2.3 Get the Data Set Length

Function prototype
tEEE_ERROR EEELib_GetDSLength(u32 id, u16 *pIDlen);

Required parameters

id	ID that shall be checked
pIDlen	pointer to the destination where the length of the checked ID is written

Return value

EEE_OK	The ID is available in the emulation.
EEE_ERR_READ_UNKNOWNID	The ID is not available in the emulation. The returned length is 0
EEE_ERR_FLOW	The function can not be executed in the library initialization phase. Execute the function after Startup-2
EEE_DRIVER_BUSY	The *EEELib* state machine is busy, the function could not check the ID

Function Description
This function checks if an ID is available in the EEPROM emulation. If yes, it furthermore returns the length of the data sets with this ID. The length information ist written to the adresss stored in pIDLen

5.2.4 Get the EEELib version

Function prototype
u32 EEELib_LibVersion(void);

Required parameters
-

Return value
EEELib version as decimal number.

Function Description
This function returns the *EEELib* version as decimal number.