Application Note

78K0R

16-Bit Single-Chip Microcontrollers

EEPROM Emulation T10

78K0R/Kx3-L
78K0R/Ix3
78K0R/Lx3
78K0R/Kx3-C
78K0R/Kx3-A

Document No. U19682EE2V0AN00

Date Published: August 2009

Legal Notes

- The information contained in this document is being issued in advance of the production cycle for the product. The parameters for the product may change before final production or NEC Electronics Corporation, at its own discretion, may withdraw the product prior to its production.
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC 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 NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special", and "Specific". The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics products before using it in a particular application.
  "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.
  "Special": 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, life support systems and medical equipment for life support, etc.
  The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data

Regional Information

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

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

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

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668, Japan
Tel: 044 4355111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554,|
U.S.A.
Tel: 408 5886000
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
0472 Düsseldorf, Germany
Tel: 0211 65030
http://www.eu.necel.com/

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908 691133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01 30675800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091 5042787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 6387200

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02 667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven,
The Netherlands
Tel: 040 2654010

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27
ZhiChunLu Haidian District,
Beijing 100083, P.R.China
Tel: 010 82351155
http://www.cn.necel.com/

NEC Electronics Shanghai Ltd.
Room 2511-2512, Bank of China
Tower,
200 Yincheng Road Central,
Pudong New Area,
Shanghai 200120, P.R. China
Tel: 021 58885400
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
12/F., Cityplaza 4,
12 Taikoo Wan Road, Hong Kong
Tel: 2886 9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R.O.C.
Tel: 02 27192377

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253 8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku, Seoul,
135-080, Korea Tel: 02-558-3737
http://www.kr.necel.com/

Table of Contents

Chapter 1. Overview 7

1.1 Naming conventions 8
1.2 General Approach 9

Chapter 2. Architecture 10

2.1 System architecture 11
2.1.1 Import-, exports- lists 12
2.1.2 Module relationship 13
2.2 Driver Arcitecture 14
2.2.1 Request response model 14
2.2.2 Physical placement of the components 14
2.2.3 EEL anchor 15
2.2.4 EEL Pool 16
2.2.5 EEL block 17
2.2.6 EEL reference table 18
2.2.7 EEL instance lookup table 18
2.2.8 EEL data flash area 19

Chapter 3. User Interface (API) 20

3.1 Constant definitions 20
3.2 Data type definitions 20
3.3 Block status type 21
3.4 Command code type 22
3.5 Status type 23
3.6 Error type 24
3.7 Request type 25
3.8 Function prototypes 26
3.8.1 EEL_Init() 27
3.8.2 EEL_Open() 28
3.8.3 EEL_Close() 29
3.8.4 EEL_Enforce(my_request) 30
3.8.5 EEL_Execute(my_request) 32
3.8.6 EEL_Handler() 35
3.8.7 EEL_CheckDriverStatus() 37
3.8.8 EEL_GetPool() 39
3.8.9 EEL_GetSpace() 40
3.8.10 EEL_GetBlockStatus(my_block_u08) 41
3.8.11 EEL_GetActiveBlock() 42
3.8.12 EEL_GetNextBlock() 43
3.8.13 EEL_GetPrevBlock() 44

Chapter 4. Commands 45

4.1 startup 46
4.2 write 50
4.3 read 53
4.4 refresh 56
4.5 format 59
4.6 prepare 62
4.7 repair 65
4.8 exclude 68
4.9 shutdown 71

Chapter 5. Implementation 73

5.1 Conditions 73
5.2 Resources 73
5.2.1 Resource distribution 73
5.2.2 Resource consumption 74

5.3 Timing 75
5.3.1 Typical timings 75
5.3.2 Maximum timings 76
5.4 File structure 79
5.4.1 NEC compiler compatible version 79
5.4.2 IAR compiler compatible version 79
Chapter 6. EEL driver integration 80

6.1 Link time configuration 80
6.1.1 Flash self-programming library configuration 80
6.1.2 EEPROM library configuration 81
6.1.3 Linker configuration 82
6.2 Run-time configuration 83
6.2.1 EEL activation and deactivation sequence 83
6.2.2 Real-time capabilities 84
6.2.3 Interrupts in enforced operation mode 85
6.2.4 Interrupts in background operation mode 85
Chapter 7. Error handling 86

Appendix A Revision History 87

Chapter 1. Overview

Conventional EEPROM memory (electrically erasable programmable read only memory) is used in embedded applications for storing data that has been modify at operation time but must maintain after power supply is switched off. It's normally used as an external component controlled by the microcontroller via appropriate communication interface (CSI, I²C or others). Some microcontrollers are already equipped with EEPROM on-chip, but in any case a real EEPROM in a system is always a matter of costs.

Modern microcontroller are using flash technologies with sufficient endurance and data retention characteristics suitable for storing of dynamical data in flash memory under software control only.

Generally EEPROM emulation is a piece of software emulating the functionality of a conventional EEPROM based on usage of internal flash memory. It's managing the used resources like flash memory space, endurance, data retention and the access to the virtual EEPROM memory during its operation.

As well real EEPROM as its emulation can only write into location eased beforehand, but there is one fundamental difference between both:

- in case of conventional EEPROM the memory content can be re-programmed(erased and written) as one access in units of one ore more bytes.

- EEPROM emulation driver has to consider the device and flash technology requirements and restrictions during its operation. One of these is the difference between the smallest erasable unit (one flash block) and the smallest writeable unit (here 1 flash-word = 4 bytes). This specific characteristic requires some tricky countermeasure to keep the data always consistent. Another constraining parameter is the limited number of block erase-cycles (endurance) requires an efficient, secure and sophisticated implementation.

This User's Manual describes the usage of NEC's EEPROM emulation library featuring EEPROM emulation on NEC's 78K0R microcontroller with embedded single voltage flash. It sketches the internal driver architecture necessary for general understanding but its emphasis is the integration and operation of the EEPROM emulation driver in embedded applications.

1.1 Naming conventions

Table 1-1 naming conventions

Acronym/Abbreviation	Description
Active block	The only one block inside the EEL pool that contains all actual instances of EEL variables. The read and write access is processed via this block.
Active pool	Collection of blocks containing valid EEL data or waiting for data
Background operation mode	EEL driver commands are executed state-wise (state by state) controlled by the EEL-Handler. In this operation mode the application undertakes the CPU control in between internal EEL states. This mode is preferably in applications using preemptive or non-preemptive operating systems.
block	(flash block) smallest erase-able unit can be erased in flash by a erase pulse
bsize, bidx, baddr	variable size, byte-index, byte-address expressed in bytes
Chopper	periodical interrupt source “chopping” the execution time of the selfprogramming commands used by the EEL driver into predefined time slices.
DFA	(Data Flash Area) Area on the bottom of the active block where pure data of the instances are stored.
DS	Data Set
EC	Erase Counter Data word in the block header
EEL	EEPROM Emulation Library
EEL anchor	ROM constant structure containing EEL variable description managed by EEL
EEL instance	Data-set of a given EEL variable written to the flash. Each write access to the EEL generates a new reference inside the ILT and a new instance in the data flash area (DFA).
EEL pool	Set of subsequent flash blocks used for saving the EEPROM data.
EEL variable	Variable registered in eel_anchor[] with its attributes “identifier” and “size”.
EES	EEPROM Emulation System (combination of Flash access library, e.g. FSL and EEPROM Emulation library EEL)
Enforced operation mode	EEL driver commands are executed completely at the caller side. In this operation mode the EEL driver retains the CPU control as long executing the command.
faddr	physical flash address
FAL	Flash Access Library (accessing code flash and data flash)
FSL	Short form of Flash Self-programming Library
Firmware	Internal software, that is managing the access to the flash system.
ID	Identifier of a variable

ILT	(Instance LookupTable) Area on the top of active block containing all active instance references.
Module	One source-file consists of a definition header file and an implementation C file.
Passive pool	Collection of blocks that does not contain data (consumed, erased, prepared, invalid…)
pool	Collection of successive flash blocks
Separator	The first flash word inside the erased area between the ILT and the DFA.
word	(flash word) smallest write-able unit can be written into flash by a write pulse
wsize, widx, waddr	variable size, word-index, word-address expressed in flash words

1.2 General Approach

For the 78K0R devices the same physical flash memory is used for storing application code as well for the EEL data. The flash memory does not support so called “dual operation”, consequently each flash access being performed by the self-programming firmware inhibits the execution of application code at the same time. From CPU point of view the application code disappears practically during the flash access. This fact influences seriously the real-time behavior of the whole system when the EEL driver is active. NEC’s implementation of the EEPROM emulation tries to defuse the real-time situation and to retrieve the maximum of flexibility and transparency for applications operation. Following claims and requirements formed the guideline for the chosen implementation way:

- simple straight forward architecture based on request-response model
- write and read access of any data portions between 1 and 255 bytes
- EEL variable identification based on unique 1 byte identifier (max 126 variables)
- inconsistencies caused by asynchronous power-off RESET are always detectable and repairable
- reparation initiated by the application to avoid a non-deterministic write/read access time
- read- and write-access time is independent of the internal driver status
- no blocking time longer than a period pre-determined by the user
- user configurable interrupt scenario for EEL operation
- no additional interrupt latency caused by the EEL driver

Basically the data stored in an EEPROM could be categorized as following:

1. static data like product number, serial number, static parameter
2. rare updatable data like immobilizer key, engine characteristic
3. agile, dynamic data like ODO-meter, window lifter position

For handling of data type (1.) we recommend to use pure self-programming functions however it is also possible to integrate it into the EEPROM emulation. Data of type (2.) and (3.) are predestined to be managed by the EEPROM emulation driver. The partitioning of the data should be done with respect to constrains resulting from the limited flash block-size. This causes limited number of independent EEL variables and is limiting total size (sum of particular sizes) of all EEL variables.

Chapter 2. Architecture

The EEPROM emulation driver architecture and strategy described in this document is chosen by NEC to offer the user a maximum of flexibility and safety under acceptable losses in real-time behavior. But this is one possible implementation. Principally based on the flash self-programming library FSL the user can implement a different EEPROM emulation strategy. However some rules and recommendation has to be considered to achieve the specified flash endurance and data retention of the EEPROM data:

1. only commands offered by NEC's self-programming library can be used for implementation
2. write access into previously erased full flash words (4 bytes) is allowed only
3. after power-on RESET read- and write-access to/from the related flash block is possible after successful internal verification by the "FSL_IVerify" command.
4. do not use a flash blocks for data and code anymore when "FSL_Write" or "FSL_EEPROMWrite" was terminated with error code “0x1C(FSL_ERR_WRITE)" or “0x1D (FSL_ERR_EEPWRITE_VERIFY)“
5. do not use a flash block for data and code anymore when "FSL_Erase" command was terminated with error code “"0x1A (FSL_ERR_ERASE)"

It is recommended to keep one data block available for redundancy reasons.

Note:

Data retention time starts when writing a word into an erased block. When adding more data records into the same block, full data retention of the whole block is achieved only, if internal verify command is executed after writing the last data record.

2.1 System architecture

The EEL driver is embedded in a strictly layered system and building up on NEC's self-programming library as a bridge between the flash control subsystem (firmware and hardware) and the EEL. For details please refer to the flash selfprogramming library user's manual.

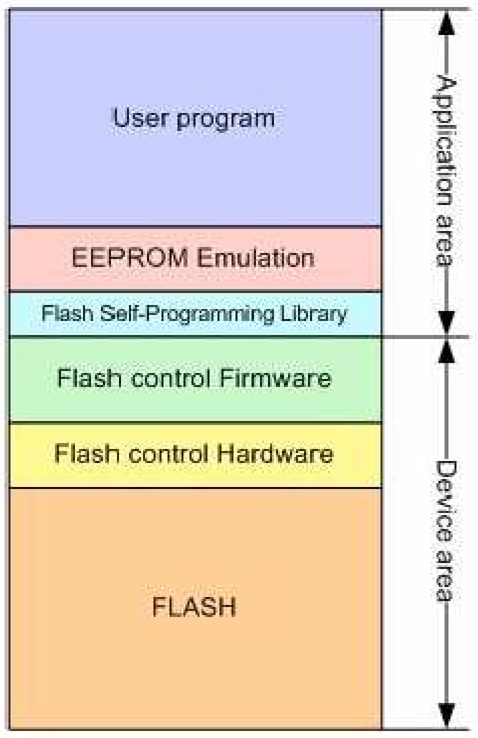

Figure 2-1 EEPROM emulation layer model

2.1.1 Import-, exports- lists

The EEL driver is using the software interface (API) offered by NEC's flash self-programming library (FSL). On its part the EEL driver offers a dedicated user interface (API) can be used by the application for operative and administrative measures like command execution and operation supervision. The precompiled EEL like the FSL, contains a user configurable parts bordered in red.

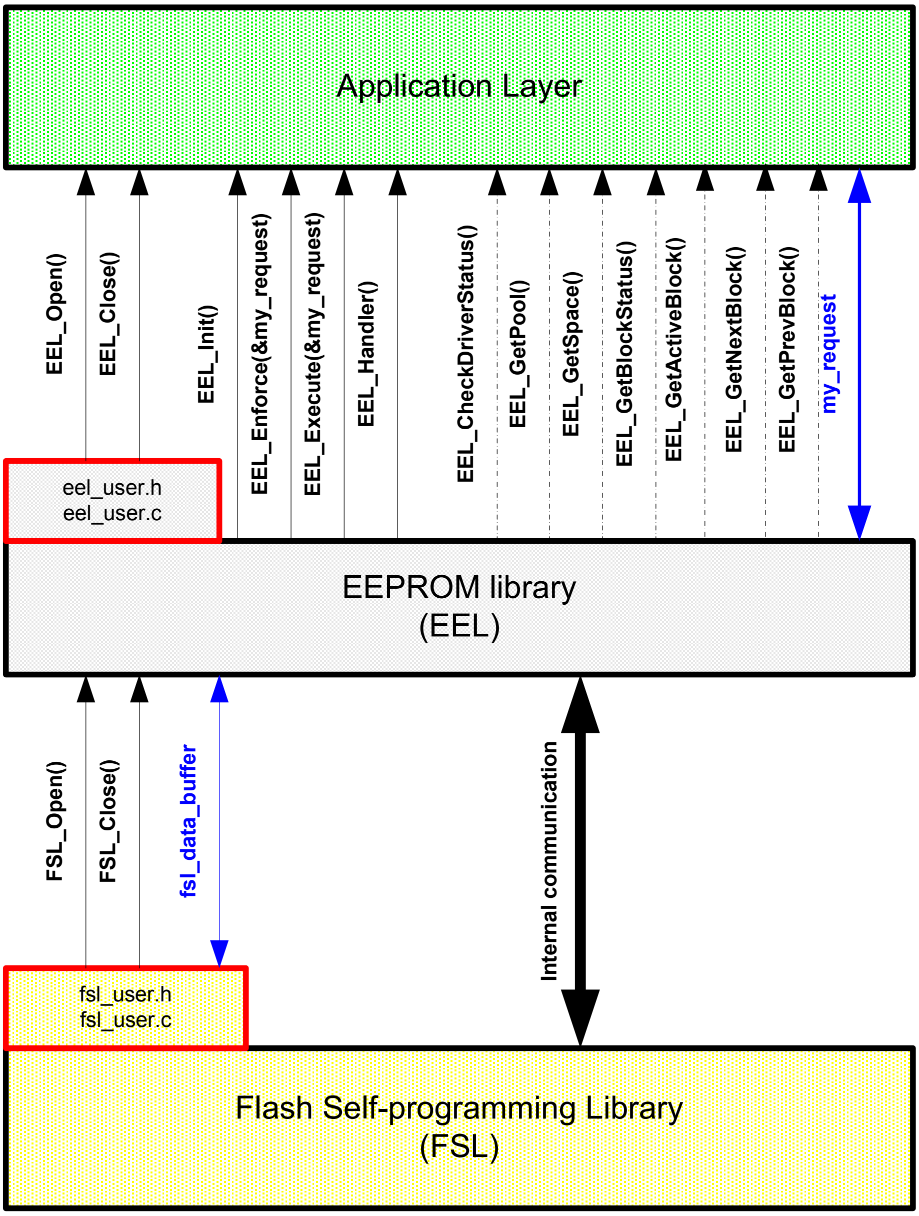

Figure 2-2 Import export diagram (RAM variables marked in blue)

2.1.2 Module relationship

The file relationship of the EEL library obeys strict hierarchical top down order

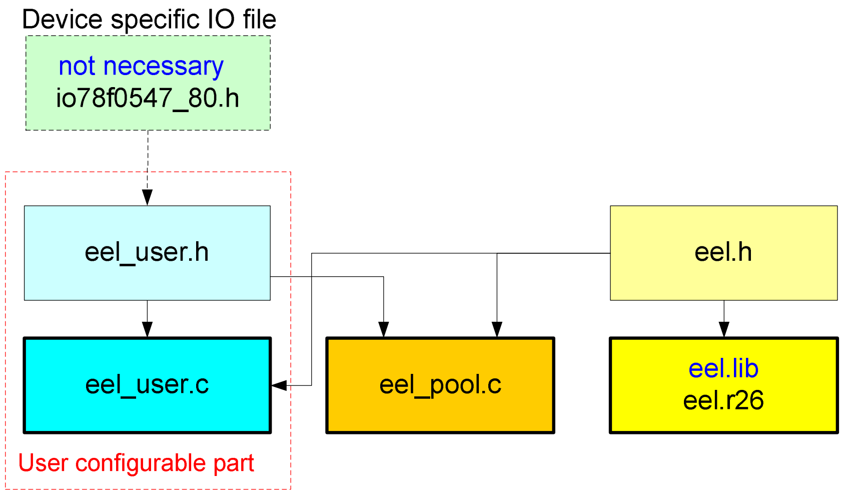

Figure 2-3 Include hierarchical diagram

Notes:

1. NEC compatible modules are marked in blue.
2. IAR version of the EEL driver has to include the device I/O-file explicit in eel_user.h

2.2 Driver Arcitecture

The strongest requirement for the design of the driver was avoiding of blocking time during driver operation time. For that reason this EEL implementation dispenses of any automatic mode, like automatic copy process (refresh), automatic erase process (prepare) and others. The application can check the internal status of the driver and can initiate appropriate command at most convenient time. This can be used to prepare erased blocks in advance in uncritical timing situation. Leaned on a typical client-server architecture the application (client) can formulate some requests (commands) and can “send” it to the server via proprietary software interface. However in case of the 78K0R EEL driver command queuing is not implemented, its architecture and the chosen application interface are already prepared for that.

2.2.1 Request response model

The user application can use the pre-defined data type eel_request_t to define a private EEL request variable. Using the request variable the application and the EEL driver can exchange information. The owner task can formulate and “send” private requests to the EEL driver. On the other hand the EEL driver can use it for sending feedback to the “requester”. The request variable is synchronizing the application and the driver. Please have a look to chapter 3.7 for details.

2.2.2 Physical placement of the components

In the EEL driver the part where the data are stored is completely independent from the part where the driver code is stored. The EEL driver code is fully relocatable and can be linked into the application code as a pre-compiled library. The EEL pool area defined by the user must be located inside the lower 256K of the the user flash. Please take care, that EEL_POOL and application code does not overlap by creation of a suitable linker description file (*.DR for NEC linker or *.XCL for the IAR compiler).

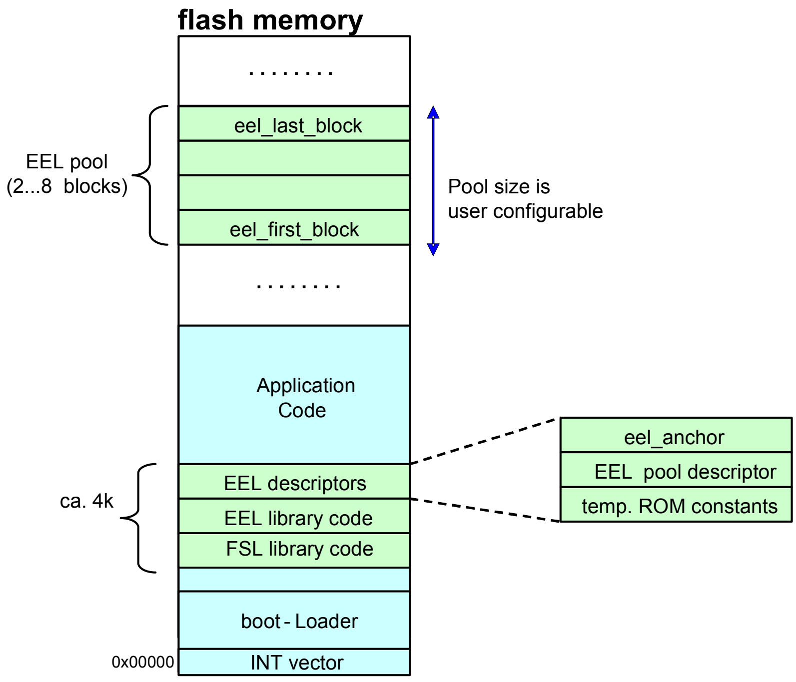

Figure 2-4 physical placement inside the device flash memory

2.2.3 EEL anchor

The ROM constant eel_anchor[...] contains the description of all variables the EEL driver can handle. It reserves unique identifier code for each EEL variable and defines its size expressed in bytes and flash-words. Other identifier than defined in eel_anchor[] cannot be used. Zero flash-word has to terminate the array eel_anchor[...]

The eel_anchor[var_no+1][4] is a two dimensional array of bytes. Both dimensions are defined automatically without user intervention. The only configurable parameter influencing the dimension is the constant EEL_VAR_NO. Each element of eel_anchor[N] is a complete EEL variable descriptor consisting of 4 bytes:

eel_anchor[N][0] EEL variable identifier (unique inside the eel_anchor). Valid range 0x01…0x7E (126 identifiers possible).

eel_anchor[N][1] EEL variable size expressed in flash word units. Valid range 0x01…0x40 (1..64 words possible).

eel_anchor[N][2] EEL variable size expressed in flash byte units (1…254) Valid range 0x01…0xFF (1..255 bytes possible).

eel_anchor[N][3] 0x00, limiter

eel_anchor[5][4]

id_1	wsize_1	bsize_1	0x00
id_2	wsize_2	bsize_2	0x00
id_3	wsize_3	bsize_3	0x00
id_4	wsize_4	bsize_4	0x00
0x00	0x00	0x00	0x00

Figure 2-5 EEL anchor structure (example)

2.2.4 EEL Pool

The EEL_pool is a flash area consists of 2-8 subsequent flash blocks where the EEL instances are stored. The flash blocks of the EEL-pool are organized like a ring where the “activated” status is passed to the next higher block by the refresh command (for details refer to the command description). When the last block is reached, the 1’st one becomes the “activated” one after “refresh”. To have at least one redundant block in spare that can be excluded in case of flash problems the recommended min. pool-size is 3 blocks. When no active block can be found by StartUp command the complete EEL_POOL has to be formatted by using the "format" command. All data are lost in that case.

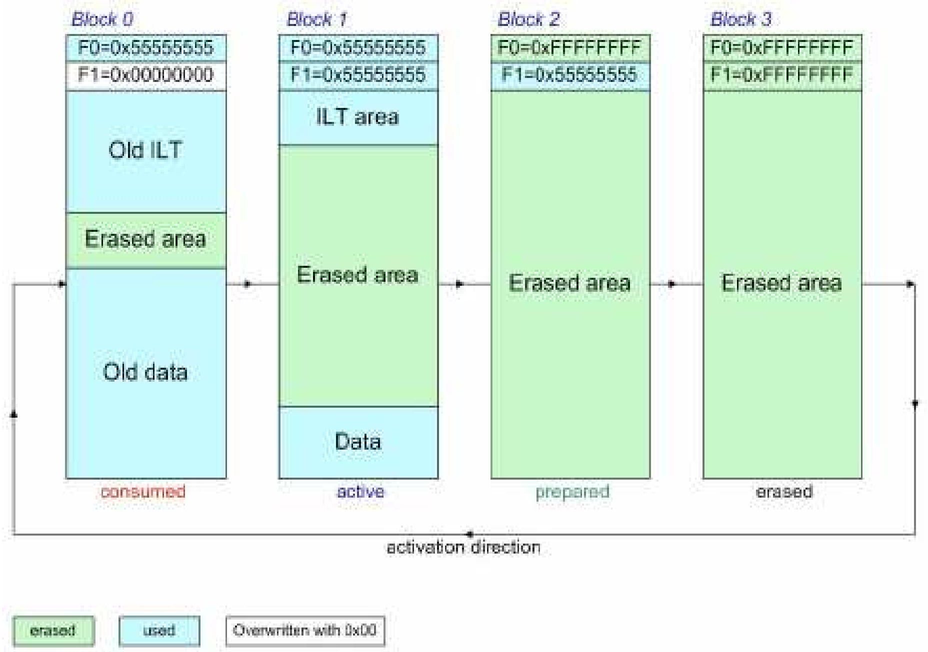

Figure 2-6 EELpool

2.2.5 EEL block

Each of the flash blocks belonging to the EEL_POOL is structured in the same wise. On the top two flash words are used as block status flags followed by the erase counter. Underlying the instance lookup table is used for stacking the instance references from the top to the bottom. At the bottom of the active block the referenced instances are stacked in the opposed direction. Both zones are running contrary to meet somewhere in the middle of the block. At least one "erased" flash word called separator has to isolate both zones. In case the separator is not available in the active block, the complete EEL_POOL hast to be formatted by using the "format" command. All data are lost in that case.

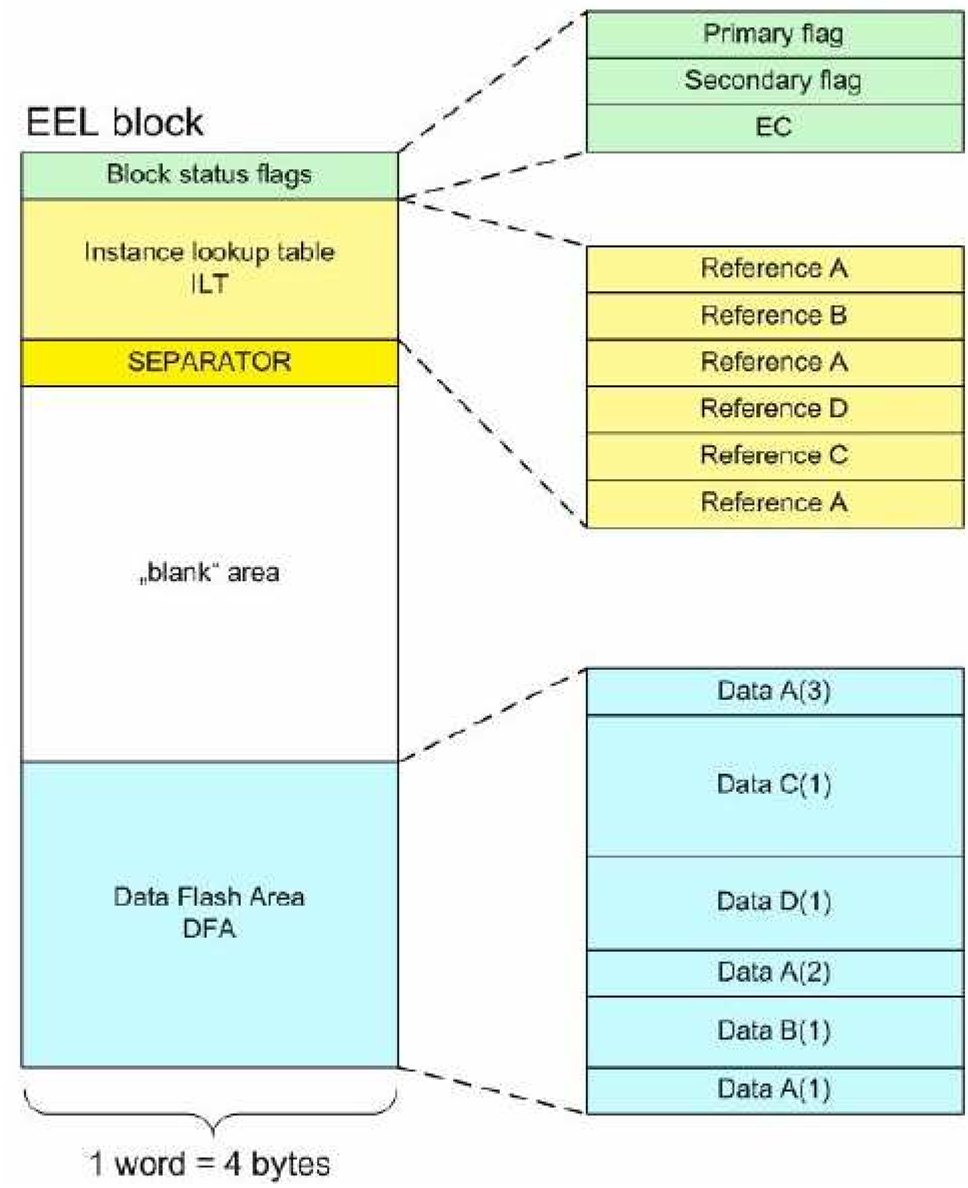

Figure 2-7 typical structure of an EEL block

2.2.6 EEL reference table

The EEL reference table is a simple RAM vector defined in eel_user.c. The number of EEL variables managed by the driver hast to be configured in the header-file eel_user.h

eel_u08 eel_reference[EEL_VAR_NO];

The read-pointer of each EEL variables is saved there to accelerate read access and makes the read access time independent of fill status of the active block. Linear search of the youngest instances is not necessary anymore. The reference table is initialized during startup command and updated in each write access. A refresh command is modifying the reference table to. Analogical, the separator word-index is used as a write pointer inside the active block to provide constant write access time. The separator widx is not visible to the user.

The reference table is automatically defined in eel_user.c. The user does not need to take care of the right dimension.

2.2.7 EEL instance lookup table

The Instance Lookup Table (ILT) is located at the beginning of the active block and realizes two main purposes:

- connection between the EEL variable identifier (identifier field) and its written instance (widx field)
- detection of problems caused by unexpected disturbance like asynchronous power on RESET, power supply voltage drop or others.

	bidx	widx	byte 0	byte 1	byte 2	byte 3	
	0	0		primary	flag		flag 0
	4	1		secondary	flag		flag 1
	8	2	EC(low)	EC(mid)	csum	EC(high)	erase cnt
	12	3	widx	wsize	CSUM	B(0)	ref
	16	4	widx	wsize	CSUM	A(0)	ref
	20	5	widx	wsize	CSUM	B(1)	ref
	24	6	widx	wsize	CSUM	C(0)	ref
	28	7	widx	wsize	CSUM	D(0)	ref
ILT	32	8	widx	wsize	CSUM	D(1)	ref
	36	9	widx	wsize	CSUM	C(1)	ref
	40	10	widx	wsize	CSUM	A(1)	ref
	44	11	widx	wsize	CSUM	B(2)	ref
	48	12	widx	wsize	CSUM	A(2)	ref
	52	13	widx	wsize	CSUM	A(3)	ref
	56	14	widx	wsize	CSUM	B(3)	ref
	60	15	widx	wsize	CSUM	A(4)	ref
	64	16	0xff	0xff	0xff	0xff	SEP
	68	17	0xff	0xff	0xff	0xff	empty
	72	18	0xff	0xff	0xff	0xff	empty
	76	19	0xff	0xff	0xff	0xff	empty

Figure 2-8 Instance lookup table (ILT) structure

Note:

Each write access is divided into two separated phases. In the first one the reference is written into the ILT to allocate the needed space inside the data flash area (DFA). In the second one the actual data of the written instance are written into the DFA.

2.2.8 EEL data flash area

The Data Flash Area (DFA) is located at the end of the active block and contains the pure data of the written instances only. The DFA is growing downstairs from higher addresses to lower addresses inside the active block.

908	227	0xff	0xff	0xff	0xff	empty
912	228	0xff	0xff	0xff	0xff	empty
916	229	A(4)			0xff	data
920	230	B(3)				data
924	231			0xff	0xff	data
928	232	A(3)			0xff	data
932	233	A(2)			0xff	data
936	234	B(2)				data
940	235			0xff	0xff	data
944	236	A(1)			0xff	data
948	237	C(1)				data
952	238					data
956	239					data
960	240		0xff	0xff	0xff	data
964	241	D(1)				data
968	242					data
972	243		0xff	0xff	0xff	data
976	244	D(0)				data
980	245					data
984	246		0xff	0xff	0xff	data
988	247	C(0)				data
992	248					data
996	249					data
1000	250		0xff	0xff	0xff	data
1004	251	B(1)				data
1008	252			0xff	0xff	data
1012	253	A(0)			0xff	data
1016	254	B(0)				data
1020	255			0xff	0xff	data
bidx	widx	active sector				

Figure 2-9 Data flash area (DFA) structure

Note:

The content of unused bytes inside the instance data is undefined

Chapter 3. User Interface (API)

The application programmer interface is completely defined in the header file eel.h. It contains all necessary constants, type definitions as well as all function prototypes interfacing the functionality of the EEL driver. The interface definition is fully C-compatible even though the implementation is done in assembler.

3.1 Constant definitions

All constants used for the EEL driver operation are available in form of enumeration types defined in eel.h. The meaning of all particular enumeration codes is described in the following chapter "Data type definitions".

3.2 Data type definitions

All data types used by the EEL driver are defined in the eel.h header file.

Table 3-1 overview of predefined data types

data_type	description
eel_block_status_t	enumeration type for coding the status of each EEL pool block
eel_command_t	enumeration type for coding the available commands
eel_status_t	enumeration type for coding the driver and request status
eel_error_t	enumeration type coding all possible errors
eel_request_t	structure type for definition of EEL request variables

3.3 Block status type

Each block of the EEL_POOL contains two 32-bit status flags F0 and F1 (primary and secondary flag) on its top. Both flags are coding all the block conditions can appear during EEPROM emulation. The data type eel_block_status_t reflects all relevant combination of both status flags.

Table 3-2 EEL block status code

block flag pattern	block status code	comment
F0=0xFFFFFFFF F1=0xFFFFFFFF	EEL_BLK_ERASED	status of a virgin erased block, means “block can be prepared”
F0=0xFFFFFFFF F1=0x555555555	EEL_BLK_PREPARED	block status after "prepare" or "format", means “block can be activated”
F0=0x55555555 F1=0x55555555	EEL_BLK_ACTIVATED	status of EEL block being “in use”, contains actual data
F0=0x55555555 F1=0x00000000	EEL_BLK_CONSUMED	block filled up with old instances, means “block can be prepared”
F0=0x00000000 F1=0x00000000	EEL_BLK_EXCLUDED	after problems the block was excluded by the application
other pattern than above	EEL_BLK_INVALID	unknown status, means "block has to be repaired"
blocks outside EEL pool	EEL_BLK_UNDEFINED	the specified block does not belong to the EEL pool area

Caution:

The block status reflects only the interpretation of both status flags F0 and F1. Power-on RESET during execution time of the erase command can produce scenarios where even though both flags are indicating the “erased” status, some words inside are not erased correctly. The only one way to ensure, that the status flags express the real physical status of the blocks is the positive termination of the STARTUP command.

3.4 Command code type

All commands provided by the EEL driver are represented by the enumeration type eel_command_t.

Table 3-3 EEL command codes

command code	comment
EEL_CMD_STARTUP	Electrical and logical plausibility checks of all blocks belonging to the EEL pool
EEL_CMD_WRITE	Writes the actual data of an EEL-variable from its mirror variable (RAM) into the active block of the EEL pool. Creates a new instance of the EEL-variable.
EEL_CMD_READ	the data of the youngest instance of the specified EEL variable are copied into its mirror-variable (RAM)
EEL_CMD_REFRESH	Copies the latest instances of all registered EEL variables into the next "fresh" block.
EEL_CMD_FORMAT	Create a virgin EEL-pool, all instances of EEL-variables are lost
EEL_CMD_PREPARE	the next “non-prepared” and “non-active” block is erased and marked as “prepared”
EEL_CMD_REPAIR	Erase and mark as “prepared” the specified block
EEL_CMD_EXCLUDE	Excludes the specified block from pool EEL management.
EEL_CMD_SHUTDOWN	Verifies the active block to ensure the data retention of the data inside. Disables read- and write-access.

3.5 Status type

The predefined type eel_status_t can be used for two different purposes. The first one is to indicate the status of the EEL request serviced by the driver. The other one meaning is to represent the internal status of the EEL driver themselves.

Table 3-4 Request and driver status codes

status code	related to	comment
EEL_STS_READY	request	the related request was finished successfully
	driver	the driver is ready to accept any EEL request
EEL_STS_ACTIVE	request	driver is processing a request in background mode
	driver	the driver is processing any request
EEL_STS_ABORTED	request	the request is not finished due to any problems
	driver	not relevant

Use cases:

1. The request-status can be used by the application (the requesting task) for checking the status of its own request (polling mode).
2. The EEL driver-status can be used by the application to check the availability of the EEL driver in advance.
3. Before entering the standby mode the application can check whether any EEL request is pending and can wait active until it’s finished.

Note:

The function EEL_CheckDriverStatus() can be used for 2) and 3).

3.6 Error type

All error codes supported by the driver during execution of EEL requests are collected in the predefined enumeration type eel_error_t. In case of problems the application can analyze the error-code to identify the reason.

Table 3-5 EEL error code

error code	meaning
EEL_OK	no error occurred during command execution
EEL_ERR_PARAMETER	parameter error (FSL) Note
EEL_ERR_PROTECTION	protection error (FSL) Note
EEL_ERR_ERASE	flash block could not been erased (FSL) Note
EEL_ERR_VERIFY	data couldn't be verified (FSL) Note
EEL_ERR_WRITE	data could not been written into flash (FSL) Note
EEL_ERR_EEPWRITE_VERIFY	verify error during writing into flash (FSL) Note
EEL_ERR_EEPWRITE_BLANK	area not blank, data remain untouched (FSL) Note
EEL_ERR_INTERRUPTED	self-prog. interrupted (FSL) Note
EEL_ERR_DRIVER_BUSY	the driver is currently busy with an other request
EEL_ERR_STARTUP_MISSING	read/write access is disabled as long "startup" isn’t executed successfully
EEL_ERR_POOL_CONSUMED	no “prepared” block available for “refresh” execution
EEL_ERR_BLOCK_CONSUMED	no space for the instance in the active block
EEL_ERR_FLMD0_LOW	FLMD0 signal error
EEL_ERR_TWO_ACTIVE_BLOCKS	startup found two EEL blocks marked as “active”
EEL_ERR_BLOCK_NOT_EMPTY	any “prepared” block contains undefined data.
EEL_ERR_BLOCK_INVALIDE	EEL block status flags couldn’t been interpreted
EEL_ERR_INSTANCE_UNKNOWN	specified EEL variable was never written
EEL_ERR_VARIABLE_CHECKSUM	checksum of the read instance does not match
EEL_ERR_NO_ACTIVE_BLOCK	startup could not recognize any “active” block
EEL_ERR_SEPARATOR_LOST	ILT and DFA not separated anymore
EEL_ERR_POOL_EXHAUSTED	less than two "healthy" blocks inside the EEL-pool
EEL_ERR_COMMAND_UNKNOWN	invalid command code detected
EEL_ERR_POOL_SIZE	invalid EEL pool configuration
EEL_ERR_VARIABLE_UNKNOWN	variable identifier not registered in eel_anchor[]
EEL_ERR_PROTECTED_BLOCK	block protected against "repair" or "exclude"

Note:

Error generated by the Flash Selfprogramming Library converted (FSL error code + 0x80) and passed directly to EEL.

3.7 Request type

Using the predefined request type the application can create request variables and use it for communication and synchronization purpose with the EEL driver. Status of the request can be polled and error code can be analyzed in case of any problems. It's practically the central point of interaction between the application and the EEL driver. The request data type is predefined in eel.h

```
// EEL request type (base type for any EEL access)
typedef struct
{
    eel_u08* address;            // 2, source/destination address
    eel_u08 identifier;          // 1, variable / block identifier
    eel_command_t command;       // 1, command has to be processed
    eel_status_t status;         // 1, status during/after execution
    eel_error_t error;           // 1, error after command execution
}eel_request_t;                  // -------------------------------
                                 // 6 bytes in total
```

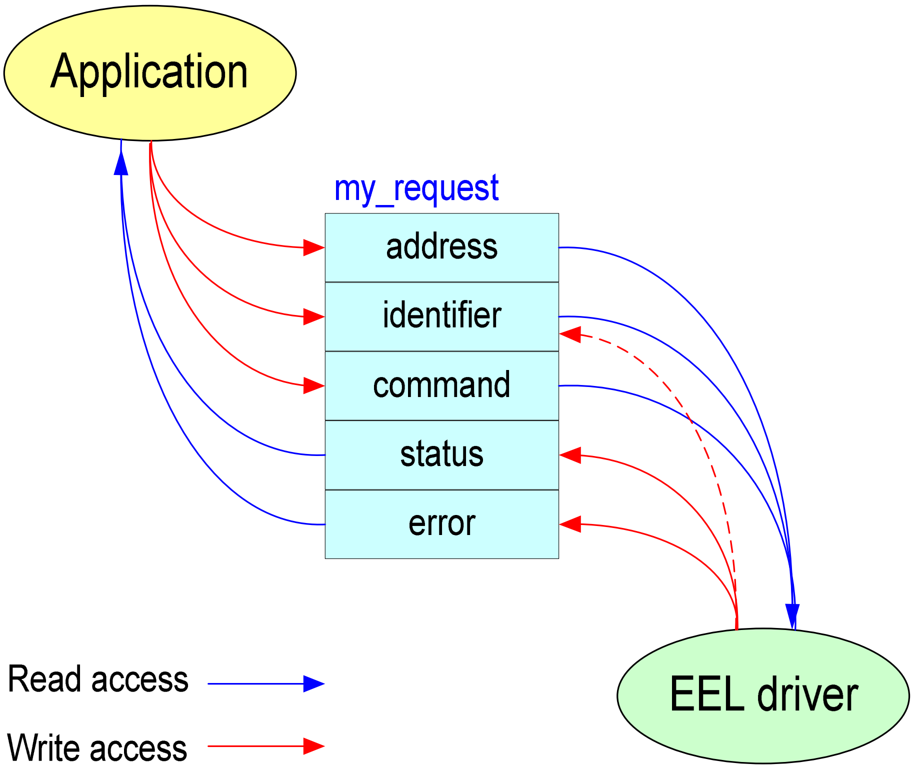

Figure 3-1 request variable synchronizing EEL and the application

Variables of that type can be used as a kind of common area where the application and the EEL driver can exchange information. The application formulates the request (command) and initiates the execution. The driver is returning the request status and error-code. The parameter-less commands like "startup" or "prepare" can return the related block number in block-field when problems occur.

3.8 Function prototypes

The functions offered by the EEL API are divided in operative and administrative. Operative functions are responsible for the pure request registration and request execution. Administrative functions can be used by the application to examine the status of some internal EEL parameters. Based on that information the application can always act and react in suitable and reasonable wise depending on its own context.

Table 3-6 EEL function overview

functions		
operative	administrative	description
EEL_Init()		driver initialization
EEL_Enforce(…)		request and execution in standalone applications
EEL_Execute(…)		request in real-time applications
EEL_Handler()		execution in real-time applications
	EEL_CheckDriverStatus()	returns the current EEL driver status
	EEL_GetPool()	returns the number of “prepared” blocks
	EEL_GetSpace()	returns the number of free words inside the active block
	EEL_GetBlockStatus(...)	returns the status of the specified block
	EEL_GetActiveBlock()	returns the number of the "active" block
	EEL_GetNextBlock()	returns the number of the subsequent, "non-excluded" block to the "active" one
	EEL_GetPrevBlock()	returns the number of the previous, "non-excluded" block to the "active" one

3.8.1 EEL_Init()

Description

This parameter less function should be used during power-on initialization of the device. All internal EEL variables are initialized but the driver remains inactive.

Interface

void EEL_Init(void)

Pre-condition

none

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
none		

Usage

EEL_init();

3.8.2 EEL_Open()

Description

This parameter less function is a user defined function which can be used to setup the system before using the EEPROM driver. For example a chopper can be started here. The macro to do this is already predefined. The usage of this function is optional

Interface

void EEL_Open(void)

Pre-condition

none

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
none		

Usage

EEL_Open();

3.8.3 EEL_Close()

Description

This parameter less function is a user defined function which can be used to setup the system after using the EEPROM driver. For example a chopper can be stopped here. The macro to do this is already predefined. The usage of this function is optional

Interface

void EEL_Close(void)

Pre-condition

none

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
none		

Usage

EEL_Close();

3.8.4 EEL_Enforce(my_request)

Description

This function can be used to execute any EEL command in so called “enforced" mode. This way of command execution is designed to support standalone applications. Such an application can call EEL_Enforce() to execute specified command and waits at the calling position until the command execution is completed. From the application point of view it works like a simple call, but it takes much “longer” time. During the command execution in “enforced" mode all enabled interrupts will be serviced with a delay specific for the self-programming commands used inside the EEL command.

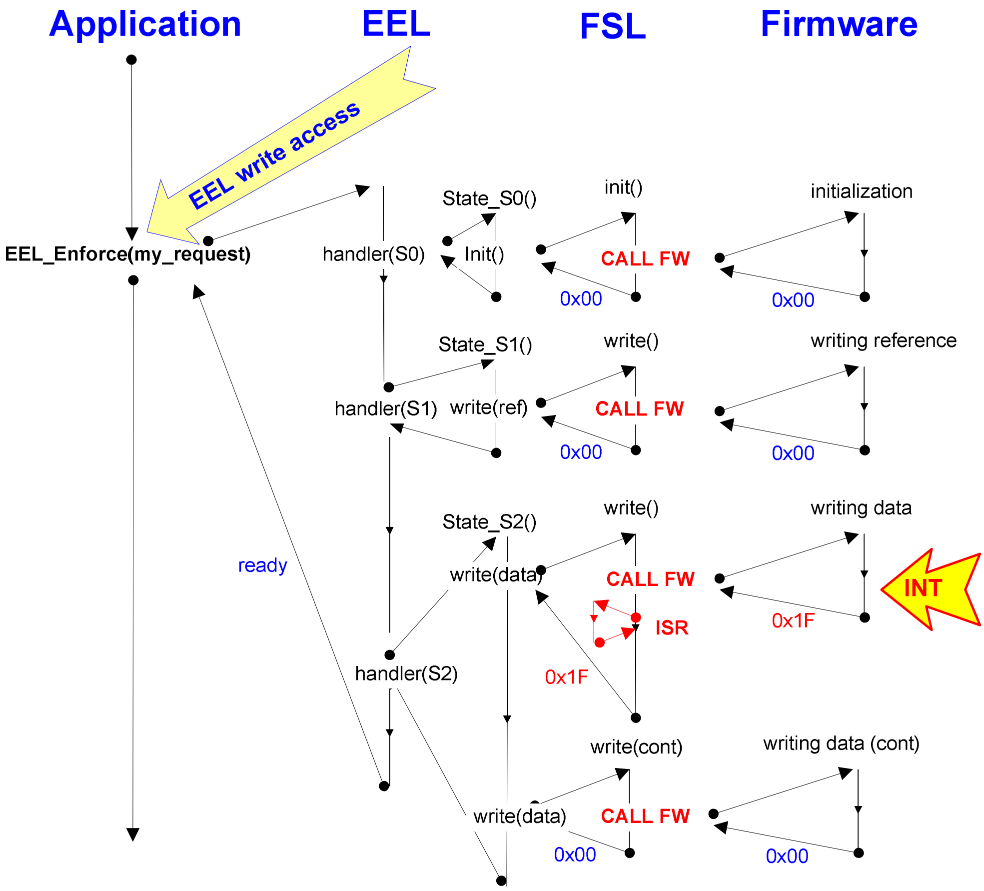

Figure 3-2 typical flow of an EEL_Enforce function

Interface

void EEL_Enforce(eel_request_t* request)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

- The arguments “request.address”, “request.identifier” and “request.command” from the request structure must be set.

Post-condition

Command finished successfully or with error

Argument

The argument of this function is a pointer to a request structure. Before calling this function the request structure must be filled. The parts of the structure which must be filled depend on the command which should be executed.. In the table below the arguments of the request structure are listed. All commands and their arguments inside the request structure are explained in detail in Chapter 4 Commands.

Argument	Type	Description
request.address	eel_u08*	depend on the command
request.identifier	eel_u08	depend on the command
request.command	eel_command_t	commands are described in chapter Chapter 4

Return types/values

The return values are also passed via the request structure. There is no direct parameter coming back. Also here the return types depends on the commands. A detailed description of the commands and their return types can be found in chapter Chapter 4 commands.

Argument	Type	Description
request.identifier	eel_u08	depend on the executed command
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	depends on the executed command

Usage

```
// “StartUp” commad request
my_eel_request.command = EEL_CMD_STARTUP;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

3.8.5 EEL_Execute(my_request)

Description

This function can be used to execute any EEL command in so called "background" mode. This way of command execution is designed to support real-time multitasking applications. Such applications reserve a time slice for the EEL task (process) and calls periodically EEL_Execute() to execute specified command in predetermined time pieces until the command execution is completed. The application can check the status of its own EEL request by polling the variable my_request.status. In between the time slices the application reclaims the fully control about the CPU and can manage time critical tasks, like watchdog reset or readout and release of communication-buffer. During the command execution in "background mode" all enabled interrupts will be serviced with a delay specific for the self-programming commands used inside the EEL command.

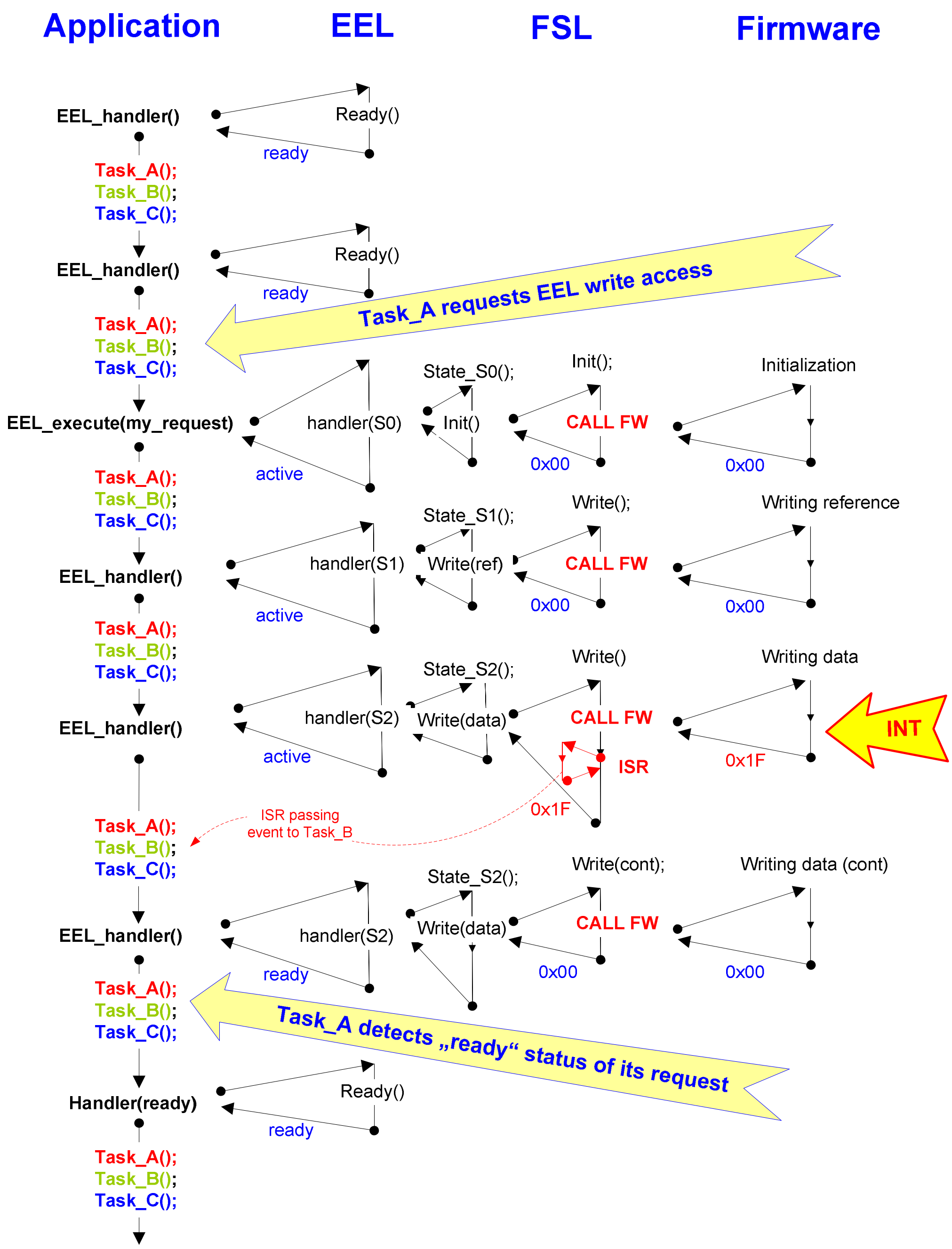

Figure 3-3 Typical flow of EEL command execution in “background mode”

Interface

void EEL_Execute(eel_request_t* request)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

- The arguments “request.address”, “request.identifier” and “request.command” from the request structure must be set.

Post-condition

The EEL_Handler() must be executed until the command is finished

Argument

The argument of this function is a pointer to the request structure. Before calling this function the request structure must be filled. The parts of the structure which must be filled depend on the command which should be executed. In the table below the arguments of the request structure are listed. All commands and their arguments inside the request structure are explained in detail in chapter Chapter 4 commands.

Argument	Type	Description
request.address	eel_u08*	depend on the command
request.identifier	eel_u08	depend on the command
request.command	eel_command_t	commands are described in chapter Chapter 4

Return types/values

The return values are also passed via the request structure. There is no direct parameter coming back . Also here the return types depends on the commands. A detailed description of the commands can be found in chapter Chapter 4 commands.

Argument	Type	Description
request.identifier	eel_u08	depends on the executed command
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	depends on the executed command

Usage

“StartUp” commad request
```
my_eel_request.command = EEL_CMD_STARTUP;
EEL_Execute(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_state = error_state;
if (my_eel_request.status == EEL_STS_ACTIVE) my_state = polling_state;
```

3.8.6 EEL_Handler()

Description

This function can be used to execute EEL command time-slice by time-slice in so called “background " mode. Typically, it should be called in the scheduler loop to share the CPU time with other processes. Theoretically the EEL_Handler() can also be called in a waiting-loop, but better to use EEL_Enforce() directly for such purpose. Have a look to the listing below that illustrates the typical use-case for the EEL handler.

Interface

void EEL_Handler(void)

Pre-condition

- The void EEL_Init(void) must be executed sucessfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- A command must be started with the function void EEL_Execute(eel_request_t* request)

Post-condition

Pending command may finished successfully or with error

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
none		

Usage

1)

```
// -----------------------------------------------
// OS scheduler's idle loop (cooperative system)
// -----------------------------------------------
do {
   EEL_Handler();
   if (task_A.tcb.status==active) (*task_A.tcb.state)();
   if (task_B.tcb.status==active) (*task_B.tcb.state)();
   if (task_C.tcb.status==active) (*task_C.tcb.state)();
   if (task_D.tcb.status==active) (*task_D.tcb.state)();
} while (true);
```

2)

```
// --------------------------
// possible, but not preferable
// --------------------------
my_eel_request.command = EEL_CMD_STARTUP;
```

```
EEL_Execute(&my_eel_request);
while (my_eel_request.status) == EEL_STS_ACTIVE
  {
  EEL_Handler();
  }
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

3.8.7 EEL_CheckDriverStatus()

Description

This function can be used to check the internal status of the EEL driver in advance before placing the EEL request. Also other administrative functions requiring "ready" status can use EEL_CheckDriverStatus() to check it.

Interface

eel_status_t EEL_CheckDriverStatus(void)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
request.status	eel_status_t	Status from the request EEL_STS_READY when he EEL driver is ready to accept a new request. EEL_STS_ACTIVE when the EEL driver is processing an other request in “background mode”

Note:

The status returned by this function cannot be “EEL_STS_ABORTED”. This value is reserved for request status only.

Usage

1) check in advance if request acceptable

```
// ---------------------------------------
if (EEL_CheckDriverStatus()==EEL_STS_READY)
  {
  my_eel_request.command = EEL_CMD_PREPARE;
  EEL_Execute(&my_eel_request);
  if (my_eel_request.status == EEL_STS_ABORTED)
my_EEL_ErrorHandler();
  };
```

2) check if driver "passive" before reading its parameter

```
// ---------------------------------------------------------
if (EEL_CheckDriverStatus()==EEL_STS_READY)
my_eel_space = EEL_GetSpace();
```

3.8.8 EEL_GetPool()

Description

This function provides the number of “prepared” coherent blocks inside the EEL pool. The application can use it to check the stock of prepared blocks in advance at most convenient time.

Interface

eel_u08 EEL_GetPool(void)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
function return value	eel_u08	provides the number of prepared, "ready to use" blocks

Usage

if (EEL_CheckDriverStatus()==EEL_STS_READY) my_eel_pool = EEL_GetPool();

3.8.9 EEL_GetSpace()

Description

This function provides the “free” space inside the active block. The application can use it to check if the available space is sufficient for the incoming write access, but it has to take care, that the driver is not “busy” at that time. The returned value represents the erased space inside the active block expressed in flash words (4 bytes)

Interface

eel_u08 EEL_GetSpace(void)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

Post-condition

none

Argument

Argument	Type	Description

Return types/values

Argument	Type	Description
function return value	eel_u08	provides free space inside the active block expressed in words

Usage

```
if (EEL_CheckDriverStatus()==EEL_STS_READY)
my_eel_space = EEL_GetSpace();
```

3.8.10 EEL_GetBlockStatus(my_block_u08)

Description

This function provides the status of the specified EEL block. It can be useful for reparation purpose when the application has to distinguish between active and other block.

Interface

eel_block_status_t EEL_GetBlockStatus(eel_u08 block)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

Post-condition

none

Argument

Argument	Type	Description
block number	eel_u08	expects the block number in hex

Return types/values

Argument	Type	Description
function return value	eel_block_status_t	returns status of specified block • EEL_BLK_ERASED • EEL_BLK_PREPARED • EEL_BLK_ACTIVATED • EEL_BLK_CONSUMED • EEL_BLK_EXCLUDED • EEL_BLK_INVALID • EEL_BLK_UNDEFINED

Usage

```
if (EEL_CheckDriverStatus()==EEL_STS_READY)
  {
  if (EEL_GetBlockStatus()==EEL_BLK_ACTIVATED)
    my_request.command = EEL_CMD_REFRESH;
  else
    my_request.command = EEL_CMD_PREPARE;
  }
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

3.8.11 EEL_GetActiveBlock()

Description

This function provides the number of currently "active" block. It can be useful for reparation purpose.

Interface

eel_u08 EEL_GetActiveBlock(void)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
function return value	eel_u08	returns the number of the currently active block in hex

Usage

```
if (EEL_CheckDriverStatus()==EEL_STS_READY)
  my_active_block = EEL_GetActiveBlock();
```

3.8.12 EEL_GetNextBlock()

Description

This function provides the number of next block to the currently "active" one. Blocks marked as "excluded" are ignored. It can be useful for reparation purpose.

Interface

eel_u08 EEL_GetNextBlock(void)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
function return value	eel_u08	returns the number of the next block to the active one in hex

Usage

```
if (EEL_CheckDriverStatus()==EEL_STS_READY)
   my_active_block = EEL_GetNextBlock();
```

3.8.13 EEL_GetPrevBlock()

Description

This function provides the number of previous block to the currently "active" one. Blocks marked as "excluded" are ignored. It can be useful for reparation purpose.

Interface

eel_u08 EEL_GetPrevBlock(void)

Pre-condition

- The void EEL_Init(void) must be executed successfully
- The command EEL_CMD_STARTUP must be executed successfully with the function “void EEL_Enforce(eel_request_t* request)” or “EEL_Execute(eel_request_t* request)”.
- The driver must be ready (return “EEL_STS_READY” from function “EEL_CheckDriverStatus(void)”)

Post-condition

none

Argument

Argument	Type	Description
none		

Return types/values

Argument	Type	Description
function return value	eel_u08	returns the number of the previous block to the active one in hex

Usage

```
if (EEL_CheckDriverStatus()==EEL_STS_READY)
my_active_block = EEL_GetPrevBlock();
```

Chapter 4. Commands

The available command codes are defined in the enumeration type eel_command_t. The EEL commands can be divided into two groups: operative (necessary for access to the virtual EEPROM) and administrative (used for administrative measures necessary for smooth and secure driver operation).

Table 4-1 EEL command overview

commands		
operative	**administrative**	**short description**
	EEL_CMD_STARTUP	plausibility check of the EEL, unlocks access to EEL
EEL_CMD _WRITE		write access to virtual EEPROM memory
EEL_CMD _READ		read access to virtual EEPROM memory
	EEL_CMD_PREPARE	formats of one additional (subsequent) block
	EEL_CMD_REFRESH	copy all recent instances into a fresh block
	EEL_CMD_REPAIR	formats of one dedicated block
	EEL_CMD_FORMAT	formats the complete EEL pool (all blocks), data are lost
	EEL_CMD_EXCLUDE	excludes one block from EEL pool management
	EEL_CMD_SHUTDOWN	electrical check of the active EEL block, locks the access to EEL

4.1 startup

The StartUp command is checking the plausibility and consistency of the EEL pool. This is the first command has to be executed before read- and write-access to the EEL is possible at all. Before the structure of each block can be analyzed logically, the electrical status of the information stored in the flash is checked by the self-programming command “verify”. After that the status of each block and its logical block structure can be analyzed. Finally the reference table (all EEL variable read pointer) and the separator-index (EEL write pointer) are initialized to achieve fast and constant read/write access time.

State sequence of the startup command:

1. plausibility check of EEL configuration data (descriptors in eel_user.c)
2. electrical verification of each block of the EEL pool
3. structure and status of the EEL pool
4. status and structure of each block

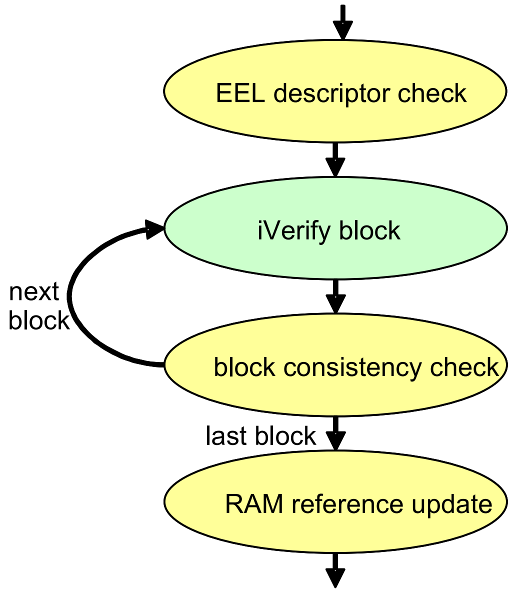

Figure 4-1 **sequence of the startup command**

Self-programming commands used by startup

- FSL_IVerify(fsl_u16 block_u16)

Startup request and feedback

To initiate the startup command the application has to specify the command code only. Any problems during startup execution time will be signaled in the request variable via its status and error members. In case of block related problems, the number of the affected block can be found in the identifier field. The application (its central error handler) has to take care for reparation. After that the startup command has to be executed again.

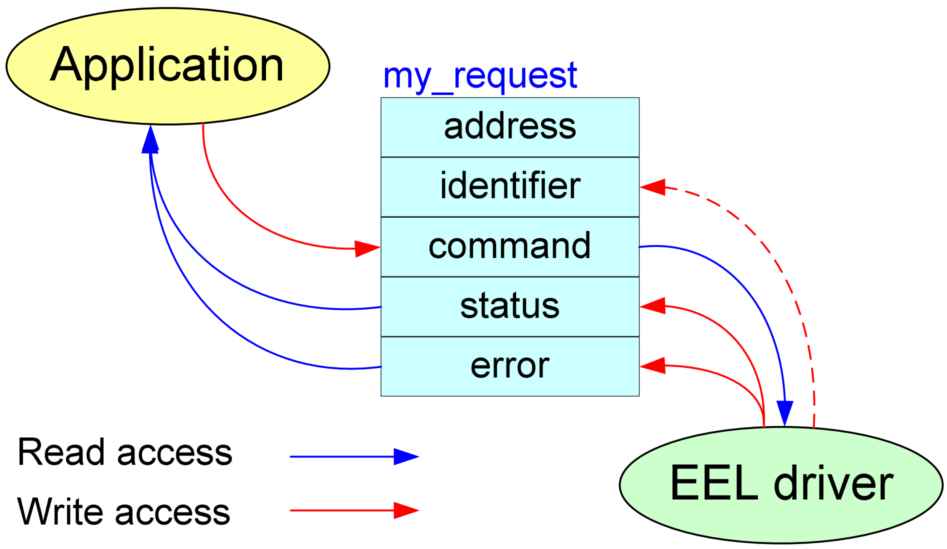

Figure 4-2 Startup request and feedback

Command input

Argument	Type	Description
request.comman d	eel_command_t	EEL_CMD_STARTUP

Command output

Argument	Type	Description
request.identifier	eel_u08	returns the block number in case of problems
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-2 Startup error code

Usage

```
// “StartUp” commad request
my_eel_request.command = EEL_CMD_STARTUP;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

Startup error handling

Whenever problems are detected during “startup”-command execution, the request status is modified by the EEL driver to “EEL_STS_ABORTED”. In such a case the application has to analyze the error code and initiate appropriate reparation. The possible error codes with corresponding reparation rules can be found below:

Table 4-2 Startup error code

startup error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_COMMAND_UNKNOWN	initial	meaning	no EEL variables known by the EEL
		reason	no identifiers inside eel_anchor[]
		remedy	register variables in eel_anchor[], recompile the project
EEL_ERR_BLOCK_INVALIDE	middle	meaning	block status flags F0/F1 could not be interpreted
		reason	probably RESET during writing F0/F1
		remedy	execute “repair” command and “startup” again.
EEL_ERR_BLOCK_NOT_EMPTY	middle	meaning	A “prepared” block is not “erased” anymore.
		reason	probably RESET during “refresh” command
		remedy	execute “repair” command and “startup” again.
EEL_ERR_VERIFY	middle	meaning	the level of some data dropped below the verify level
		reason	RESET during writing into flash, data retention
		remedy	1) affected block is the “active”–> execute “refresh”
			2) affected block is not “active” -> execute “repair”
EEL_ERR_TWO_ACTIVE_BLOCKS	middle	meaning	there are two “active” blocks detected in the EEL pool
		reason	RESET during execution of “refresh” command
		remedy	execute “repair” command and “startup” again.
EEL_ERR_SEPARATOR_LOST	heavy	meaning	ILT and DFA not separated anymore
		reason	EMI, malfunction, flash problems
		remedy	execute “format” , all data are lost

EEL_ERR_NO_ACTIVE_BLOCK	heavy	meaning	no “active” block detected during startup
		reason	possibly EMI, malfunction, flash problems
		remedy	execute “format”, all data are lost
EEL_ERR_POOL_EXHAUSTED	fatal	meaning	EEL pool consists of less than two “healthy” blocks
		reason	flash endurance exceeded
		remedy	none

4.2 write

The write command is writing a data-set of a registered EEL variable from its RAM mirror into the virtual EEPROM memory. The application has to specify the identifier and the starting address of the RAM-mirror variable before initiating the command execution.

Caution:

The EEL RAM mirror variables shouldn't be located in the short-address area 0xFFE20…0xFFEDF because this area cannot be used by the firmware as a data_buffer. To reduce the RAM consumption of the driver the EEL RAM variables are used directly as a kind of "temporary data-buffer" during the write command. Other user variables can be located in area 0xFFE20…0xFFEDF.

State sequence of the write command:

1. allocating the space for the data by writing the new reference into the ILT
2. writing the new data set into the allocated space inside the DFA
3. actualize the corresponding EEL variable reference inside the eel_reference[…]

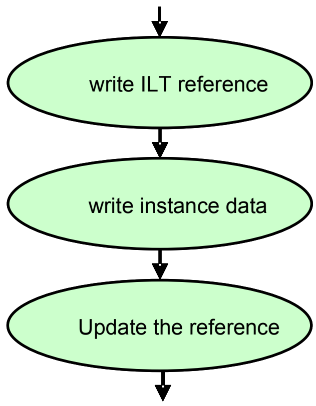

Figure 4-3 **sequence of the write command**

Self-programming commands used by write command

1. FSL_EEPROMWrite(my_addr_u32, my_wordcount_u08)

Write request and feedback

The application has to specify the EEL variable identifier of and the starting address of the RAM-mirror variable.

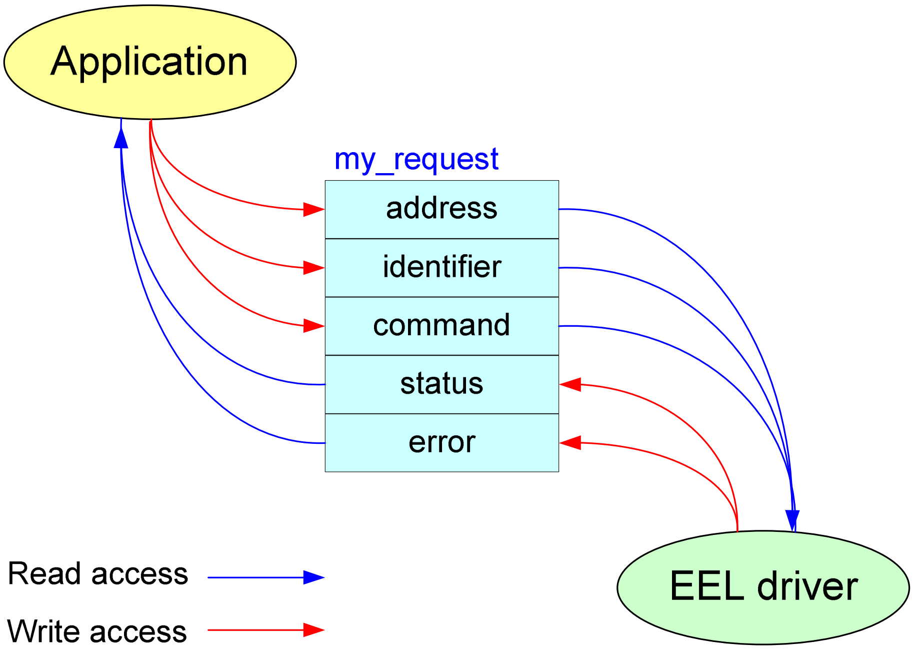

Figure 4-4 write request and feedback

Command input

Argument	Type	Description
request.address	eel_u08*	source address of the variable to be written
request.identifier	eel_u08	identifier of the variable to be written
request.command	eel_command_t	EEL_CMD_WRITE

Command output

Argument	Type	Description
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-3 Write error handling

Usage

```
// "Write" commad request
my_eel_request.command = EEL_CMD_WRITE;
my_eel_request.identifer = 'A';
my_eel_request.address = (eel_u08*)&A[0];
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

Write error handling

Whenever problems are detected during “write”-command execution, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react to properly to that exception. The possible error codes with recommended reparation rules can be found below:

Table 4-3 Write error handling

write error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_STARTUP_MISSING	normal	meaning	write access is not yet enabled
		reason	“startup” command wasn’t successful till now
		remedy	execute “startup” and “write” again
EEL_ERR_BLOCK_CONSUMED	normal	meaning	the active block is full
		reason	size of the EEL variable exceeds the available space
		remedy	execute the “refresh” command and “write” again
EEL_ERR_VARIABLE_UNKNOWN	initial	meaning	specified EEL variable unknown by the EEL driver
		reason	used identifier is not registered in eel_anchor[]
		remedy	register variable in eel_anchor[], recompile the project
EEL_ERR_FLMD0_LOW	unlikely	meaning	FLMD0 signal error
		reason	FLMD0 signal remains LOW during self-programming
		remedy	check FLMD0 hardware and software
EEL_ERR_PROTECTION	initial	meaning	EEL driver tries to overwrite protected area
		reason	possibly EMI, malfunction, bootcluster, flash shield
		remedy	check EEL pool range, flash shield
EEL_ERR_EEPWRITE_BLANK	heavy	meaning	the destination flash area in not erased enymore
		reason	possibly EMI, malfunction, flash problems
		remedy	execute the “refresh” command and“write” again
EEL_ERR_EEPWRITE_VERIFY	heavy	meaning	the written data could not be verified electrically
		reason	power, EMI or flash problem
		remedy	retry up to 3 times, if not successful: execute “refresh” and “exclude” the previous block and "write" again
EEL_ERR_WRITE	heavy	meaning	the data could not be written correctly into flash
		reason	power, EMI or flash problem
		remedy	retry up to 3 times, if not successful: execute “refresh” and “exclude” the previous block and "write" again
EEL_ERR_NO_ACTIVE_BLOCK	heavy	meaning	no “active” block found
		reason	possibly EMI, malfunction
		remedy	execute “format” command, all data are lost

4.3 read

The read command can be used to read the youngest written data-set from the virtual EEPROM memory into the specified RAM-mirror variable. Beside the command-code, the application has to specify the identifier of the EEL-variable and the starting address of the RAM-mirror variable.

State sequence of the read command:

The read command returns immediately the result of the read-access, independent whether “background” or “enforced” mode was used for the execution. The actual data-set is copied immediately from flash into the RAM mirror variable.

1. search active block
2. search the actual instance
3. check the checksum
4. copy data of the instance into RAM mirror variable

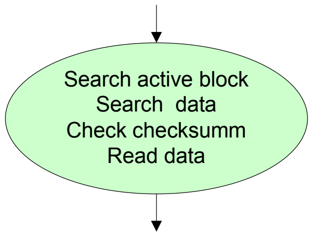

Figure 4-5 sequence of the read command

Self-programming commands used by read command

none FSL function is used

Read request and feedback

The application has to specify the EEL variable identifier and the starting address of the RAM-mirror variable.

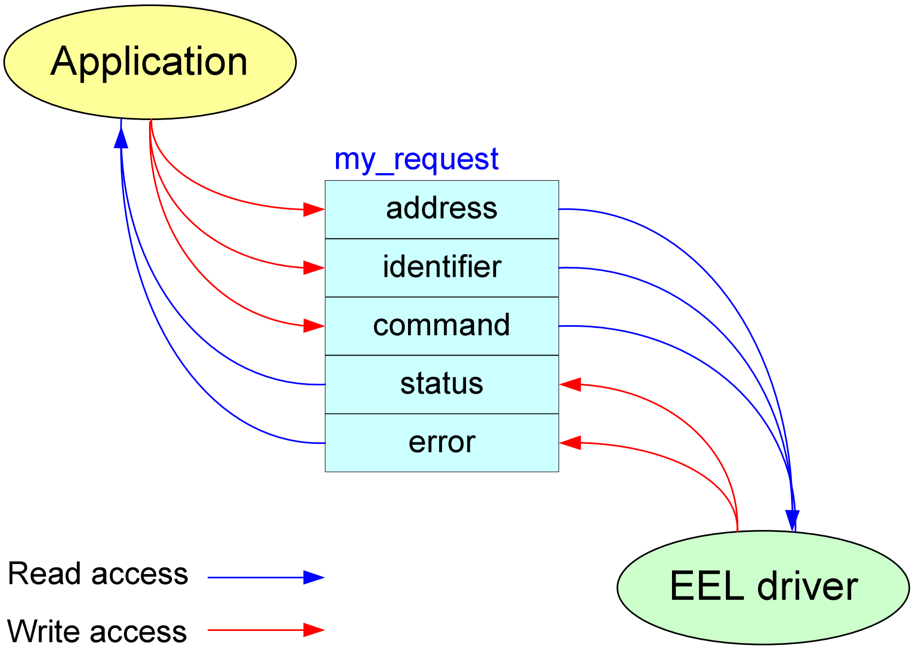

Figure 4-6 Read request and feedback

Command input

Argument	Type	Description
request.address	eel_u08*	destination address of the variable to be read
request.identifier	eel_u08	identifier of the variable to be read
request.command	eel_command_t	EEL_CMD_READ

Command output

Argument	Type	Description
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in **Error! Reference source not found.**

Usage

```
// "Read" commad request
my_eel_request.command = EEL_CMD_READ;
my_eel_request.identifer = 'A';
my_eel_request.address = (eel_u08*)&A[0];
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

Read error handling

Whenever problems are detected during "read"-command execution, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react properly to that exception. The possible error codes with corresponding reparation rules can be found below.

Table 4-4 Read error handling

read error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_STARTUP_MISSING	normal	meaning	read access is not yet enabled
		reason	“startup” command wasn’t successful till now
		remedy	execute “startup” and “read” again
EEL_ERR_VARIABLE_UNKNOWN	initial	meaning	specified EEL variable unknown by the EEL driver
		reason	used identifier is not registered in eel_anchor[]
		remedy	register variable in eel_anchor[], recompile the project
EEL_ERR_INSTANCE_UNKNOWN	normal	meaning	no data set found for the used EEL variable
		reason	the used EEL variable was never written into EEL
		remedy	initialize the EEL variable by writing initial value
EEL_ERR_VARIABLE_CHECKSUM	normal	meaning	checksum error detected
		reason	RESET during last EEL write access
		remedy	over-"write” the EEL variable with corrected value
EEL_ERR_NO_ACTIVE_BLOCK	heavy	meaning	no “active” block found
		reason	possibly EMI, malfunction
		remedy	execute “format” command, all data are lost

4.4 refresh

Each write access to the virtual EEPROM consumes some “erased” space within the active block as long as enough space is available. When the active block becomes full, the write access is not possible anymore. To enable the write access new space has to be created. The refresh command is doing that by copying the relevant information only from the currently "active" (full) block into a "prepared" (empty) one.

Especially when fast, immediately write access is required by the application, it has to take care for enough space inside the active block in advance. This can be performed by the application using the EEL_GetSpace() function and refresh command. Depending on the situation the application can realized it immediately or at more comfortable time in advance.

State sequence of the refresh command:

The refresh command is performed in three steps:

1. copy all latest instances of all registered EEL variables into the subsequent block.
2. mark the new, “fresh” block is marked as “active”
3. mark the old one block as “consumed”.

However asynchronous RESET during execution of the refresh command can produced some inconsistencies that has to be detected by the startup command and has to be repaired by the application.

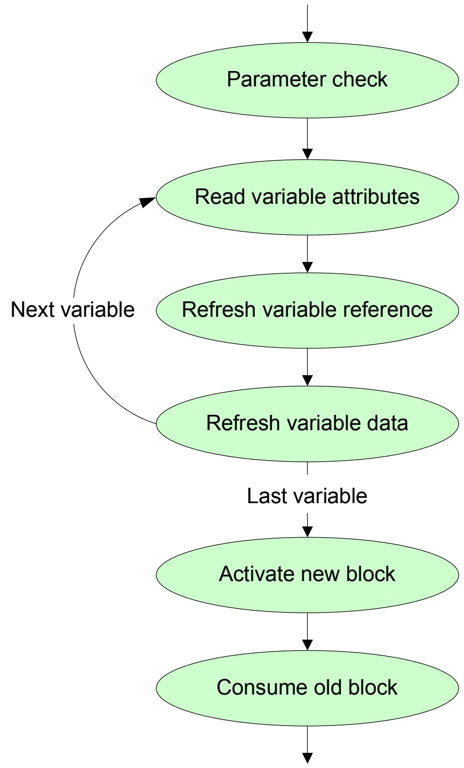

Figure 4-7 sequence of the refresh command

Self-programming commands used by refresh command

- FSL_EEPROMWrite(my_addr_u32, my_wordcount_u08)
- FSL_Write(my_addr_u32, my_wordcount_u08)

Refresh request and feedback

The application has to specify the command code only.

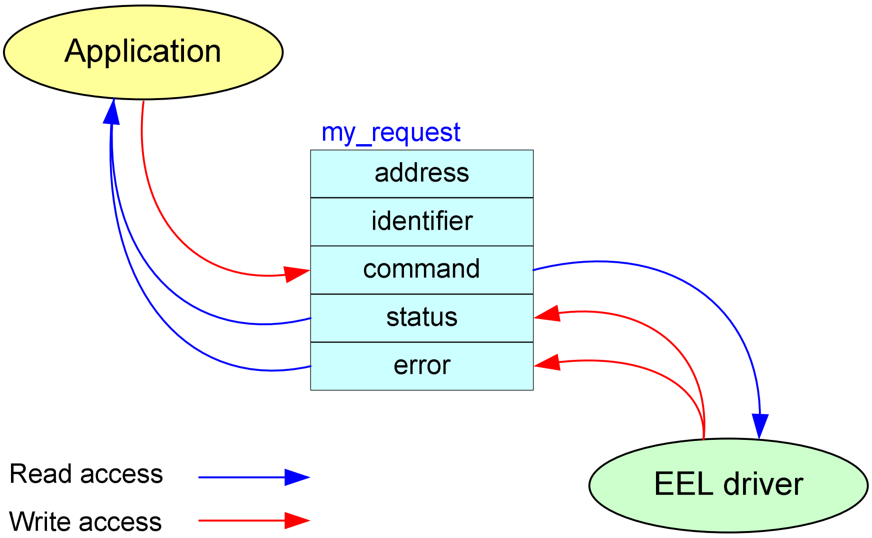

Figure 4-8 **Refresh request and feedback.**

Command input

Argument	Type	Description
request.comma nd	eel_command_t	EEL_CMD_ REFRESH

Command output

Argument	Type	Description
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-5 Refresh error handling

Usage

```
// “Refresh” commad request
my_eel_request.command = EEL_CMD_REFRESH;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

Refresh error handling

Whenever problems are detected during “refresh”-command execution, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react properly to that exception. The possible error codes with corresponding reparation rules can be found below:

Table 4-5 Refresh error handling

refresh error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task is running
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_POOL_CONSUMED	normal	meaning	no “prepared” block for "refresh"
		reason	all clocks were consumed or are invalid
		remedy	execute “prepare” and “refresh” again
EEL_ERR_FLMD0_LOW	unlikely	meaning	FLMD0 signal error
		reason	FLMD0 signal remains LOW during self-programming
		remedy	check FLMD0 hardware, software
EEL_ERR_VARIABLE_UNKNOWN	initial	meaning	specified EEL variable unknown by the EEL driver
		reason	used identifier is not registered in eel_anchor[]
		remedy	register variable in eel_anchor[], recompile the project
EEL_ERR_BLOCK_CONSUMED	normal	meaning	no space for initial data inside the “new” block
		reason	size of the EEL variable exceeds the available space
		remedy	reduce the amount of data (count or size of variables) and re-compile the project.
EEL_ERR_EEPWRITE_BLANK	heavy	meaning	the destination flash area is not erased anymore
		reason	possibly EMI, malfunction, flash problems
		remedy	"repair" next block and "refresh" again
EEL_ERR_EEPWRITE_VERIFY	heavy	meaning	the written data could not be verified electrically
		reason	possibly EMI, malfunction, flash problems
		remedy	“exclude” next block, execute "prepare" and "refresh" again
EEL_ERR_PROTECTION	initial	meaning	EEL driver tries to overwrite protected area
		reason	possibly EMI, malfunction, bootcluster, flash shield
		remedy	check EEL pool range, flash shield
EEL_ERR_WRITE	heavy	meaning	the data could not be written correctly into flash
		reason	possibly EMI, malfunction, flash problems
		remedy	“exclude” next block, execute "prepare" and "refresh" again
EEL_ERR_NO_ACTIVE_BLOCK	heavy	meaning	no “active” block found
		reason	possibly EMI, malfunction, flash problems
		remedy	execute “format” command, all data are lost

4.5 format

The format command is a command that should be used in exceptional cases only. It erases the whole EEL pool, marks one block as “activated” and all other blocks as “prepared”. All data in the previously “active” block are lost. The application has to take care for re-incarnation (initial write) of all EEL variable after “format”.

State sequence of the format command:

The format command erases the complete EEL pool block by block and writes the header data into each of it. The block next to the previously active one becomes "activated" after format. All other blocks becomes "prepared". The erase counter is managed by the format command.

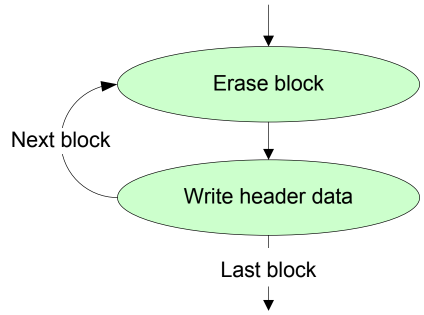

Figure 4-9 sequence of the format command

Self-programming commands used by format command

- FSL_Erase(fsl_u16 block_u16)
- FSL_EEPROMWrite(my_addr_u32, my_wordcount_u08)
- FSL_IVerify(fsl_u16 block_u16)

Format request and feedback

The application has to specify the command code only. When problems occur during execution the affected block number is provided by the driver in the identifier field of the request variable.

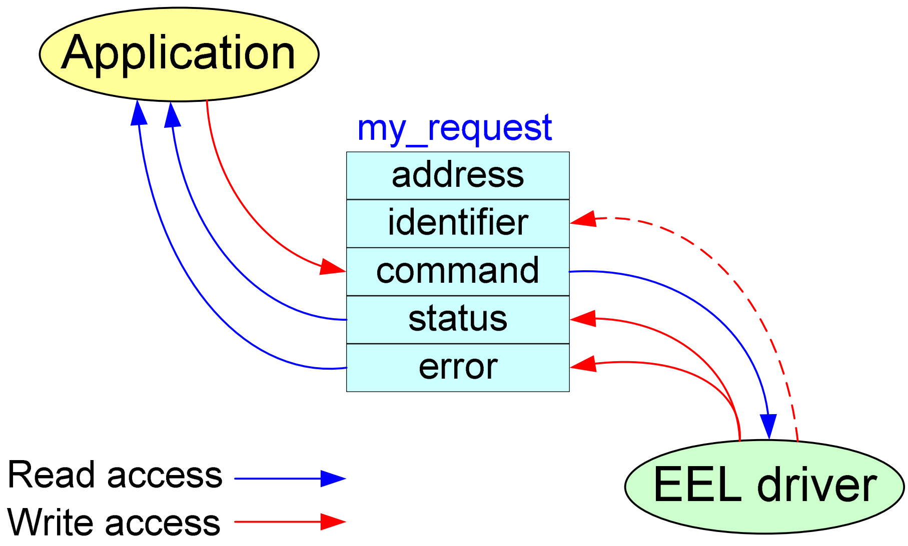

Figure 4-10 **Format request and feedback**

Command input

Argument	Type	Description
request.command	eel_command_t	EEL_CMD_FORMAT

Command output

Argument	Type	Description
request.identifier	eel_u08	returns the block number in case of problems
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-6 Format error handling

Usage

```
// “Format” commad request
my_eel_request.command = EEL_CMD_FORMAT;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

Format error handling

Whenever problems are detected during “format”-command execution, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react properly to that exception. The possible error codes with corresponding reparation rules can be found below:

Table 4-6 **Format error handling**

format error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_FLMD0_LOW	unlikely	meaning	FLMD0 signal error
		reason	FLMD0 signal remains LOW during self-programming
		remedy	check FLMD0 hardware and software
EEL_ERR_PROTECTION	initial	meaning	EEL driver tries to overwrite protected area
		reason	possibly EMI, malfunction, bootcluster, flash shield
		remedy	check EEL pool range, flash shield
EEL_ERR_EEPWRITE_BLANK	unlikely	meaning	the destination flash area is not erased anymore
		reason	possibly EMI, malfunction, flash problems
		remedy	depends on the origin of the problem
EEL_ERR_EEPWRITE_VERIFY	heavy	meaning	the written data could not be verified electrically
		reason	possibly EMI, malfunction, flash problems
		remedy	"exclude" the block and "format" again
EEL_ERR_WRITE	heavy	meaning	the data could not be written correctly into flash
		reason	possibly EMI, malfunction, flash problems
		remedy	"exclude" the block and "format" again
EEL_ERR_ERASE	heavy	meaning	the specified flash block could not be erased
		reason	possibly flash problems
		remedy	"exclude" the block and "format" again
EEL_ERR_POOL_EXHAUSTED	heavy	meaning	not enough available blocks in the pool (min. 2 necessary)
		reason	too many blocks excluded
		remedy	EEL dead

4.6 prepare

The prepare command looks for the next “consumed” or "erased" block inside the EEL pool, erases and marks it as “prepared”. The application can execute this command whenever it can take the liberty to spend time for that. In case that all blocks of the EEL pool are already “prepared” the command will terminate immediately with minimum of CPU load. It can be used by the application in advance, just to avoid driver blocking at uncomfortable time period. Blocks marked as “excluded” or “invalid” are over-jumped by the “prepare” command.

Especially when immediately write access is required by the application, it has to take care for availability of at least one “prepared” block for execution of unexpected “refresh” command. The administrative function EEL_GetPool() can be used for that purpose

State sequence of the prepare command:

The prepare command is performed in two steps: 1) searching and erasing of the next “consumed” or "erased" block. 2) marking of the erased block as “prepared”. However asynchronous RESET during execution of the refresh command can produced some inconsistencies that has to be detected by the startup command and has to be repaired by the application.

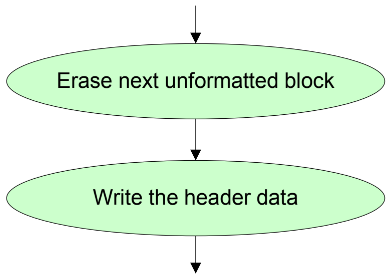

Figure 4-11 sequence of the prepare command

Self-programming commands used by prepare command

- FSL_Erase(fsl_u16 block_u16)
- FSL_EEPROMWrite(my_addr_u32, my_wordcount_u08)

Prepare request and feedback

The application has to specify the command code only. When problems occur during execution the affected block number is provided by the driver in the identifier field of the request variable.

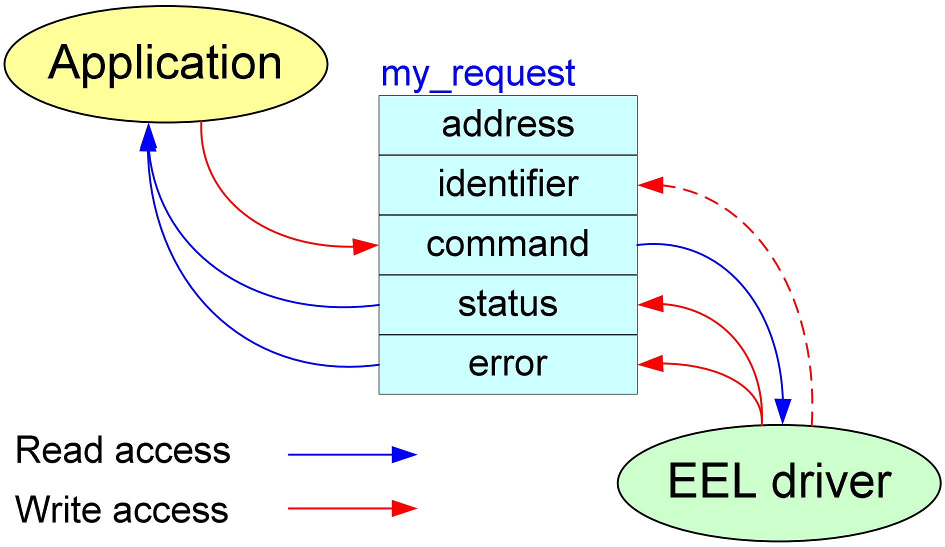

Figure 4-12 Prepare request and feedback

Command input

Argument	Type	Description
request.command	eel_command_t	EEL_CMD_PREPARE

Command output

Argument	Type	Description
request.identifier	eel_u08	returns the block number in case of problems
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-7 Prepare error handling

Usage

```
// "Prepare" commad request
my_eel_request.command = EEL_CMD_PREPARE;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

Prepare error handling

Whenever problems are detected during execution of the "prepare"-command, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react properly to that exception. The possible error codes with corresponding reparation rules can be found below.

Table 4-7 Prepare error handling

prepare error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_FLMD0_LOW	heavy	meaning	FLMD0 signal error
		reason	FLMD0 signal remains LOW during self-programming
		remedy	check FLMD0 hardware and software
EEL_ERR_EEPWRITE_BLANK	unlikely	meaning	the destination flash area is not erased anymore
		reason	possibly EMI, malfunction, flash problems
		remedy	depends on the origin of the problem
EEL_ERR_EEPWRITE_VERIFY	heavy	meaning	the written data could not be verified electrically
		reason	possibly EMI, malfunction, flash problems
		remedy	"exclude" the block and "prepare" again
EEL_ERR_PROTECTION	initial	meaning	EEL driver tries to overwrite protected area
		reason	possibly EMI, malfunction, bootcluster, flash shield
		remedy	check EEL pool range, flash shield
EEL_ERR_WRITE	heavy	meaning	the data could not be written correctly into flash
		reason	possibly EMI, malfunction, flash problems
		remedy	"exclude" the block and "prepare" again
EEL_ERR_ERASE	heavy	meaning	the specified flash block could not be erased
		reason	possibly flash problems
		remedy	"exclude" the block and "prepare" again
EEL_ERR_NO_ACTIVE_BLOCK	heavy	meaning	no “active” block detected
		reason	possibly EMI, malfunction, flash problems
		remedy	execute “format” command, all data are lost

4.7 repair

The repair command erases and marks as “prepared” a specific block. This command can be used for reparation purpose of undefined or inconsistent blocks.

State sequence of the repaire command:

The repair command is performed in two steps: 1) erasing of the specified block. 2) marking of the erased block as “prepared”. However asynchronous RESET during execution of the refresh command can produced some inconsistencies that has to be detected by the startup command and has to be repaired by the application.

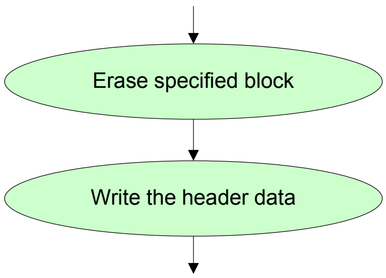

Figure 4-13 sequence of the repair command

Self-programming commands used by repair command

- FSL_Erase(fsl_u16 block_u16)
- FSL_EEPROMWrite(my_addr_u32, my_wordcount_u08)

Repair request and feedback

The application has to specify the command code and the block number has to be repaired.

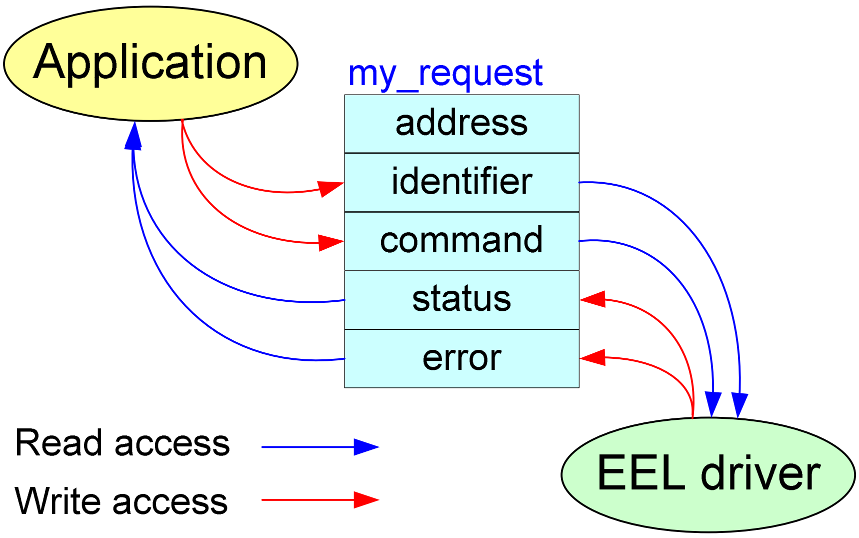

Figure 4-14 Repair request and feedback

Command input

Argument	Type	Description
request.identifier	eel_u08	block number to be repaired
request.command	eel_command_t	EEL_CMD_REPAIR

Command output

Argument	Type	Description
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-8 Repair error handling

Usage

```
// "Repair" commad request
my_eel_request.command = EEL_CMD_REPAIR;
my_eel_request.identifier= my_eel_block_u08;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
```

Repair error handling

Whenever problems are detected during execution of the “repair”-command, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react properly to that exception. The possible error codes with corresponding reparation rules can be found below:

Table 4-8 Repair error handling

repair error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_FLMD0_LOW	heavy	meaning	FLMD0 signal error
		reason	FLMD0 signal remains LOW during self-programming
		remedy	check FLMD0 hardware and software
EEL_ERR_PROTECTED_BLOCK	heavy	meaning	EEL driver tries to overwrite protected area
		reason	the block is “active” or it doesn't belong to EEL pool
		remedy	check EEL pool range
EEL_ERR_PROTECTION	initial	meaning	EEL driver tries to erase protected area
		reason	possibly EMI, malfunction, bootcluster, flash shield
		remedy	check/correct the flow, flash shield
EEL_ERR_EEPWRITE_BLANK	unlikely	meaning	the destination flash area is not erased anymore
		reason	possibly EMI, malfunction, flash problems
		remedy	depends on the origin of the problem
EEL_ERR_NO_ACTIVE_BLOCK	heavy	meaning	no “active” block detected
		reason	possibly EMI, malfunction, flash problems
		remedy	execute “format” command, all data are lost
EEL_ERR_EEPWRITE_VERIFY	heavy	meaning	the written data could not be verified electrically
		reason	possibly EMI, malfunction, flash problems
		remedy	"exclude" the block
EEL_ERR_WRITE	heavy	meaning	the data could not be written correctly into flash
		reason	possibly EMI, malfunction, flash problems
		remedy	“exclude” the block
EEL_ERR_ERASE	heavy	meaning	the specified flash block could not be erased
		reason	possibly flash problems
		remedy	"exclude" the block

4.8 exclude

The exclude command marks the specified block as “excluded”. It can be used for exclude any EEL block, except the active one, from the EEL block management. Data are not stored in that block anymore.

After a block was excluded the user has to check by a special routine if the EEPROM emulation has still enough available blocks to work properly.

The following flow should be used:

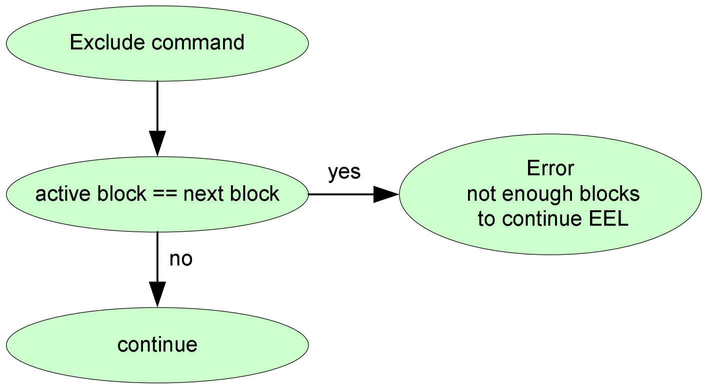

Figure 4-15 flow to use with the exclude command

To read the active and the next block pls. use the functions

- EEL_GetActiveBlock() and

- EEL_GetNextBlock()

Pls. refer also to “usage” of the exclude command.

State sequence of the exclude command:

The exclude command overwrites both status flags of the specified EEL block with the pattern 0x00000000. However asynchronous RESET during execution of the refresh command can produced some inconsistencies that has to be detected by the startup command and has to be repaired by the application.

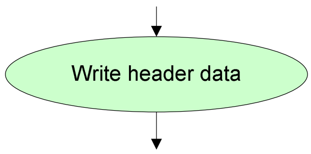

Figure 4-16 sequence of the exclude command

Self-programming commands used by exclude command

- FSL_Write(my_addr_u32, my_wordcount_u08)

Exclude request and feedback

The application has to specify the command code and the block number has to be excluded.

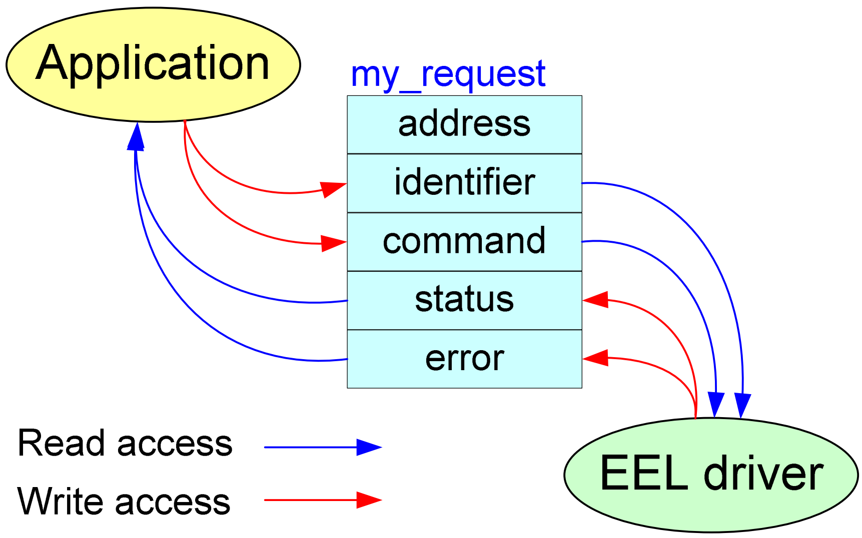

Figure 4-17 Exclude request and feedback

Command input

Argument	Type	Description
request.identifier	eel_u08	block number to be excluded
request.command	eel_command_t	EEL_CMD_EXCLUDE

Command output

Argument	Type	Description
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-9 Exclude error handling

Usage

```
// “Exclude” commad request
my_eel_request.command = EEL_CMD_EXCLUDE;
my_eel_request.identifier= my_eel_block_u08;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) my_EEL_ErrorHandler();
if ( EEL_GetActiveBlock() ==  EEL_GetNextBlock()) my_EEL_Dead_Error();
```

Exclude error handling

Whenever problems are detected during execution of the “shutdown”-command, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react properly to that exception. The possible error codes with corresponding reparation rules can be found below:
Furthermore it may happen that there are not enough block available after an exclude command. In this case the EEL Driver can not be used anymore.

Table 4-9 **Exclude error handling**

exclude error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already “busy” with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is “ready” and retry
EEL_ERR_FLMD0_LOW	heavy	meaning	FLMD0 signal error
		reason	FLMD0 signal remains LOW during self-programming
		remedy	check FLMD0 hardware and software
EEL_ERR_PROTECTED_BLOCK	heavy	meaning	EEL driver tries to overwrite protected area
		reason	the block doesn't belong to EEL pool
		remedy	check EEL pool range
EEL_ERR_PROTECTION	initial	meaning	the specified block cannot be excluded
		reason	possibly EMI, malfunction, bootcluster, flash shield
		remedy	check/correct the software flow, flash shield
EEL_ERR_WRITE	unlikely	meaning	block flags couldn’t be overwritten with 0x00000000
		reason	possibly flash problem
		remedy	None. Block status "excluded" could not be written correctly. F0/F1 are probably damaged and indicates a block status “invalid”. Blocks marked as “invalid” are treated as “excluded”

4.9 shutdown

The shutdown command verifies electrically the "active" block to ensure the maximum specified data retention for the EEL data.

State sequence of the shutdown command:

The shutdown performs the internal verify and marks the status of the driver as "non started up". In that status read and write access to the EEL in disabled.

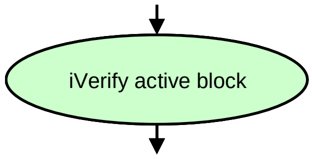

Figure 4-18 sequence of the shutdown command

Self-programming commands used by shutdown command

- FSL_IVerify(fsl_u16 block_u16)

Shutdown request and feedback

The application has to specify the command code only.

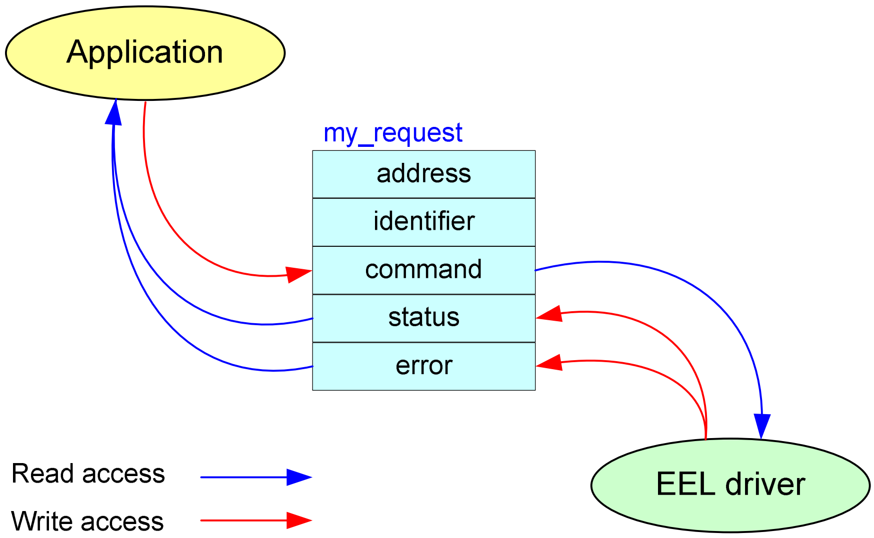

Figure 4-19 Shutdown request and feedback

Command input

Argument	Type	Description
request.command	eel_command_t	EEL_CMD_SHUTDOWN

Command output

Argument	Type	Description
request.status	eel_status_t	Status from the request EEL_STS_READY EEL_STS_ACTIVE EEL_STS_ABORTED
request.error	eel_request_t*	error codes are described in Table 4-10 Shutdown error handling

Usage

```
// "Shutdown" commad request
my_eel_request.command = EEL_CMD_SHUTDOWN;
EEL_Enforce(&my_eel_request);
if (my_eel_request.status == EEL_STS_ABORTED) EEL_ErrorHandler();
```

Shutdown error handling

Whenever problems are detected during execution of the "shutdown"-command, the request status is modified by the EEL driver to EEL_STS_ABORTED. In such a case the application can analyze the error code and react properly to that exception. The possible error codes with corresponding reparation rules can be found below:

Table 4-10 Shutdown error handling

shutdown error code	class	error background and handling	
EEL_ERR_DRIVER_BUSY	normal	meaning	driver is already "busy" with another request
		reason	another task already initiated its own request
		remedy	wait until driver-status is "ready" and retry
EEL_ERR_VERIFY	light	meaning	the active block could not be verified sucessfully
		reason	any data bit in "active" block dropped below the verify level
		remedy	execute "refresh" and "shutdown" again.
EEL_ERR_NO_ACTIVE_BLOCK	heavy	meaning	no "active" block detected during startup
		reason	possibly EMI, malfunction
		remedy	execute "format" command, all data are lost

Chapter 5. Implementation

5.1 Conditions

To keep the driver structure as small as possible the following conditions must be kept.

- The EEL library must be located inside the lowest 64k Flash.
- The EEL pool must be located inside the lowest 256k Flash.
- The max. number of user defined EEL variables must no exceed 126.
- The user must make sure that the RAM memory area used for the Self programming functionality will not be used from the application. For more details pls. refer to the corresponding device users manual.
- The FSL (Flash Selfprogramming library) must be included.

5.2 Resources

5.2.1 Resource distribution

The EEL is consuming some resources distributed across the device memory address space.

From compiler/linker point of view, the EEL driver consists of three logical parts:

- user configurable source file where all driver parameter are defined
- precompiled EEL library containing the whole driver functionality
- self-programming library interfacing the firmware

Each of the above parts claims resources as shown in the figure below:

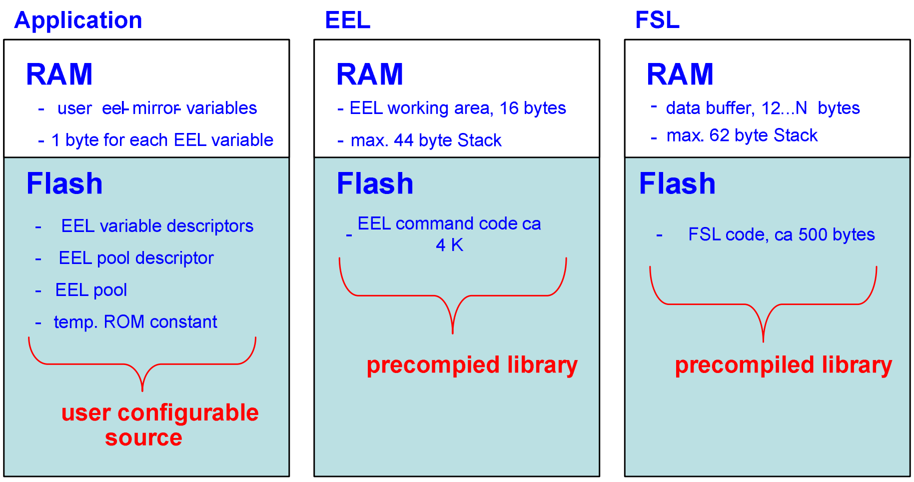

Figure 5-1 resource distribution

5.2.2 Resource consumption

The resource consumption is directly influenced by the number of required EEL variables and by the number of required flash blocks of the EEL pool

Table 5-1 resource consumption

object	type	size	unit	comment
fsl_data_buffer	RAM	min. 12	byte	allocated by the FSL (user configurable)
EEL_driver data	RAM	16	byte	internal EEL driver variables
eel_references	RAM	N	bytes	1 byte for each EEL variable
Stack	RAM	additional 106 b. max	bytes	Additional to the user stack including the FSL stack consumption.
EEL_pool	ROM	max.8k min. 2k	byte	EEL data area inside the code flash
EEL_descriptors	ROM	8	byte	EEL driver descriptors
eel_anchor	ROM	4* (N+1)	byte	EEL variable descriptors
EEL_code	ROM	3,4k	byte	EEL driver functionality

Note:

“N” describes the number of variables

5.3 Timing

The execution time of the functions and commands is influenced by following factors:

- execution time of used self-programming commands
- eventually necessity of firmware internal retries of erase- or write-pulses
- number of blocks used as virtual EEPROM
- status of the blocks like number of variables, size of variables...
- time consumed by the application in between the chopper time-slices. Only if the chopper is used.
- time consumed in the interrupt services. Only if interrupts are enabled.
- overhead time caused by the interruption of self-programming commands. Only if interrupts are enabled.

All these factors make it difficult to specify the real execution time of the EEL commands absolutely. To get an impression of the executions times for a typical scenario you can use the following typical times

5.3.1 Typical timings

Conditions:

Device: uPD781012 (78K0R/KF3-L)

Frequency: 20 MHz

Table 5-2 typical timing of commands

command	condition	non interrupted	interrupted each 1ms
EEL_CMD_STARTUP	8 blocks	59.48ms	64,7ms
	4 blocks	29,8ms	32,14ms
	2 blocks	13,6ms	15,13ms
EEL_CMD_WRITE	1 byte	2,00 ms	2,11ms
	13 bytes	2,28 ms	2,41ms
	27 bytes	2,53ms	2,84ms
EEL_CMD_READ	1 byte	0,18ms	0,19ms
	13 bytes	0,22ms	0,238ms
	27 bytes	0,27ms	0,286ms
EEL_CMD_PREPARE	1 block	14,02ms	15,76ms
EEL_CMD_REFRESH	4 EEL variables	21,6ms	24,18ms
EEL_CMD_REPAIR	1 block	14ms	15,63ms
EEL_CMD_FORMAT	8 blocks	154,5ms	173,47ms
	4 blocks	77,34ms	86,3ms
	2 blocks	38,6ms	43,26
EEL_CMD_EXCLUDE	1 block	0,7ms	0,73ms
EEL_CMD_SHUTDON	1 block	5,73ms	6,34ms

Table 5-3 typical timing of functions

functions	condition	timing
EEL_Init()	all	800us
EEL_CheckDriverStatus()	all	6us
EEL_GetPool()	8 blocks	400us
	4 blocks	190us
	2 blocks	91us
EEL_GetSpace()	all	300us
EEL_GetBlockStatus(...)	all	22us
EEL_GetActiveBlock()	8 blocks	154us
	4 blocks	72us
	2 blocks	40us
EEL_GetNextBlock()	8 blocks	155us
	4 blocks	90us
	2 blocks	56us
EEL_GetPrevBlock()	8 blocks	155us
	4 blocks	90us
	2 blocks	56us

Note:

This timing information is not a specification. They are typical values measured on one specific device in the laboratory. In reality the EEL command execution time can be influenced by the retry-counters inside the used FSL commands like FSL_Write(...), FSL_EEPROMWrite(...) and FSL_Erase(...). For details please refer to the users manual of the target device or the users manual of the FSL library.

5.3.2 Maximum timings

The max timing values are mainly defined by the max selfprogramming executions times. In the following table you will formulas to calculate the typ. and max calculation times of the EE functions and commands.

Following abbreviations are used.

From EEL

These times depend on the user setup of the EEL (e.g number of blocks, number of variables,..)

blk_num Number of used blocks for the EEL_pool

var_num Number of used variables

var_size_word Variable size expressed in words (4 byte units)

all_var_size_word Size of all variables expressed in words (4 byte units)

t_user_chopper_int user predefined chopper time

t_user_int_execution execution time of the chopper int routine

From FSL

These times depends on the self programming lib. You can find these times in the FSL (flash self programming library) user manual of the corresponding device.

t_fsl_init execution time of the FSL_Init

t_fsl_iverify execution time of the FSL_IVerify

t_fsl_EEPwrite(x) execution time of the FSL_EEPROMWrite where x is the number of words to be written

t_fsl_erase execution time of the FSL_Erase

t_fsl_write execution time of the FSL_Write

t_fsl_int_response time until the int. service is reached from the of firmware

t_fsl_int_suspension time until the application is reached from the end of the int. service routine

Table 5-4 **timing of functions**

functions	timing
EEL_Init()	t_fsl_init + t_fsl_open + t_fsl_close + 100/fcpu +(400/fcpu * blk_num)
EEL_CheckDriverStatus()	220/fcpu
EEL_GetPool()	1200/fcpu * blk_num
EEL_GetSpace()	20000/fcpu
EEL_GetBlockStatus(...)	500/fcpu
EEL_GetActiveBlock()	1000/fcpu +(400/fcpu * blk_num)
EEL_GetNextBlock()	1000/fcpu +(400/fcpu * blk_num)
EEL_GetPrevBlock()	1000/fcpu +(400/fcpu * blk_num)

Table 5-5 **timing of commands used with EEL_Enforce(..)**

command	timing
EEL_CMD_STARTUP	((t_fsl_iverify + (62000/fcpu))*blk_num) +20000/fcpu+ (all_var_size_word *(450/fcpu)) + (var_num *(400/fcpu))
EEL_CMD_WRITE	t_fsl_EEPwrite(1)+ (t_fsl_EEPwrite(var_size_word)+ 16000/fcpu
EEL_CMD_READ	2000/fcpu +(400/fcpu * blk_num) +(380/fcpu * var_size_word)
EEL_CMD_REFRESH	(31400/fcpu + t_fsl_EEPwrite(1)) * (all_var_size_word + var_num)
EEL_CMD_FORMAT	(t_fsl_erase + t_fsl_EEPwrite(3) + t_fsl_iverify + 9500/fcpu) * blk_num
EEL_CMD_PREPARE	(t_fsl_erase + t_fsl_EEPwrite(3) + 8000/fcpu)
EEL_CMD_REPAIR	(t_fsl_erase + t_fsl_EEPwrite(3) + 6500/fcpu)
EEL_CMD_EXCLUDE	(t_fsl_write (2) + 3000/fcpu)
EEL_CMD_SHUTDOWN	(t_fsl_iverify + 1500/fcpu +(400/fcpu * blk_num))

Table 5-6 timing of commands used with EEL_Execute(..)

command	timing
EEL_CMD_STARTUP	1200/fcpu* blk_num
EEL_CMD_WRITE	6000/fcpu + t_fsl_EEPwrite(1)
EEL_CMD_READ	2000/fcpu +(400/fcpu * blk_num)+(380/fcpu * var_size_word)
EEL_CMD_REFRESH	20000/fcpu
EEL_CMD_FORMAT	4000/fcpu
EEL_CMD_PREPARE	7500/fcpu
EEL_CMD_REPAIR	4200/fcpu
EEL_CMD_EXCLUDE	1000/fcpu
EEL_CMD_SHUTDOWN	1500/fcpu + (400/fcpu * blk_num) + t_fsl_iverify

Table 5-7 timing of commands continued with EEL_Handler()

command	timing
all	t_user_chopper_int + t_fsl_int_response + t_user_int_execution +fsl_int_suspension + 200/fcpu

5.4 File structure

There are two platform specific versions of the 78K0R EEL driver. Both libraries are precompiled offers the same application programming interface. The file structure and hierarchy is also widely the same.

5.4.1 NEC compiler compatible version

This EEL driver version is supporting the compiler version 1.20 and later.

Package content:

eel.h - API definition header file

eel.lib - precompiled EEL library

eel_user.h - user configurable definitions of the EEL driver

eel_user.c - user configurable part of the EEL driver

eel_pool.c - EEL pool area allocation

eel_sample_linker_file.dr - linker description file

5.4.2 IAR compiler compatible version

This EEL driver version is supporting the compiler version 4.00 and later.

Package content:

eel.h - API definition header file

eel.r26 - precompiled EEL library

eel_user.h - user configurable definitions of the EEL driver

eel_user.c - user configurable part of the EEL driver

eel_pool.c - EEL pool area allocation

eel_sample_linker_file.xcl - linker description file (debug version)

Chapter 6. EEL driver integration

6.1 Link time configuration

Before the EEL driver can be integrated several parameter has to be configured. Following files are touched by this configuration: fsl_user.h, eel_user.h, eel_user.c and linker description file (*.xcl or *.dr file)

6.1.1 Flash self-programming library configuration

The self-programming library is used by the EEPROM emulation to access the flash memory. Be aware, that the chosen configuration is valid and constant for the operation time of each self-programming command. The functions FSL_Open () and FSL_Close() are called directly by the EEL driver and its handler.

- copy all the files into your project subdirectory
- add all fsl*.* files into your project (IDE or make-file) fsl_user.h

constants:

- FSL_SYSTEM_FREQUENCY, adapt the system frequency expressed in [Hz]
- FSL_LOW_VOLTAGE_MODE, define whether low-voltage mode is used or not
- FSL_DATA_BUFFER_SIZE, the min. size for correct EEL operation is 12 bytes
- FSL_MKmn_MASK, interrupt configuration during firmware operation, effective when FSL_INT_BACKUP defined

macros:

- FSL_FLMD0_HIGH,
- FSL_FLMD0_LOW, assignment of the FLMD0 control, could be an internal register or external port

controls:

- FSL_INT_BACKUP, determines whether a specific interrupt scenario defined as FSL_MKmn_MASK is effective during firmware operation

fsl_user.c

code:

- FSL_Open(), adapt the functions due to the requirements
- FSL_Close()

6.1.2 EEPROM library configuration

Before being integrated into the user application, the EEPROM emulation must be adapted to the user requirements. The main parameters are memory mode and the number and size of EEL variables.

- copy all the files into your project subdirectory
- add all eel*.* files into your project (IDE or make-file)

eel_user.h

constants:

- EEL_FIRST_BLOCK, assignment of the EEL pool, first block definition
- EEL_LAST_BLOCK, assignment of the EEL pool, last block definition
- EEL_VAR_NO, number of EEL variables handled by the EEL driver macros:
- EEL_START_CHOPPER, starts the periodical interrupt used as a chopper [Note]
- EEL_STOP_CHOPPER, stops the chopper interrupt [Note]
- import all RAM-variables (types) you want to write into the "virtual EEPROM"

Note:

The chopper must be only used when the Background mode (Handler()) is used. In the enforced mode it is not necessary to use the chopper.

eel_user.c

ROM constants:

- eel_anchor[…][4], fill in description data of all your EEL variables (identifier and size) code:
- EEL_Open(), is a user definded function to adapt the requirements before starting the EEL driver. For example the chopper can be started here. The use of this function is optional.
- EEL_Close(), is a user definded function to adapt the requirements after closing the EEL driver. For example the chopper can be stopped here. The use of this function is optional.

Note

In module eel_user.c some configuration checks are realize to omit contradictory configuration parameters. Following errors can be generated during compilation:

- EEL ERROR=003, EEL data flash size invalid! At most 8 blocks can be assigned as a virtual EEPEOM area in code flash.
- EEL ERROR=004, at least 1 EEL-variable has to be registered in eel_anchor[..] !

6.1.3 Linker configuration

Using the linker description file the application can bind necessary logical segments the EEL requires to absolute addresses the application can offer. Examples of platform specific linker description files are part of the installation package:

- eel_sample_linker_file.DR for NEC's CC78K0R tool chain
- eel_sample_linker_file.XCL for the IAR platform.

Required logical segments are:

- FSL_CODE code segment where the FSL code is located
- EEL_POOL specify the address space of the EEL pool

6.2 Run-time configuration

6.2.1 EEL activation and deactivation sequence

One important issue in systems using EEPROM (internal, external as well EEL) is the integrity and consistency of the stored data. In all the systems listed above the write access takes certain time an unexpected RESET or power break can produce data inconsistencies. To ensure proper EEL driver operation and to discover potential inconsistencies in an early stage the following activation sequence must be used:

Startup phase:

- execute EEL_Init(), power-on initialization, driver remains passive
- execute EEL_Open(), activation of chopper interrupt and other user defined functionalities [Note]
- execute the startup-command to check the driver consistency. In case of problems process suitable reparation and repeat step

Normal operation:

- execute of any defined EEL driver function or commands

Shutdown phase:

- execute of shutdown-command to ensure the max. data retention.
- execute EEL_Close(), deactivate the chopper and other user defined functionalities [Note].
- EEL driver is inactive, standby mode can be entered

Note:

The EEL_Open() and EEL_Close() functions are user defined and not mandatory.

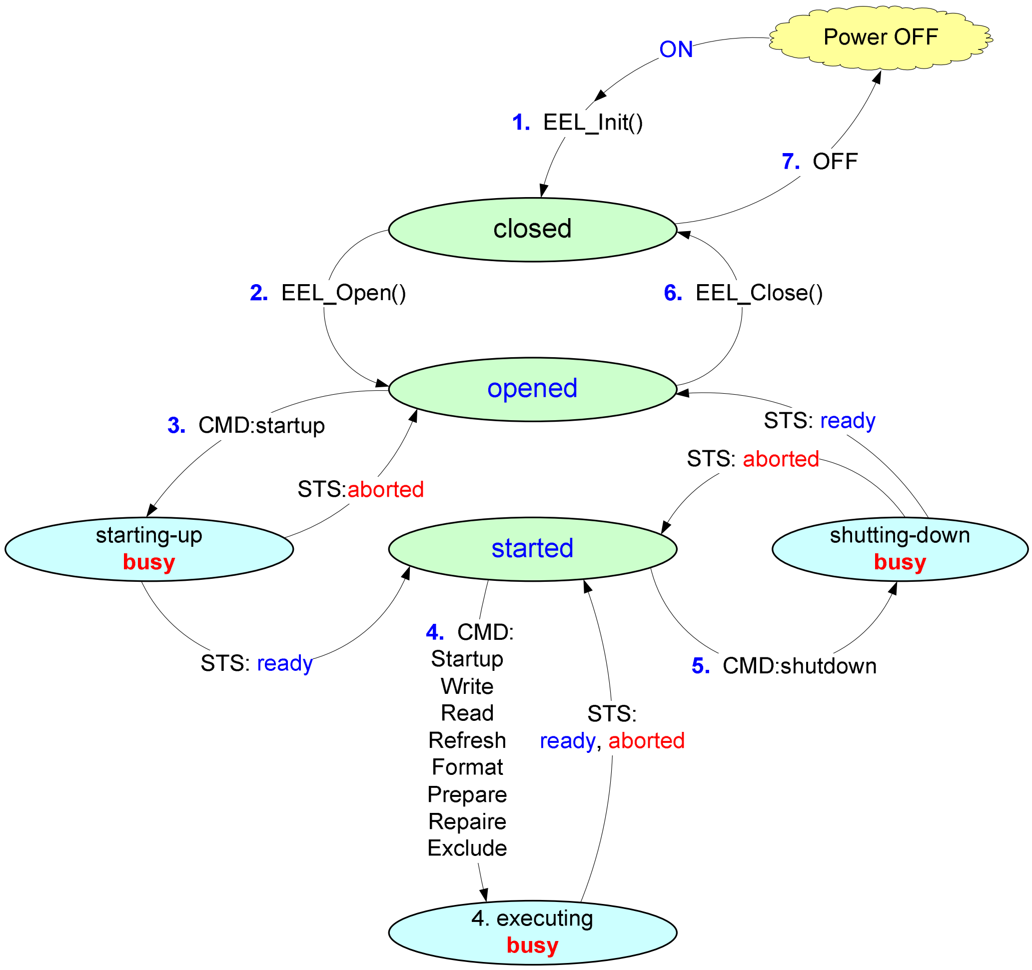

Figure 6-1 EEL driver activation and deactivation sequence

6.2.2 Real-time capabilities

The EEL driver is using commands of the self-programming library their execution time is not always constant. It depends on the device oscillator configuration (internal/external) as well on the system voltage (normal/low voltage). The execution time of some self-programming commands is not always deterministic. Especially the real execution time of the Erase- and Write-command can enlarge during the device live time. Some systems cannot accept this fact due to its critical timing requirements (i.e. communication protocols). Also watchdog based systems has to re-trigger it at certain, precise pre-defined time point. To fulfill such strict timing requirements during EEPROM emulation running in the background the complete system has to be configured very carefully. Execution time of selfprogramming commands is specified in the device user's manual or the corresponding FSL (Flash Selfprogramming) user's manual.

The execution time of the self-programming commands isn't the only parameter influencing the real-time behavior of the EEPROM emulation. Each time a running FSL command is interrupted by a non-masked interrupt source the corresponding interrupt service routine is invoked after certain non-deterministic delay. This interrupt latency depends several factors like: the interrupted command, the used oscillator the internal status of the interrupted firmware. For device specific interrupt latency please have a look to the user's manual of the used device or the corresponding FSL (Flash Selfprogramming) user's manual.

The interrupt latency in firmware operation cannot be influenced by the application, but the problem of non-deterministic command execution time can be solved by using any periodical non-masked interrupt source for interruption of the command execution. We call such an interrupt “chopper” because it’s interrupting running self-programming commands each predetermined period. Of course, jitter caused by the interrupt latency mentioned before has to be taken into account.

6.2.3 Interrupts in enforced operation mode

When EEL commands are executed in “enforced” mode (EEL_enforce(..) function), the application gives up the control of the CPU until the entire command is finished. However during the whole command execution time all non-masked interrupts are serviced, but the application is “blocked” for this period. The worst-case interrupt latency specified for the used device has to be taken into account. From application point of view the EEL command execution is like a simple function call.

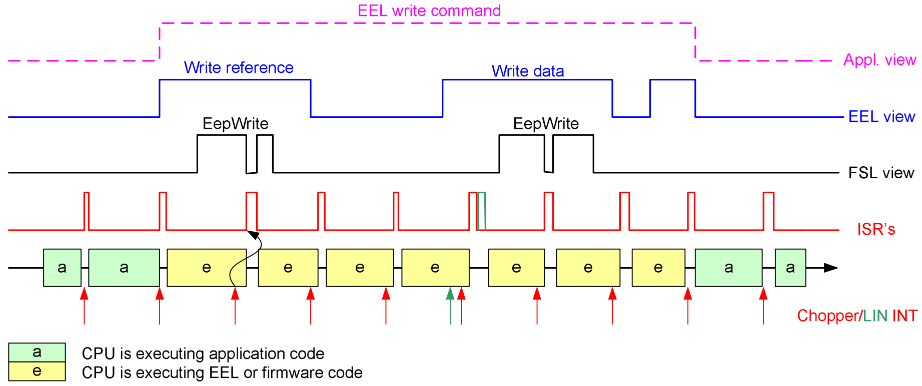

Figure 6-2 Timing example of interrupted "enforced" operation mode

6.2.4 Interrupts in background operation mode

When EEL commands are executed in “background” mode(EEL_execute(..) and EEL_Handler()), the application regains the CPU control for any desired time after a non-masked interrupt was serviced. The application gets the chance to react to the interrupt event, but also watchdog or other timed actions can be processed. The resumption of the not finished command is done by the EEL_Handler() who is re-calling the EEL driver recurrently. The worst-case interrupt latency specified for the used device has to be taken into account.

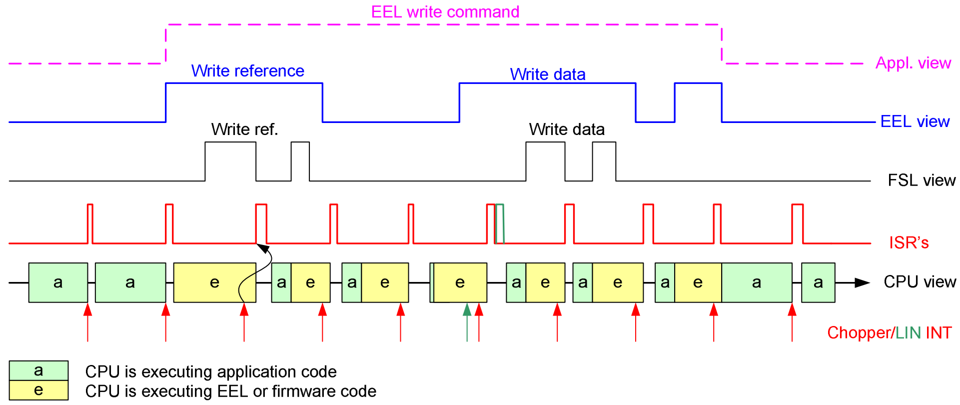

Figure 6-3 Timing example of interrupted "background" operation mode

Chapter 7. Error handling

During operation time several problems can be signalized by the EEL driver. Some of them are indicating normal internal status that can be used by the application to maintain the operation of the EEL driver. Other signalizes more or less serious problems and requires reparation sequences to be removed. All the possible error codes are described in the command related chapters. The developer can use this description to implement a standard, central error handler for his application. The EEL driver does not correct/repair such problems automatically to avoid any unnecessary negative impact of the real-time characteristics. The application retains the flexibility to decide about CPU allocation used for the reparation process in dependency on the current situation.

Appendix A Revision History

Table A-1 Major Revisions in this Edition

Chapter	Page	Describtion
4.8	68	Seqence of the exclude flow has changed. “Figure 4.15” added
4.8	69	“usage” of the exclude command
4.8	69	Exclude error handling has changed
4.2	52	EEL_ERR_PROTECTION error added in table 4-3
4.4	58	EEL_ERR_PROTECTION error added in table 4-5
4.6	64	EEL_ERR_PROTECTION error added in table 4-7
4.6	64	EEL_ERR_PROTECTED_BLOCK error removed in table 4-7
4.7	67	EEL_ERR_ PROTECTED_BLOCK error added in table 4-8
4.8	70	EEL_ERR_ PROTECTED_BLOCK error added in table 4-9
4.7	67	EEL_ERR_NO_ACTIVE_BLOCK error added in table 4-8