FIELDBUS FOUNDATION Field Device Software
PROFIBUS PA Device Software

EEPROM-Interface User Manual

Version 2.41

Date: November, 2011

SOFTING Industrial Automation GmbH
Richard-Reitzner-Allee 6
D-85540 Haar
Phone (++49) 89 45656-0
Fax (++49) 89 45656-399

Table of Contents

Terms and Abbreviations

EEP	EEPROM Handling Component
EEPROM	Electrically Erasable Read Only Memory
FBLK	Function Block
FD	Field Device
HW	Hardware
I²C	Inter-IC Bus (IC = Integrated Circuit)
ID	Identifier
IF	Interface
MCU	Micro Controlling Unit
NV	Non-Volatile
NP	Non-Volatile Parameter
NE	Non-Volatile Extra Data
OS	Operating System
OSIF	Operating System Interface
RAM	Random Access Memory

References

/1/ OSIF User Manual V2.41; Softing, November 2011 (OSIF User Manual.pdf)

1 Introduction

The EEPROM device is used to provide non-volatile storage of important data needed by several software components. Access of these components to those EEPROM data has to be coordinated.

The difficulties in accessing EEPROM devices result from the common-used serial interface. This interface introduces wide differences to the access to other peripheral components, which are in general interfaced “memory mapped”.

Additionally there are some more tradeoffs to be regarded when writing to EEPROMs, raising additional problems for their access handling.

To simplify and coordinate access to an EEPROM device’s data, the Softing Field Device (FD) Protocol Software provides a central and common interface. Certain software components need not have to deal with the data structures inside the EEPROM as well as the special handling of the write access.

2 Structure

The EEP interface provides functions that allow access to the EEPROM data for the several components of the FD protocol software (and to application software components). Inside the EEP interface the functions provided by the HW component (as defined in `EEPROM.H`) are used to physically access the EEPROM device.

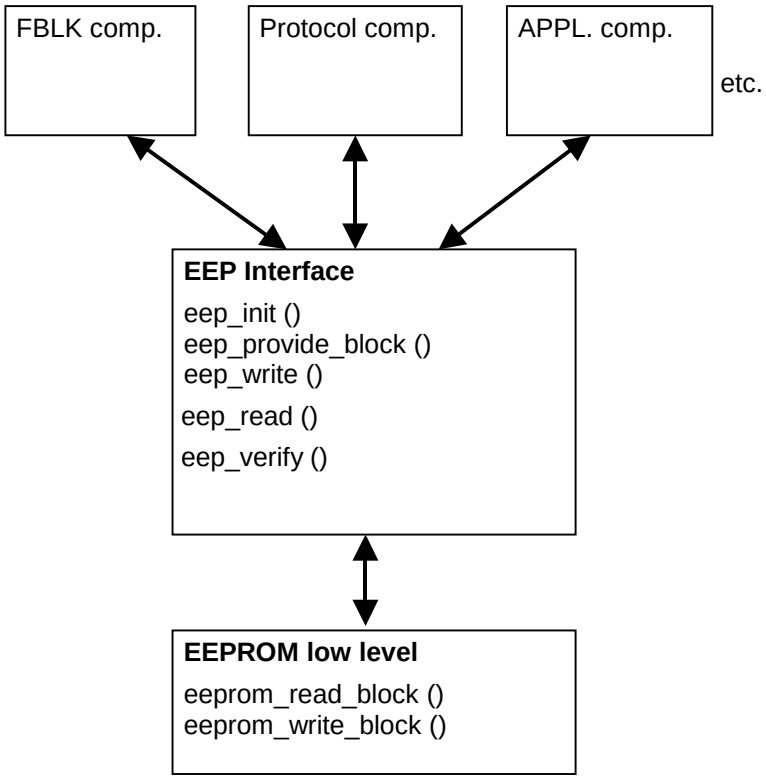

Fig. 2-1: EEP Interface Structure

The EEP interface deals with block management, block initialization etc, while the EEPROM low level access routines are responsible for interfacing the EEPROM device by the appropriate (serial or whatever) protocol (e.g. I²C). Thus, low level interface in combination with the HW component has to initialize the used peripherals (serial interface, port pins, timer etc.).

It is recommended that all access to the EEPROM is done via the EEP-IF in order to prevent the EEPROM from concurrent operations. Additionally, the block mechanisms provide an easy way to keep all EEPROM data, used by several components, as independent as possible.

3 Theory of Operation

3.1 Segmentation into Data Blocks

Non-volatile data of several components are filed in separate blocks. This prevents the data from being accidentally overwritten by other components. The certain component just need to know the size of its block and the structure the data are organized into within its block, but not the block's position and organization within the EEPROM.

The blocks are identified by ID numbers, which are globally defined. The component selects the block to be accessed by passing that block ID to the interface functions. The sequence of block IDs may have gaps, but the highest block ID must not exceed the maximum number of blocks (Refer to the define *EEP_NBR_OF_BLOCKS* in file eep_if.h).

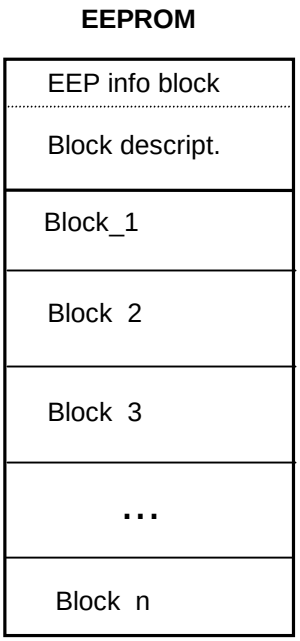

Fig. 3-2: EEP Block Structure

The maximum number of the blocks is fixed at compile time. The actual number of blocks as well as their size is configured during run time and stored at the beginning of the EEPROM.

Access is allowed only within block limits. The interface functions prohibit any access over these block limits.

Each block contains a checksum field, which allows checking the data consistency within the block.

The last two blocks (four blocks, when NV_DATA_DOUBLE_BUFFER is set) are used to store NP data and NE data.

3.2 RAM Image

In general block data are not accessed directly inside the EEPROM, but using a kind of data cache in RAM, namely the RAM image. Thus, the internal timing of the EEPROM device is insulated from the interface routines. The reading performance is greatly improved due to the large decrease in access time On the other hand it also results in an additional delay when writing. The data is physically written within the EEPROM device a certain time interval after the interface function may be returned.

For efficiency reasons concerning memory usage of non-volatile (NV) data buffering, the EEP user components work directly on the RAM image. The EEP-IF provides the start address of each block within the RAM image to its owner and, on request of the owner, just focuses on keeping the data consistent between RAM image and EEPROM device.

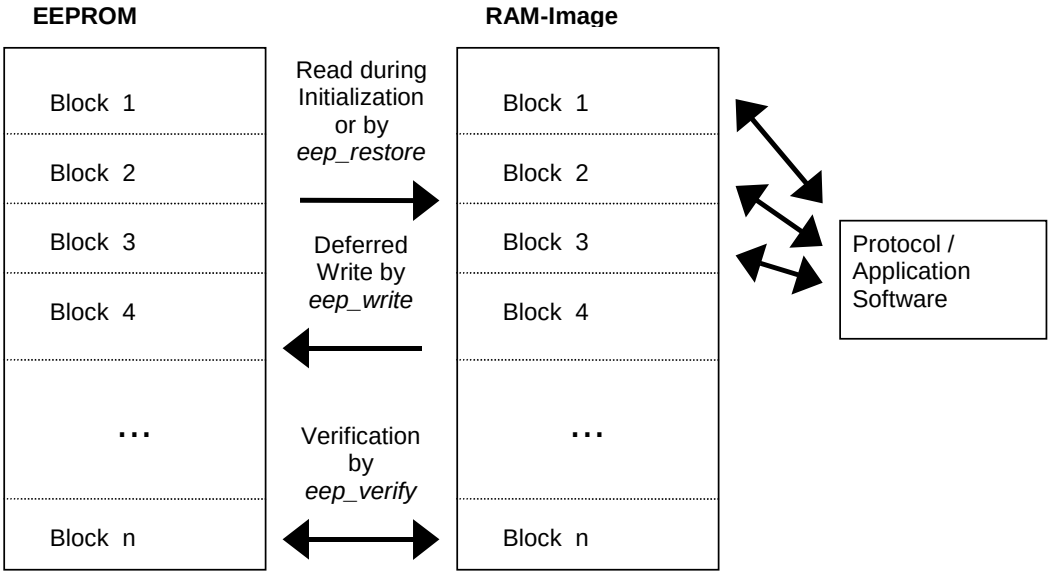

Fig. 3-3: EEP RAM Image Organization

To overcome the problem, that the interface function returns before the access is finished successfully, the write function may be called with parameter "*mode*" set to "EEP_WAIT". In this case the function does not return until the write access is terminated and passes the result of writing as the return value.

Note: The NE block is excluded from the cyclically verification of the blocks in the EEPROM.

3.3 Accessing EEP data

In general, the EEP operation divides into two, optional three phases: device initialization, block configuration, if needed, and device operation. In the initialization phase, all interfaces are established and data is copied from the EEPROM device to the RAM image, if possible. The block users have to request the start address of their blocks and to check the data for correctness / consistency. If the block content does not agree with what was expected, the block configuration phase has to be entered, generating new block descriptions and finally activating them. Now, the new block start addresses have to be requested in the same procedure as during initialization. Partial reconfiguration is not possible; if anyone requests a new or changed block, the previous block descriptions are destroyed. The blocks may be filled with default values, then the components can proceed to the 'normal' device operation phase, where block contents may be read or written.

Since all EEP-IF user components work directly on the RAM image, there is no need for an explicit 'read' function. All blocks are filled during startup with the appropriate data, which may be accessed directly further on. In case someone accidentally alters the RAM image and wants to skip all changes, EEP-IF provides an 'undo' function named "*eep_restore ()*", which may overwrite RAM image data with the according EEPROM contents.

Writing data to the EEPROM is a two step process: First the according positions in the RAM image are altered, then the new block data has to be written to the EEPROM device. To prevent any possible background mechanisms checking the RAM image for consistency from generating an alarm, before altering any block data in the RAM image the function "*eep_start_write ()*" has to be called. Now the RAM image may be written as needed. Finally, by calling "*eep_write ()*", the block checksum is updated and the data is transferred to the EEPROM. Simultaneously, the block's state is reset to non-writing, thus re-enabling the background check mechanisms.

There is a special feature for block data, that has to be updated more frequently, comparable to the block checksum (e.g. for the function blocks the 'STATIC_REVISION' parameter): In the block configuration data an "auto-write" area for such data can be reserved at the start of the block, where the data is always transferred to EEPROM when completing a write access by "*eep_write ()*". This area is written automatically, when the block checksum is updated; there is no need for an appropriate call of "*eep_write ()*".

3.4 EEP-IF in a Multitasking Environment

EEP-IF provides multitasking support using OSIF (ref. to /1/) mechanisms for inter-task communication. As a limitation, the multitasking support works only in the operation phase, i.e. tasks may write or restore block data concurrently. Block configuration must not be run concurrently, i.e. configuration requests may be issued by several tasks, but one after another. Finally, the activation of the block descriptions must be done by exactly one task. If anyone requests another block configuration after activation, the newly finalised configuration will be destroyed, whereby all block IDs with their according data pointers are invalidated.

The functions for writing or restoring block data are reentrant and designed for use in a multitasking environment. Access to the EEPROM device is outsourced to a task of its own, which is interfaced by a service interface using OSIF. Thus, handling rsp. serialization of concurrent EEPROM access is provided automatically. Nevertheless, EEP-IF routines must not be accessed from interrupt service routines, since they may cause the calling task to wait for completion of the access, which does not make sense (and therefor is not allowed) for interrupt service routines.

The multitasking issues are widely hidden by the interface routines. If desired, the task switch to the EEP task will be invisible to the calling task; the interface routine will not continue until the EEP task has confirmed the EEPROM access.

In conformance to OSIF, all tasks interacting with the EEP task (i.e. using EEP-IF functions) must be enabled to handle an OSIF_EVENT_EEPROM. This item has to be added to task's properties in file `OSIF_CFG.C`.

3.5 System Startup

By calling "*eep_init ()*" the EEP interface initializes its internal data structures and starts the HW interface by calling "*eeprom_init ()*" Next, some management information as well as the block descriptions are read and finally the complete EEPROM data is transferred to the RAM image by use of "*eeprom_read_block ()*". If anything fails, the read process is cancelled and the according data structures are initialized to zero, i.e. no blocks available.

The EEP user components now may request the starting addresses of "their" data blocks by calling "*eep_provide_block ()*". If (as seen from the EEP-IF point of view) the data block is ok, i.e. existent with a correct checksum, a pointer to the block within the RAM image is generated; otherwise the function returns an error code. The EEP user components are responsible for checking the data for consistency of structure and contents.

If the block structure does not agree with the EEP user's expectation, the structure information must be rebuilt and the data blocks filled with default information. New block descriptions are passed to EEP-IF by the function "*eep_config_change ()*". When all block descriptions are updated, the new configuration has to be activated by calling "*eep_config_act ()*". Any previous configuration will be deleted. Now the pointers to the data blocks can be requested, again by calling "*eep_provide_block ()*". Since the blocks do not contain any data when newly created, now some default data should be written to the blocks using the standard write mechanism.

3.6 EEP Information

The EEP information structure is by default at the beginning of the EEPROM and contains information about the EEPROM. The structure is as follows:

```
typedef struct {
    USIGN16      size_eep_data;  /* size of data area               */
    USIGN8       num_blocks;     /* number of existing blocks       */
    USIGN8       page_size;      /* size of EEPROM pages            */
    T_EEP_STATE  state;          /* current EEPROM state            */
    T_EEP_ADDR   start_data;     /* start address of data area      */
    T_CHKSUM     crc;            /* checksum over block descriptions */
} T_EEP_INFO;
```

The *"size_eep_data"* is the used size of the EEPROM without EEP information structure and the block configuration structure. The total size used of the EEPROM is calculated by the following term:

```
Total EEPROM size used =   size_eep_data
                         + sizeof(T_EEP_INFO)
                         + num_blocks * sizeof(T_EEP_BLOCK_DSC;
```

The *"num_blocks"* component contains the number of logical EEPROM blocks used by the device. "page_size" is the page size of the EEPROM. The *"state"* component is used to set the state of the EEPROM during startup phase, configuration phase and normal operation. The state can be:

```
typedef enum {
    EEP_VALID               = 0,
    EEP_UNAVAIL,
    EEP_IN_CONFIG
} T_EEP_STATE;
```

EEP_VALID means that the EEPROM is accessible and the block configuration is valid. *EEP_UNAVAIL* means, that the EEPROM is not accessible. *EEP_IN_CONFIG* means, that the EEPROM is in a configuration phase, because the configuration has changed.

"start_data" is the offset, relative from the beginning of the EEPROM, where the first block of the EEPROM starts. *"crc"* is the check sum of the block configuration in the EEPROM.

3.7 Block Configuration

Function "*eep_read_info ()*", called by "*eep_init ()*", gains all information about block configuration from a table located at the begin of the EEPROM. To enter these block descriptions if needed, EEP-IF provides the function "*eep_config_change ()*", which receives block configuration entries described by the following structure:

```
typedef struct
```

```
{
    USIGN8       block_id;             /* block id                  */
    USIGN8       auto_write_length;    /* size if auto-store data   */
    USIGN16      block_size;           /* size of block             */
}
T_EEP_BLOCK_DSC;
```

"*block_id*" contains the ID of the block to be configured; "*auto_write_length*" describes the size of the area (starting at offset 0 of the block), which will be automatically written to the EEPROM each time when the checksum is updated. Finally, "*block_size*" informs about the overall block size (in units of bytes), including the automatically updated data.

Since more than one of these descriptors may be passed to "*eep_config_change ()*", the number of descriptors must be passed also. On the other side, "*eep_config_change ()*" may be called repeatedly (i.e. more than once) before activating the configuration. If different configurations for one and the same block are given, the last one will be used.

3.8 NV Data Verification

As already mentioned, the use of the RAM image separates the write procedure of NV data and the actual transfer and storage of data physically to the EEPROM device. This handling unfortunately complicates a mechanism to directly verify the write process. This is done by a separate background task which continuously, as far as there is CPU power available, scans the EEPROM for a discrepancy to the RAM image. The background task is executed every 100ms.

EEP-IF provides a function to be used in such a background process. As most other EEP-IF functions, it is block oriented, i.e. verifies in units of data blocks. In order to disturb any higher priority write access as less as possible, data access within a block is segmented into 32 byte slices, resulting in a max. write delay of < 4 msec at 100 kHz serial data transmission clock.

The programmer of the application can decide to enable/disable the NV data verification. Please refer to module appl_if.c and function appl_init(). The NV data verification is enabled by default.

4 Inserting New EEP Blocks

To use this concept for his own purpose, the application programmer may define his own blocks in EEPROM. Regarding some requirements this may be done without influence to any other component using EEP-IF.

4.1 Necessary Extensions in File “eep_if.h”

This file contains the general data structures for accessing the EEP interface as well as the definitions for the block IDs.

The application programmer may add new IDs with respect to the following limitations:

- Each ID must be unique
- The existing IDs must not be changed
- The numerical value of the ID must not exceed the value of “EEP_NBR_OF_BLOCKS”

4.2 Handling in EEP User Components

During the device configuration phase, the according block description has to be entered by calling “*eep_config_change ()*”, then, after activation of the current configuration, the start address of the appropriate RAM image area has to be gathered by use of “*eep_provide_block ()*”.

During subsequent power-up procedures, in the device start phase the block user has to gather the start address of the appropriate RAM image area by use of “*eep_provide_block ()*”, which will now normally contain the NV data. The user has to check the block for consistency and, if ok, may proceed to the device operation phase, or, if not ok, enter the configuration phase once again.

5 Interface Functions

The prototypes of the interface functions are defined in the file “EEP_IF.H”. In general, the EEP-IF functions used during the initialization and configuration phase are not reentrant, i.e. they must not be called inside interrupt service routines. If a (preemptive) multitasking OS is used, it must be sure not to call these EEP-IF routines while another EEPROM procedure (in another task) is in progress.

As already mentioned, only those functions for writing or restoring block data are reentrant and designed for use in a multitasking environment. Nevertheless, calls from interrupt service routines are not allowed.

5.1 Common Data Structures and Definitions

The type definition T_EEP_RESULT contains the result codes as used by the EEP-IF functions:

```
typedef enum _T_EEP_RESULT
{
    EEP_OK              = 0x00, /* function completed successf.   */
    EEP_NOT_AVAIL       = 0x01, /* EEPROM device is not available */
    EEP_NOT_EXIST       = 0x02, /* requested ID not existent      */
    EEP_MEM_INSUFF      = 0x03, /* not enough memory to proceed   */
    EEP_PARAM_ERR       = 0x04, /* param. invalid or out of range */
    EEP_READ_ERR        = 0x05, /* read operation not successf.   */
    EEP_PRG_ERR         = 0x06, /* program operation not sucessf. */
    EEP_CS_ERR          = 0x07, /* block checksum inconsistent    */
    EEP_INCONSISTENT    = 0x08, /* block configuration inconsis.  */
    EEP_INVALID_SERVICE = 0x20, /* inv. Request to EEP task       */
    EEP_IN_CHANGE       = 0xF0  /* EEP config. not fixed yet      */
} T_EEP_RESULT;
```

These codes must not be changed without recompilation of the protocol components, but may be extended, if needed.

5.2 Initialization

5.2.1 eep_init ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_init(VOID);

RETURN_VALUE
    EEP_OK           => completed successfully
    EEP_READ_ERR     => EEP info block could not be read
    EEP_CS_ERR       => checksum invalid in min. one block
    EEP_INCONSISTENT => block descriptions are inconsistent
```

This function has to be called once during system initialization. Before calling “*eep_init ()*” no other EEP_IF routine must be used. It prepares the internal EEP management structures and reads EEPROM data to the RAM image.

If initialization runs without problems, the function returns EEP_OK. If reading the EEP information block already fails, as a result EEP_READ_ERR is returned. EEP_CS_ERR indicates that the checksum in at least one of the blocks (the EEP info block or the data blocks) is invalid. "*eep_init ()*" also checks the consistency of the data size as stored in the EEP info block against the real amount of data as computed from the block descriptions and returns EEP_INCONSISTENT, if not matching.

5.2.2 eep_provide_block ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_provide_block
  (
    IN  USIGN8      block_id,    /* ID of block                 */
    IN  USIGN16     block_len,   /* size of block               */
    OUT VOID * *    p_ram_image  /* pointer to data image       */
  );

RETURN_VALUE
    EEP_OK           => completed successfully
    EEP_NOT_EXIST    => block with ID 'block_id' does not exist
    EEP_INCONSISTENT => block size does not agree with 'block_len'
    EEP_CS_ERR       => block checksum invalid -> data maybe corrupt
    EEP_READ_ERR     => EEPROM could not be read during initialization
```

To get the start address of their data area within the RAM image, a block's owner has to call function "*eep_provide_block ()*". The ID of the desired block is passed via parameter *block_id*, the expected size of the block via parameter *block_len*. When completed successfully (indicated by the return value EEP_OK), parameter **p_ram_image* will point to the block data start address as requested and the block owner may work on the data.

If a block with ID *block_id* does not exist at all, the function returns EEP_NOT_EXIST. If the size of the block as found in the block's description does match with *block_len*, as result the value EEP_INCONSISTENT is created. EEP_CS_ERR as a result indicates probable corrupt contents inside the block, detected by an incorrect checksum. EEP_READ_ERR as result indicates that it was not possible to read data from the EEPROM.

5.3 Configuration Phase

Block descriptions may be entered to the system during the configuration phase. If all needed descriptions are loaded, which is done by one or several calls to "*eep_config_change ()*", the configuration must be activated, i.e. checked for consistency and transferred to EEPROM. Any subsequent "*eep_config_change ()*" request will destroy the previously activated configuration, so that the complete configuration process must be repeated. The configuration phase finishes by gathering all required block data addresses by subsequent calls to function "*eep_provide_block ()*", once for each block ID to be used.

During configuration, other EEP-IF calls are not allowed. However, all other routines will return the value EEP_IN_CHANGE.

5.3.1 eep_config_change ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_config_change
  (
    IN USIGN8               no_of_eep_block_dsc,
    IN T_EEP_BLOCK_DSC *    p_eep_block_dsc
  );
RETURN_VALUE
```

```
EEP_OK          => completed successfully
EEP_PARAM_ERR   => block ID(s) exceed limit
```

To pass block description entries to the EEP-IF, the function "*eep_config_change ()*" is used. With one call one or more descriptors are entered. The parameter *no_of_eep_block_dsc* contains the number of descriptors passed to the function, parameter *p_eep_block_dsc* points to the descriptor array with (at least) *no_of_eep_block_dsc* elements of type *T_EEP_BLOCK_DSC* (ref. to "3.7 Block Configuration"). If the ID field in a descriptor entry exceeds the EEP_NBR_OF_BLOCKS limit, the function returns EEP_PARAM_ERR, otherwise EEP_OK.

"*eep_config_change ()*" may also be called more than once to enter all block descriptors as needed. If there is more than one descriptor for the same block, the last one will be used. This is also true, if these descriptors are contained in one descriptor array: the entry with the higher array index will be used.

A call to "*eep_config_change ()*" will invalidate any existing configuration, even if activated just before. The EEPROM cannot be used, until the configuration is completed.

5.3.2 eep_config_act ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_config_act (VOID);
```

```
RETURN_VALUE
    EEP_OK          => activation succeeded
    EEP_MEM_INSUFF  => data does not fit into EEPROM
    EEP_PRG_ERR     => error while programming EEPROM
```

After loading the block descriptions, the internal management structures have to be rebuilt and the current block configuration must be saved to the EEPROM. This is done by a call to function "*eep_config_act ()*". If the activation completed successfully, EEP_OK is returned and the new start addresses may be fetched.

If the requested NV storage space exceeds the amount of available data space in the EEPROM device, the error code EEP_MEM_INSUFF is generated. If activation failed, a new block configuration with lower requirements to the NV memory amount should be loaded. The result EEP_PRG_ERR indicates, that for any reason the block configuration could not be saved to the EEPROM.

5.4 Operation Phase

The main EEPROM handling in the operation phase confines to writing data to the EEPROM in order to achieve non-volatile storing. Additionally, to provide something comparable like an "undo" function, it is possible to read back the EEPROM contents to the RAM-Image.

As mentioned in "3.4 EEP-IF in a Multitasking Environment", these functions work even if called concurrently from a different task context. As a trade-off, they may cause the calling task to wait by mechanisms of OSIF until the EEPROM access has completed.

Since any change of data operates directly on the RAM image, the write process divides into three substates. First, it has to be announced, that the RAM image content is to be altered and therefore the block checksum will run invalid, which is done by a call to function "*eep_start_write ()*". This will prevent any processes checking the blocks for consistency in the background from generating an alarm due to the now invalid checksum. Next, the block data may be changed by the block's owner by directly accessing the RAM image – no interface function is needed here. Finally, by calling "eep_write ()", the altered data is written physically to the EEPROM, the checksum is updated and also written to the EEPROM, together with the data in the auto-write area. From now on, the background check process may continue to do its job.

5.4.1 eep_start_write ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_start_write
(
    IN  USIGN8      block_id     /* ID of block                  */
);

RETURN_VALUE
    EEP_OK           => no error occurred
    EEP_PARAM_ERR    => wrong parameter
    EEP_IN_CHANGE    => EEP configuration not consistent temporarily
```

"*eep_start_write ()*" informs the system about the subsequent alteration of data in the block, which is indicated by the parameter *block_id*. After receiving the value EEP_OK, the block owner may perform the write access to the RAM image data.

If no block with ID *block_id* exists, "*eep_start_write ()*" returns the error code EEP_PARAM_ERR. If there is no valid EEP block configuration present at the moment, EEP_IN_CHANGE is generated. This error code will disappear as soon as a valid block configuration is activated.

5.4.2 eep_write ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_write
(
    IN  USIGN8      block_id,    /* ID of block                  */
    IN  USIGN16     offset,      /* data offset inside the block  */
    IN  USIGN16     length,      /* length of data in bytes       */
    IN  USIGN8      mode         /* selected write mode           */
);

RETURN_VALUE
    EEP_OK           => no error occured
    EEP_PARAM_ERR    => wrong parameter
    EEP_PRG_ERR      => error while programming EEPROM
    EEP_IN_CHANGE    => EEP configuration not consistent temporarily
```

The function "*eep_write ()*" writes block data from the RAM image to the EEPROM device. The data block is addressed by parameter *block_id*; parameter *offset* passes the data location inside the block while parameter *length* describes the number of bytes to be written. If *offset* and *length* are set to 0, the complete block is written.

Parameter *mode* controls the behavior of the routine:

EEP_WAIT — the function does not return before the final write cycle containing the block checksum (and the auto-write area, if configured) is completed in the EEPROM device. Only EEP_WAIT is supported.

If writing to the EEPROM was successful EEP_OK is returned. If the addressed block does not exist or if *offset* or *length* do not agree with the block description, "*write_to_eeprom ()*" returns EEP_PARAM_ERR. The error code EEP_PRG_ERR is returned, if writing to the EEPROM device could not be finished successfully (may occur only, if EEP_WAIT was selected by the *mode* parameter). Again, if there is no valid EEP block configuration present at the moment, EEP_IN_CHANGE is generated. This error code will disappear as soon as a valid block configuration is activated.

5.4.3 eep_restore ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_restore
(
    IN  USIGN8      block_id,    /* ID of block                   */
    IN  USIGN16     offset,      /* data offset inside the block  */
    IN  USIGN16     length,      /* length of data in bytes       */
    IN  USIGN8      mode         /* selected write mode           */
);

RETURN_VALUE
    EEP_OK          => no error occurred
    EEP_PARAM_ERR   => wrong parameter
    EEP_READ_ERR    => error while reading from EEPROM
    EEP_IN_CHANGE   => EEP configuration not consistent temporarily
```

"*eep_restore ()*" re-reads the block data from EEPROM to the RAM image. The data block is addressed by parameter *block_id*; parameter *offset* passes the data location inside the block while parameter *length* describes the number of bytes. If *offset* and *length* are set to 0, the complete block is read.

Analogously to "*eep_write ()*", parameter *mode* controls the timing behavior of the routine:

EEP_WAIT — the function does not return before all data is read from the EEPROM device. Only EEP_WAIT is supported!

If all data could be transferred to the RAM image, EEP_OK is returned. EEP_PARAM_ERR as the result indicates, that the addressed block does not exist or if *offset* or *length* do not agree with the block description. If the EEPROM could not be accessed, EEP_READ_ERR is generated. If there is no valid EEP block configuration present at the moment, EEP_IN_CHANGE is generated.

5.4.4 eep_verify ()

```
FUNCTION GLOBAL T_EEP_RESULT eep_verify
(
    IN  USIGN8      block_id     /* ID of block                  */
);

RETURN_VALUE
    EEP_OK          => no error occurred
    EEP_PARAM_ERR   => wrong parameter
    EEP_READ_ERR    => error while reading from EEPROM
    EEP_IN_CHANGE   => EEP data verification temporarily not possible
    EEP_CS_ERR      => checksum of block invalid
    EEP_INCONSISTENT=> EEP data do not agree with RAM image data
```

"*eep_verify ()*" performs verification of an EEPROM data block, identified by *block_id*, against the according RAM image location. To ensure the integrity of the RAM image, the block checksum is verified first. If the RAM image is altered concurrently, the function returns 'EEP_IN_CHANGE'.

To allow continuos checking of the EEPROM data, maybe in background in a separate task, "*eep_verify ()*" divides all accesses into slices of max. 32 bytes. Any write access by higher priority tasks thus won't be delayed for more than approx. 4 msec.

If verification of the block completed successfully, the result is set to EEP_OK. If the addressed block does not exist, EEP_PARAM_ERR is returned. EEP_IN_CHANGE indicates, that the EEPROM configuration at all is not valid currently (due to a concurring call to "*eep_config_change ()*"), or the RAM image data for the block has been altered in the meantime (due to concurring calls to "*eep_start_write ()*" rsp. "*eep_write ()*"). Anyway, this situation should disappear during subsequent calls to "*eep_verify ()*".

If the RAM image of the block is corrupt, EEP_CS_ERR is returned. If the EEPROM device physically rejects any read access, the result is EEP_READ_ERR. Finally, if the verification of the EEPROM content against the according RAM image data fails, the function returns EEP_INCONSISTENT.

The verification of the blocks is enabled by default (Refer to module *appl_if.c* and function *appl_init*). The verification of the block with NE data is excluded from the verification, because the check sum of this block is calculated only when the block is written into the EEPROM, but the data in the RAM image may change more often.

6 Meaning and Handling of NP and NE Block in EEPROM

The meaning of NP and NE data is a bit special and needs some more information about handling and usage. By default, both block types, NP and NE have two blocks (double buffered) in EEPROM. One block is used as backup, when the other block is corrupted, because of power fail during EEPROM write access.

- The NP data block is used to store function block or transducer block parameters with the N attribute. This attribute means, that these parameters must be stored in non-volatile memory.
- The NE data block is used to store special data in non-volatile memory. Currently, this feature is used for the Integrator function block for Fieldbus Foundation and the Totalizer function block for PROFIBUS PA. The allocation of the NE data storage is described later.

6.1 Enabling the double buffer for NP and NE

This feature can be enabled or disabled by an entry in the file Makeconfig.mak for the specific target. Note, that the file Makeeconfig.mak exists for all targets!

For example in `target\mak\ff_r_release\Makeconfig.mak`:

Enabled by:

```
export NV_DATA_STORAGE = NV_DATA_DOUBLE_BUFFER
```

Disabled by:

```
export NV_DATA_STORAGE =
```

Note, this double buffer feature can only be enabled or disabled for NP and NE together! It must be disabled, if the EEPROM driver is supporting double data storage in EEPROM by itself!

6.2 Adjust the cycle time for NP and NE block in EEPROM

Common to NP block and NE block in EEPROM is, that the data for these blocks can be stored cyclically into EEPROM, depending on the particular setting of the cycle time for NP block and NE block.

NV_CYCLE_TIME is used for the NP block in EEPROM. The EXTRA_NV_CYCLE_TIME is used for the for NE block in EEPROM. The definitions are part of the file fdev_cfg.h (FF) and the file pdev_cfg.h (PROFIBUS PA). The unit for this cycle time is 1 second. Think of the life time of an EEPROM.

6.2.1 Adjustment of cycle time for the NP block

A value of 0 for the NV_CYCLE_TIME means, that the cyclically update of the NP block in the EEPROM is disabled, but each write to these parameters via the network will update the data in the EEPROM.

```
#define  NV_RAM_CYCLE_TIME        0ul        /* Update disabled */
```

A value greater 0 for the NV_CYCLE_TIME means, that the cyclically update will be performed every NV_CYCLE_TIME seconds. When the parameter is written via the network, the parameter is updated immediately in EEPROM. Think of the life time of an EEPROM!

```
#define  NV_RAM_CYCLE_TIME         3600ul    /* Update every hour */
```

6.2.2 Adjustment of the cycle time for the NE block

A value of 0 for the EXTRA_NV_CYCLE_TIME means, that the cyclically update of the NE block in the EEPROM is disabled.

```
#define  EXTRA_NV_RAM_CYCLE_TIME   0ul        /* Update disabled */
```

A value greater 0 for the EXTRA_NV_CYCLE_TIME means, that the cyclically update will be performed every EXTRA_NV_CYCLE_TIME seconds. Think of the life time of an EEPROM!

```
#define  EXTRA_NV_RAM_CYCLE_TIME    3600ul    /* Update every hour */
```

6.3 Allocating memory to handle the NE data

The storage for the NE data is allocated by the block, which is using the NE data. Currently NE data is used for the Integrator function block (FF) and the Totalizer function block (PROFIBUS PA). The reason is for example, the total parameter of these blocks. This parameter has the data type float (4 byte), but is calculated and stored with a higher precision, double (8 byte). This higher precision value is stored cyclically in EEPROM, depending on the EXTRA_NV_CYCLE_TIME.

The memory can be allocated with the following function:

```
storage_ptr = fbs_alloc_extra_nv_ram(USIGN8 block_id, USIGN8 size);
```

block_id is a unique block identifier. Valid block_id's are generated by the GenVFD and are part of the file fbif_idx.h for FF and pbif_idx.h for PROFIBUS PA.

Please refer to file appl_if.c for more information about the cycle time for NE data.

7 Low Level Routines

The routines necessary for physically accessing the EEPROM device are contained in module `EEPROM.C`, which is found together with the high level routines within the EEP component. All higher layer procedures use exclusively these routines for accessing the EEPROM device.

7.1 Theory of Operation

In general due to their serial interface EEPROM devices are connected to a corresponding serial interface of the MCU. The actual data transfer from and to the EEPROM may take some time, but is handled interrupt driven in the background.

Following a write operation, EEPROM devices normally need additional time after the data transfer phase to program their internal memory cells. During this time no other access is possible. Additionally, not before expiration of that time it is made sure that the data are securely stored into the EEPROM's memory. For this reason the completion of the (internal) write procedure is checked by a timer controlled polling of the EEPROM device; after completion or timeout the upper layers are informed about the result by using a call-back function (whose address has been passed to the write routine on entry).

7.2 Interface Functions

The interface to the low level EEPROM access routines is defined in file EEPROM.H. The according functions are implemented in file EEPROM.C

7.2.1 Initialization

```
extern unsigned char eeprom_init (void);
```

Initializes the (low level) EEPROM interface including the used MCU peripherals.

7.2.2 Writing Data

```
extern unsigned char eeprom_write_block
(
    T_EEP_ADDR                  eeprom_addr,  /* EEPROM address          */
    const unsigned char FAR_D * src_addr,     /* source address          */
    T_EEPROM_CNT                count,        /* count of bytes          */
    PF_EEPROM_RES               eeprom_res    /* call back function      */
);

RETURN_VALUE
    != 0            => timeout ended
    =  0            => timeout still running or parameter error
```

Function "*eeprom_write_block ()*" transfers a block (the term "block" here is not concurrent with the block concept of EEP-IF, but means just a portion of data of defined length) of size *count* from memory address *src_addr* to location *eeprom_addr* inside the EEPROM device. After completion of writing (or timeout) the callback function "*eeprom_res ()*" is called with a parameter of type "*T_EEPROM_CNT*", where the number of not written bytes is passed. Invocation of the callback function is skipped if parameter *eeprom_res* is set to NULL. "*eeprom_write_block ()*" itself returns as soon as the data transfer to the EEPROM device has completed.

NOTE: A number of not written bytes > 0 indicates an error.

The function concerns on the segmentation of the data block, if the write affords exceeding page limits of the EEPROM. Thus, the size of block to be written is limited only by the size of the EEPROM, not by its page size.

If parameter *src_addr*"is set to NULL, the block specified by parameter *count* and parameter *eeprom_addr* is overwritten with the pattern *Erase_Data*, i.e. erased.

As long as there is another write procedure in progress or the parameters are invalid, "*eeprom_write_block ()*" returns a value of 0, otherwise a result != 0 shows the successful completion of the function.

7.2.3 Reading Data

```
unsigned char eeprom_read_block
(
    unsigned char FAR_D *       dst_addr,     /* destination address     */
    T_EEPROM_ADDR               eeprom_addr,  /* EEPROM address          */
    T_EEPROM_CNT                count,        /* count of bytes          */
    PF_EEPROM_RES               eeprom_res     /* call back function     */
);

RETURN_VALUE
    != 0            => timeout ended
    =  0            => timeout still running or read error
```

Reads a block of size *count* from location *eeprom_addr* within the EEPROM device and copies the data to memory address *dst_addr*.

As long as there is another write procedure in progress or the parameters are invalid, "*eeprom_read_block ()*" returns a value of 0, otherwise a result != 0 shows the successful completion of the function.

After completion of reading (or timeout) the call back function specified by parameter *eeprom_res* is called with a parameter of type *T_EEPROM_CNT*, where the number of not read bytes is passed. The invocation of the call back function is skipped if parameter *eeprom_res* is set to NULL..

NOTE: A number of not read bytes > 0 indicates an error.