XC2000 / XE166 Family

AP1618313

EEPROM Emulation Driver for XC2000 / XE166

Application Note

V1.3, 2012-07

Microcontrollers

Edition 2012-07

Published by
Infineon Technologies AG
81726 Munich, Germany

LEGAL DISCLAIMER

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

AP16183
Revision History: V1.3 2012-07

Previous Version(s): V1.2

Page	Subjects (major changes since last revision)
-	Update source code
25	Change of table title

Trademarks

DAVE™ is a trademark of Infineon Technologies AG.

We Listen to Your Comments
Is there any information in this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

Table of Contents

1 Introduction

Many applications require the frequent storage or update of data on to non-volatile memory during run-time, and this data is usually required to be retained for a period of time after the power supply is removed. EEPROM has traditionally been used for this task, but in recent years Flash memory has become a more cost-effective and faster alternative. New microcontrollers are therefore often offered without any EEPROM memory, but with a large Flash memory.

This application note describes how to emulate EEPROM behavior using Flash memory, with a simple implementation of an EEPROM emulation driver for the Infineon XC2000 / XE166 family of products.

1.1 Definition of Terms

The following terms are used in this document:

Table 1

Term	Explanation
Flash array	Flash array size is device dependent. Please refer to the User Manual of the particular device, under the section heading “Embedded Flash Memory”, for more information.
Flash module number	The flash array number in the microcontroller. Please refer to device User Manual under “Address Mapping” for the numbering description based on the memory address range.
Emulation data set	The size of the emulated EEPROM. A complete emulation data set contains all the EEPROM logical addresses for each EEPROM page.
EEPROM page	An EEPROM page consists of 128 bytes. Each emulation data set contains one or more EEPROM pages.
Data block	One data block contains several EEPROM pages. Together, these EEPROM pages make up an emulation data set. The total emulation size of the EEPROM will contain two or more data blocks.
Total emulation size	The total size of Flash memory used for EEPROM emulation.
Active block	The data block which contains the latest emulation data set.

2 Emulation Driver Features

The EEPROM emulation driver has the following features:

- Each write function programs one EEPROM page to Flash memory
- Each read function fetches one byte or one EEPROM page from Flash memory
- Supports any size of emulation data set in multiples of 128 bytes
- Allows the CPU to process interrupts in real-time for devices with more than 1 Flash array
- User-configurable data block size
- The endurance for 1 sector (4 Kbytes) for a 5 year retention span is 480,000 cycles[1]; the emulated EEPROM endurance is dependent on the total emulation size used, and is obtained from the following equation:

(1)

$$\text{Emulated EEPROM endurance} = \frac{\text{Total emulation size}}{4096\ \text{Bytes}} \times 480\,000$$

1) The data sheet states that the endurance is 15000 cycles per one sector, for 5 years. The algorithm writes 128 bytes each time and erases per sector, so one sector (4 kbytes) can write 32 times before it erased. Therefore the endurance for one sector is 15000 cycles * 32 = 480,000.

3 System Requirements and Limitations

For the EEPROM driver to work correctly, the following system requirements and limitations must be taken into consideration.

System Requirements

- Only one Flash module can be used in this EEPROM emulation driver.
- The emulation data set size must be smaller than the data block size.
- The data block size used for emulation must be in multiples of 4 Kbytes with a minimum of 1 sector (4 Kbytes).
  – This is because every erase is per sector, so each data block size must be sector aligned.
- The total emulation size must be in the multiples of the data block size, with a minimum size of two data blocks.
- A minimum of 1 byte is required in each EEPROM page for EEPROM addressing. The maximum effective data storage is therefore 127 bytes per EEPROM page.
- A minimum of 128 bytes of the user's RAM buffer is required to store the data to be programmed to the emulated EEPROM, and 128 bytes to store the data read from emulated EEPROM.
  – The same RAM buffer can be used for read and write.

Limitations

- When programming or erasing is in progress, it is not possible to write to the emulated EEPROM.
- If an emulated EEPROM write operation is unexpectedly aborted due to power loss, the current data may be lost and may not be recoverable.
- Typically it takes 3.6ms to write one EEPROM page into Flash, but in the worst case it takes up to **n * 3.6ms + 3.6ms** if data copying is in progress (Where **n** refers to the number of EEPROM pages in the emulation data set).
- Background Flash programming and erasing is supported for devices with more than one Flash array. For devices with a single Flash array the program and erase operations will stall the CPU until these operations are completed.
- The EEPROM emulation driver must be initialized before it is used.
- It is recommended to initialize a complete emulation data set by writing to all the EEPROM logical address in the emulated EEPROM to ensure the accuracy of data recovery after a system reset or power failure
- When advance mode is enabled, any user application must ensure that the next data block of the active block is empty or erased.

4 EEPROM Emulation Algorithm

4.1 EEPROM Emulation Data Block

The EEPROM emulation algorithm used by the emulation driver is based on the ‘round-robin’ principle. The Flash memory used for EEPROM emulation can be viewed, to be partitioned into a specific number of data blocks. One of these data blocks contains the latest emulation data set and is called the active block.

The data set is divided into N number of EEPROM pages. Each EEPROM page requires 1 to 2 bytes for the header, used for addressing purposes. Therefore each EEPROM page contains a minimum of 126 bytes of data and a maximum 2 bytes of header.

Note: The size of the header depends on the EEPROM page addressing for the emulation data set, and can be configured by the user. Please refer to MASKING_VALUE in Chapter 5.1, User-configurable Parameters, for more details.

The EEPROM logical address is masked with 0x05 and is located at the end of each EEPROM page as the header. The logical address is masked to ensure the validity of the address.

Note: The user is able to configure the number of bits to be masked. Please refer to VALIDITY_BIT in Chapter 5.1, User-configurable Parameters, for more details.

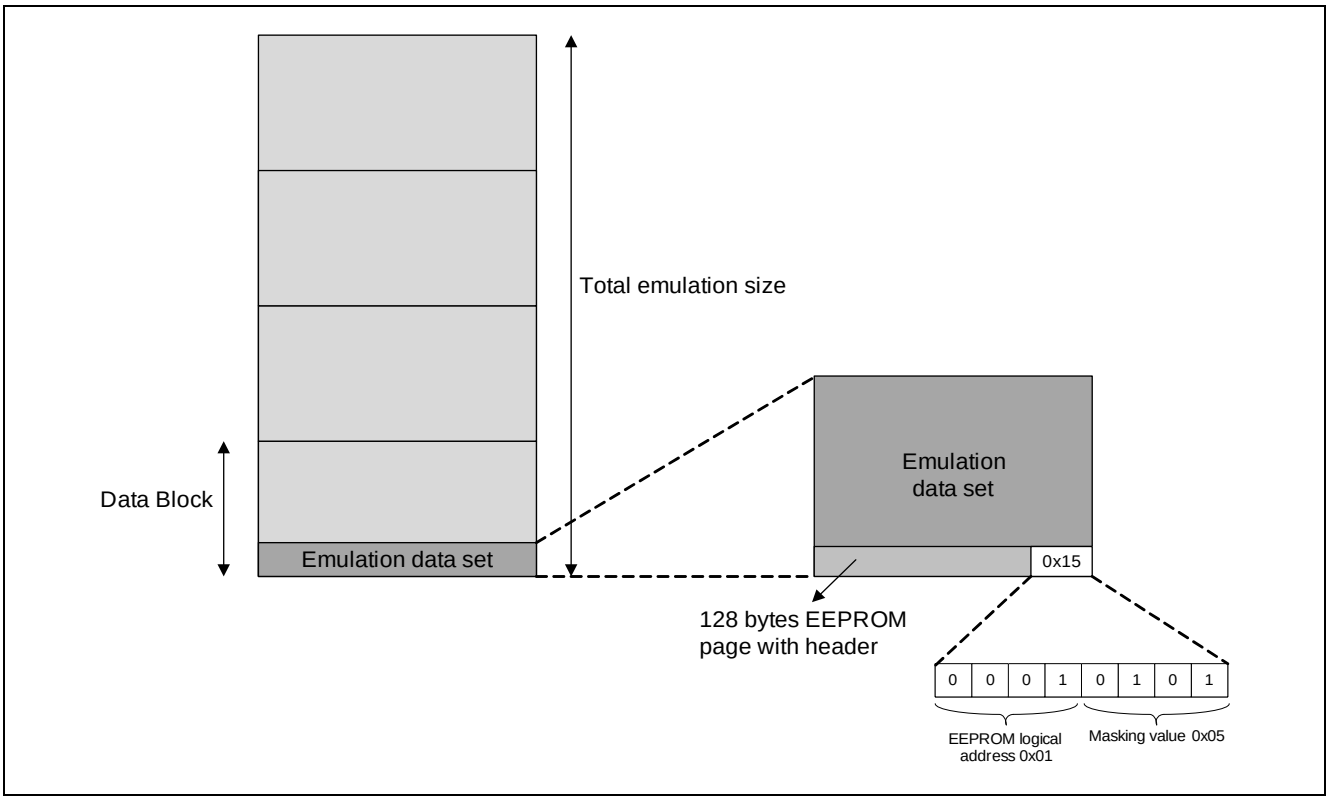

Figure 1 EEPROM Emulation Data Block

4.2 Write and Erase

Each write to the emulated EEPROM is one EEPROM page and requires the user to input the EEPROM logical address when calling the *EEPROM_intWrite* function.

The write to Flash process is performed in the background and so the user should avoid calling the write function again before the current operation is completed. If the write process is called when it is already running, the driver does not start any new write operation but returns EEPROM_BUSY.

There are two possible write sequences, each selected with the user-configurable parameter ADVANCE_MODE:

ADVANCE_MODE Disabled

*Note: This option is recommended for applications that are **not** timing-critical.*

When ADVANCE MODE is disabled, the function updates the write pointer to the next physical write address of the Flash memory and checks if the active block is full, after the write operation is completed.

If the active block is full, the EEPROM emulation algorithm will copy the latest emulation data set to the new data block, which then becomes the new active block.

Finally the function erases the old data block and returns the value 00_H (COMPLETE).

ADVANCE_MODE Enabled

With ADVANCE_MODE enabled there is full application control of the write sequence.

In this option the application is required to call the function EEPROM_vIntProcess after the write operation completes. This can be achieved by coding the function call as part of the Interrupt Service Routine (ISR) when the interrupt is used to indicate the Flash ready status, or by polling the Flash busy bit and calling the function once the Flash is available again.

The EEPROM_vIntProcess function updates the write pointer to the next physical write address of the Flash memory and updates EEPROM_JobType status. If EEPROM_JobType status is not IDLE after calling the function EEPROM_vIntProcess for the first time, it is required to call the function again until EEPROM_JobType indicates IDLE.

When the active block is filled up with data, the EEPROM emulation algorithm will copy the latest emulation data set to the new data block, which then becomes the new active block. An EEPROM write is not then allowed until the whole copy process is complete (This is the ‘worst-case’ time for writing data - see Chapter 3 for more details).

When the data copy has finished, the variable Erase_Status is set to ERASE. This indicates that there are data blocks that need to be erased. The user must call the function EEPROM_vEraseDataBlock to erase the filled data blocks. The function must be called again until the Erase_Status indicates CLEAR. The erase function can be called at any time, and so the CPU can perform the Flash erase operation when it is not performing any critical task. When the last data block is filled, the algorithm will erase the data blocks that the user did not erase before wrapping around and starting again in the first data block. This option is recommended for timing-critical applications, which need the flexibility to determine when the old data block should be erased.

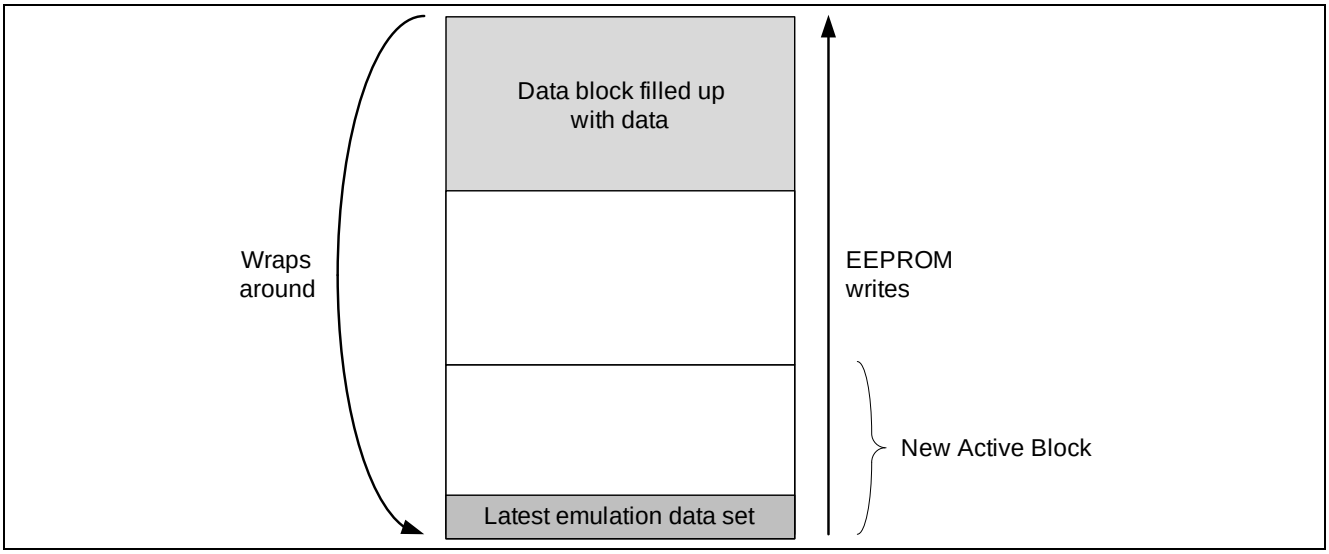

Figure 2 EEPROM Emulation Algorithm

The following figures illustrate how the algorithm works on two, 4 Kbyte data blocks and an emulation data set of 512kbytes (4 EEPROM pages).

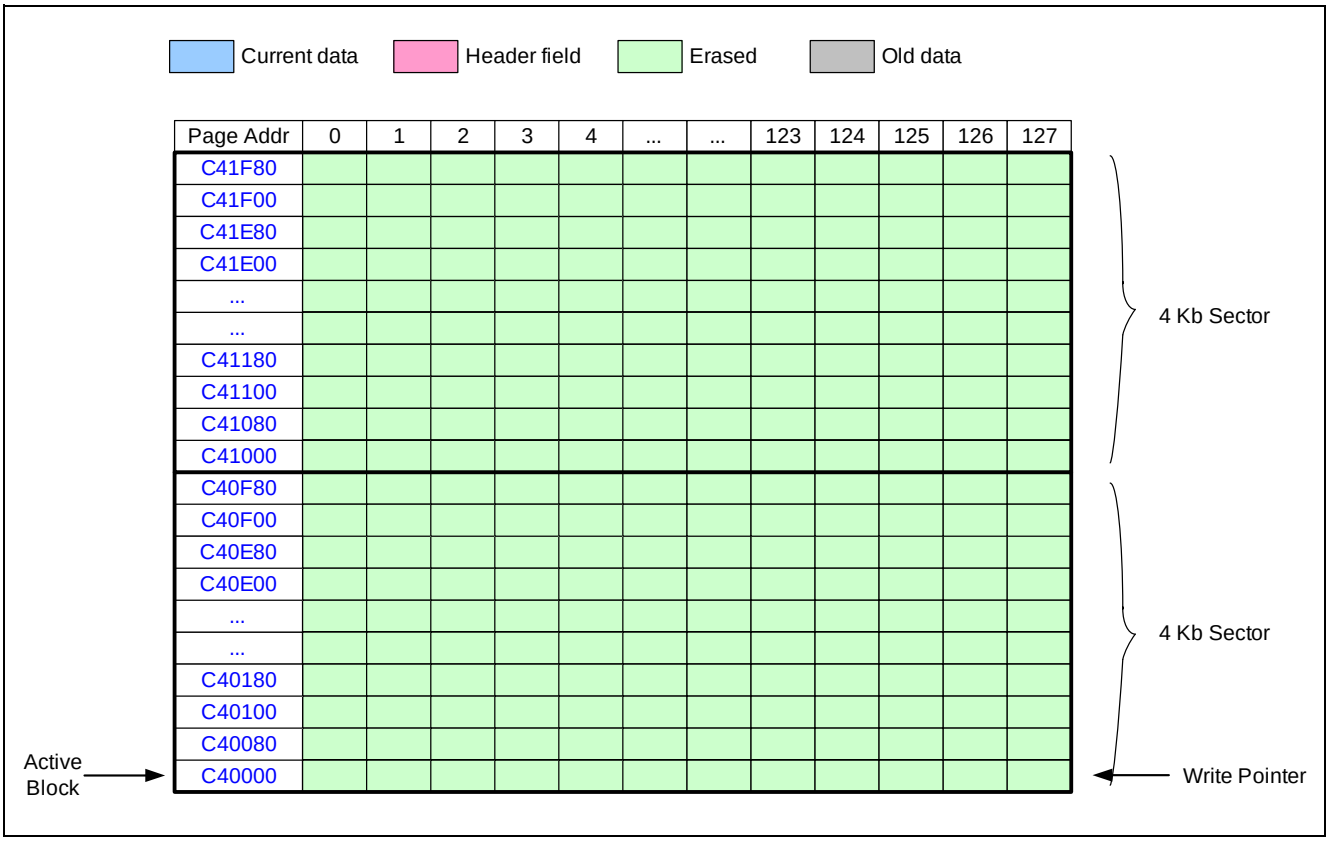

Figure 3 Data Blocks are Erased when Initialized by Calling the EEPROM_vInit Function

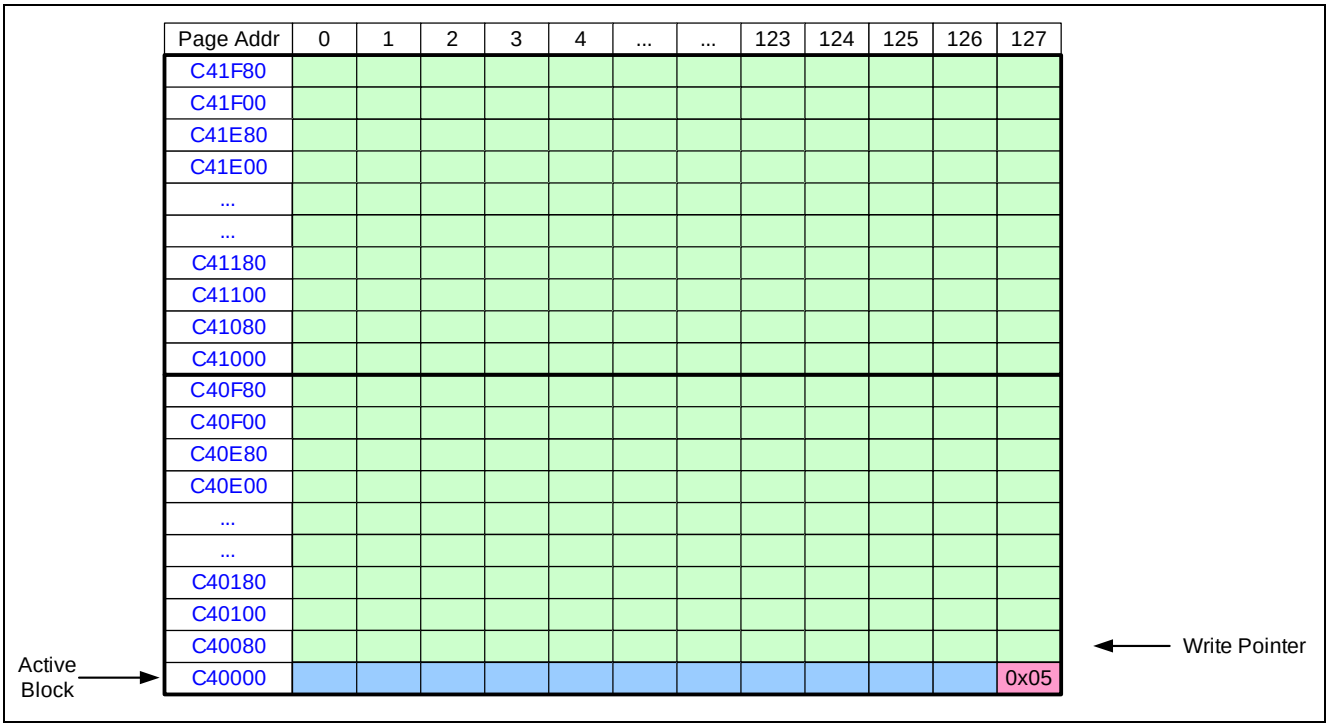

Figure 4 Data is Written into EEPROM Address 0

Consecutive writes to the data block. Only the last written data for the same address will be interpreted as the current data.

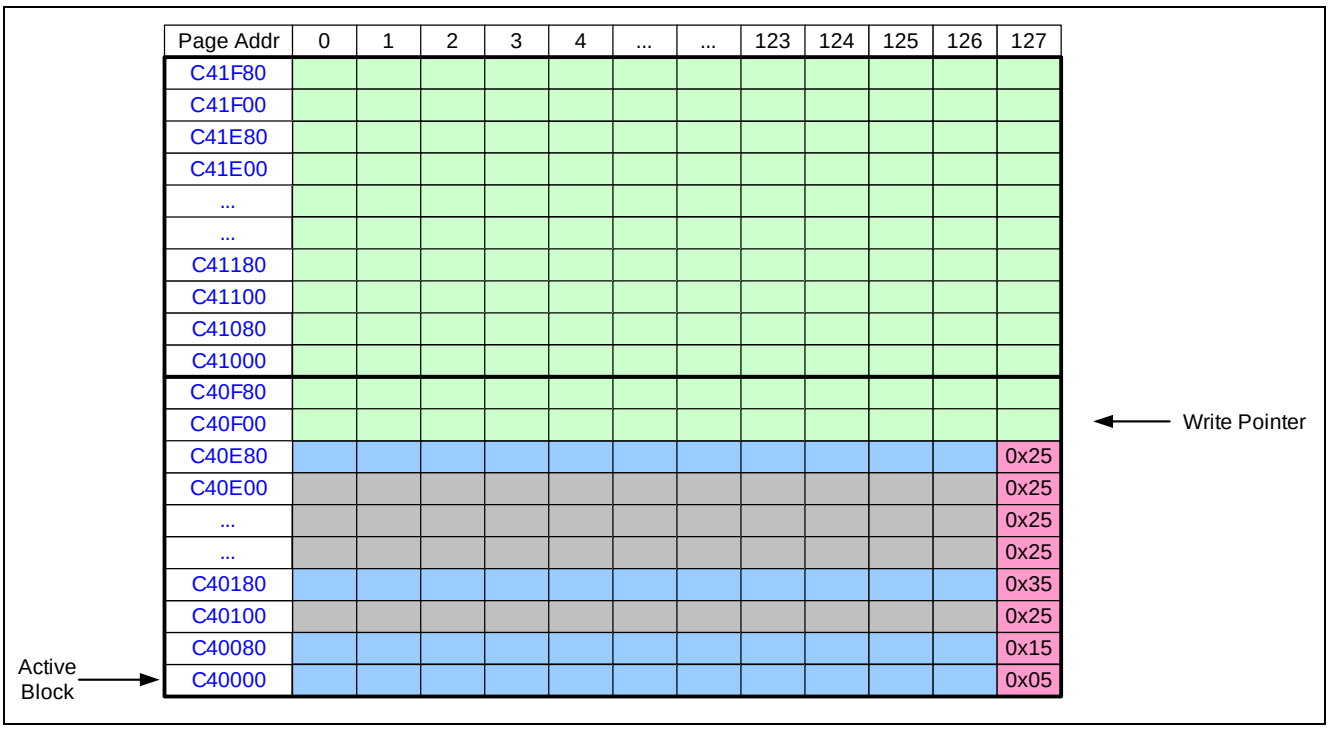

Figure 5 Consecutive Writes to the Data Block

Subsequent writes to fill-up the whole active block. The write pointer is now pointing to the next data block.

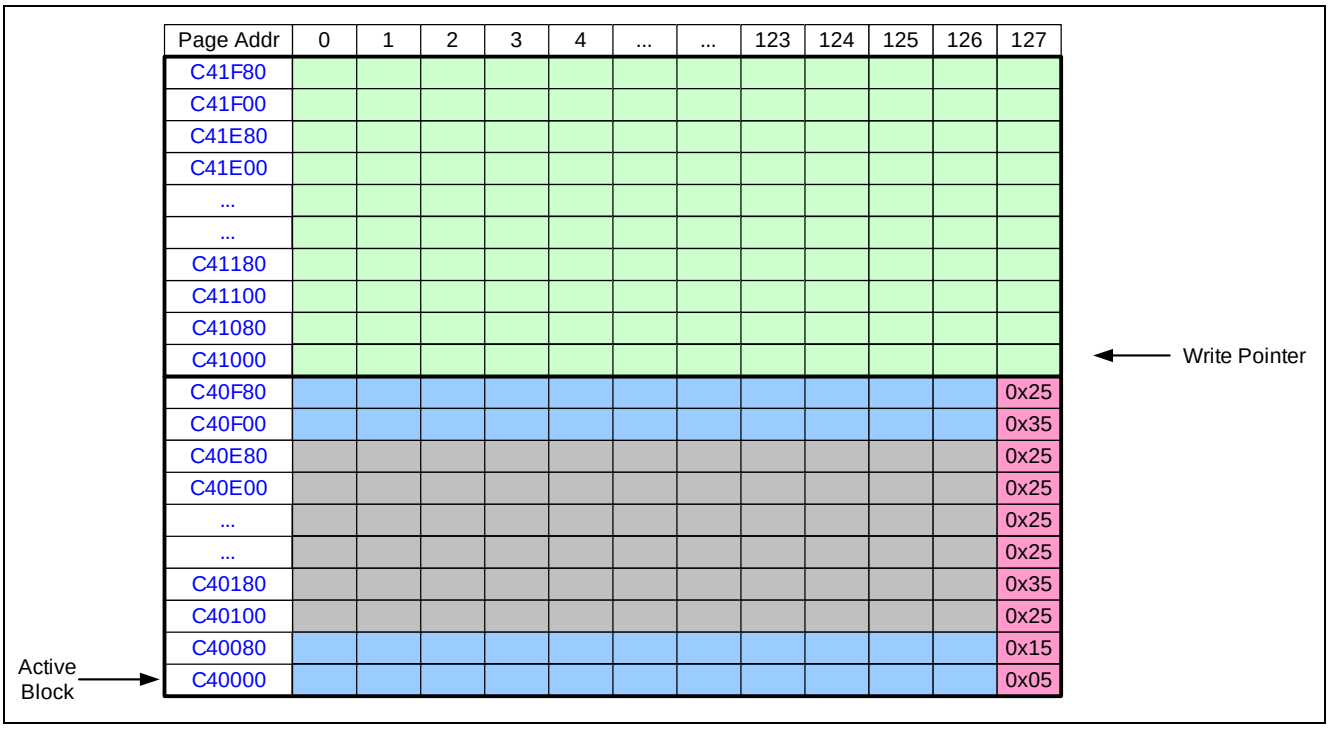

Figure 6 Subsequent Writes fill-up the Active Block

When the write pointer is pointing to the next data block it will start to copy the current data to the new data block. It is not possible to write to the emulated EEPROM until the entire emulated data set is copied to the new data block.

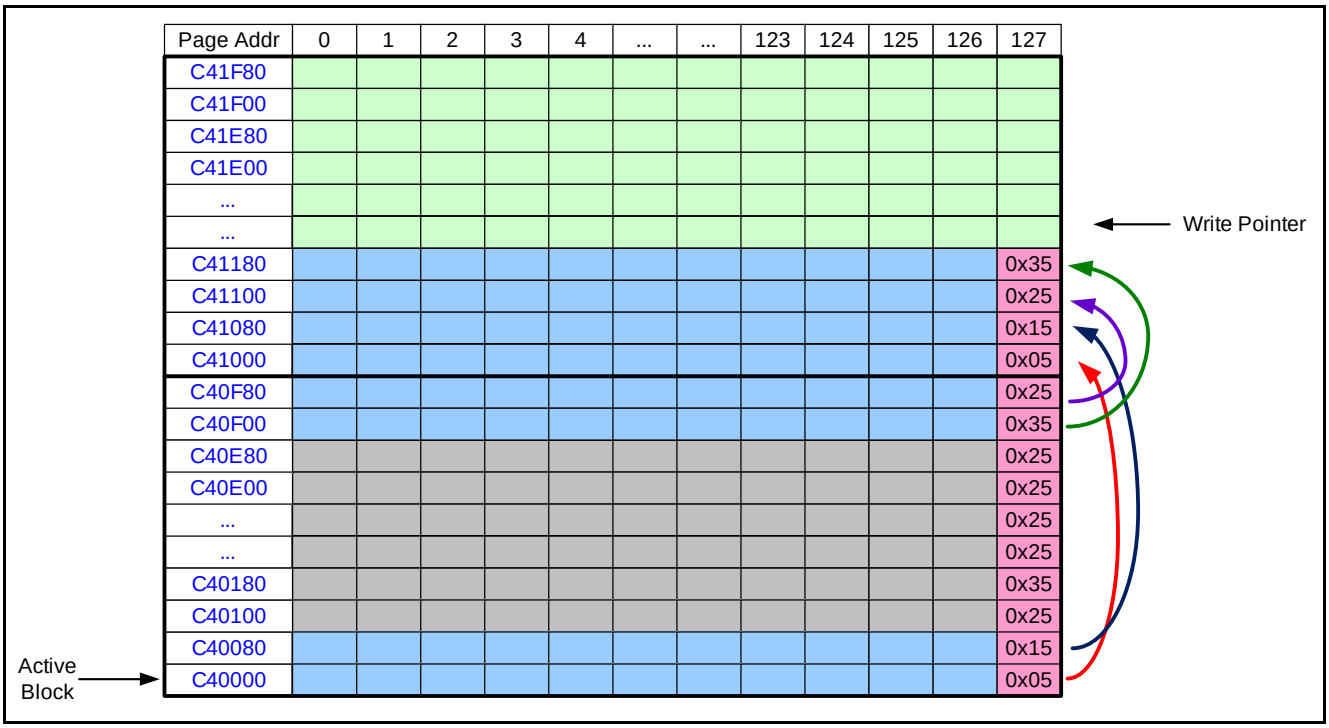

Figure 7 Copy latest Data Set to the new Data Block

After copying all of the current data into the new data block, the Active Block pointer address is updated and the old data block should be erased by calling the EEPROM_vEraseDataBlock function.

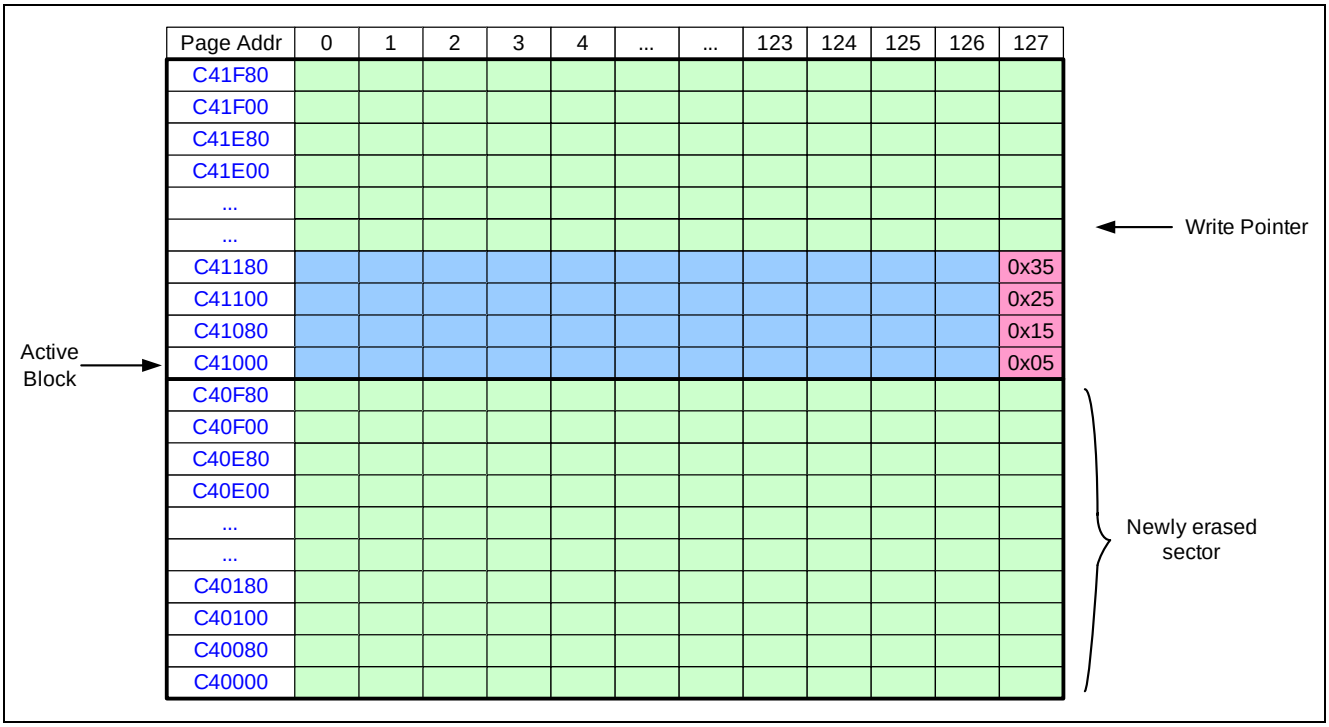

Figure 8 Update Active Block Pointer and Erase the old Data Block

4.3 System Reset and Error Handling

After system reset or a power failure, the EEPROM driver must be re-initialized by calling the EEPROM_Init function. During initialization, the active block is identified and the write pointer address restored.

If the data in the new active block is incomplete, the driver checks the previous block to look for the complete emulation data set. If the complete emulation data set is found, the incomplete data set in the active block will be erased and the complete emulation data set is copied to the active block.

If no complete emulation data set is found in the previous block, it will return DATA_INCOMPLETE after initialization.

When EEPROM_Init returns DATA_INCOMPLETE after initialization, the driver is considered to have successfully initialized, and so it is possible to read and write from the emulated EEPROM. The user application is therefore able to read the missing data based on the EEPROM logical address, and write new data into the emulated EEPROM.

If all the data blocks in the emulated EEPROM are filled, the driver returns DATA_FULL. The recommendation is then to delete the data in Flash memory and run the EEPROM initialization again. This is necessary because the EEPROM driver is not yet ready for any EEPROM operation. If the initialization is successful, all data blocks except the active block will be erased during the EEPROM driver initialization, avoiding an incomplete erase which can be caused by power failure during the erase process. If all the data blocks used for EEPROM emulation are empty (i.e. it is the first time that EEPROM emulation has been used), all data blocks are erased.

4.4 Read EEPROM

To read data from the emulated EEPROM, call the function *EEPROM_intRead*.

Each read fetches one EEPROM page (128 bytes) to the user's data buffer. The user must provide the EEPROM logical address and also the data pointer of the RAM buffer when calling the function.

When an EEPROM read is requested, the function searches the header field in the active data block to match the supplied address. The search will begin from the write pointer address to the start address of the active block. The first EEPROM page that matches the address will be the latest data for that particular address.

Note: The algorithm does not provide consistent read time. The read time is longer if the data to be read is situated further from the write pointer address. A consistent read time can be achieved if all the physical flash address of the EEPROM logical address is buffered and accessed from RAM.

The read function can be called any time, even during data copying or data block erasing. If a read request is submitted when the Flash memory is busy, the read function will not start immediately, but wait until the Flash is available again.

5 EEPROM Emulation Driver

The EEPROM emulation driver consists of two files, as shown in **Table 2**:

Table 2 EEPROM Emulation Driver Files

File Name	Description
eeprom.c	The EEPROM emulation driver source file
eeprom.h	The header file, including user-configurable parameters

5.1 User-Configurable Parameters

The parameters listed in **Table 3** are to be configured in the **eeprom.h** header file according to the application requirements.

Table 3 User-Configurable Parameters

Parameter Name	Description
EEPROM_START_ADDRESS	The physical start address of the Flash memory address used for EEPROM emulation. - EEPROM start address must be aligned to that of a 4 kbytes Flash sector.
NO_OF_PAGE	The total EEPROM page required for emulation. - Each page is 128 bytes, and contains a EEPROM logical address.
EEPROM_BLOCK_SIZE	The data block size used to hold the emulation data set. - The size must be in the multiples of 4 kbytes. - The minimum size is one sector, which is equivalent to 4 kbytes.
TOTAL_EMULATION_SIZE	The total size in Flash memory used for EEPROM emulation. - The size must be in multiples of EEPROM_BLOCK_SIZE. - The minimum size is 2 times the EEPROM_BLOCK_SIZE.
FLASH_MODULE	The Flash module number.
MASKING_VALUE	This value will be masked on the last 16 bits of the EEPROM page to determine the size of the header. The size of the header will be determined by this value. The minimum masking size is 1 byte (0x00FF). For example, a MASKING_VALUE of 0x0FFF refers to 12 bits of the header.
VALIDITY_BIT	The number of bits for 0x05 to be masked with the EEPROM logical address. For example, if 2 bits are used, only 01_B will be used as the validity bit instead of 101_B. If 12 bits are used for the header, 10 bits will be used as the EEPROM logical address.
ADVANCE_MODE	**ON** For timing-critical applications where the user application is required to handle EEPROM_vIntProcess and EEPROM_EraseDataBlock functions by themselves. **OFF** For applications which are **not** time-critical, where the EEPROM driver will handle the full sequence of a write operation, including erasing data blocks as necessary.

5.2 EEPROM Emulation Driver Function

Table 4 summarizes the EEPROM emulation driver functions used to run the emulation.

Table 4 Summary of EEPROM Emulation Driver Functions

Function Name	Description	Link
EEPROM_Init	Initializes the EEPROM emulation driver.	Section 5.2.1
EEPROM_intWrite	Writes data to the emulated EEPROM with the EEPROM logical address.	Section 5.2.2
EEPROM_intRead	Reads the latest emulated EEPROM data with the supplied EEPROM logical address.	Section 5.2.3
EEPROM_vIntProcess	The call back function after the write operation.	Section 5.2.4
EEPROM_vEraseDataBlock	The function to erase the filled data block.	Section 5.2.5

5.2.1 EEPROM_Init Function

The EEPROM_Init function is used to initialize the EEPROM emulation driver. It must be called after a reset, and only needs to be called once in-between resets.

During compilation, if the user configurable parameters are invalid, the compiler indicates an error and its cause. The user must therefore be careful in configuring the parameters to their specific requirements.

When this function is called in the user application, it locates the active block; i.e. the data block that contains the latest data. The function then proceeds to verify whether the active block contains ‘complete’ data. If incomplete data is found, data recovery will be executed if possible.

The EEPROM_Init function also provides the driver with the physical flash address for the write pointer. If the recovered data is found to be an incomplete emulation data set, the function will return DATA_INCOMPLETE. If all the data blocks are full and data recovery is not possible, the function will return DATA_FULL. If there is no data found, the function will return DATA_EMPTY.

After identifying the active block location, all other data blocks are erased to avoid the situation of Flash cells that are not erased due to an incomplete erase. If there is no data found, all sectors inside the total emulation size are erased.

Table 5 EEPROM_Init Function

Function Name	EEPROM_Init
Function Prototype	int EEPROM_Init (void)
Input Parameter	None
Return Value	01_H: INIT_SUCCESS (Initialization is complete with complete data in the Flash) 02_H: DATA_INCOMPLETE (Initialization is complete with incomplete emulation data set) 04_H: DATA_EMPTY (Initialization is complete and no data found in Flash) 08_H: DATA_FULL (Initialization failed, all data blocks are full)

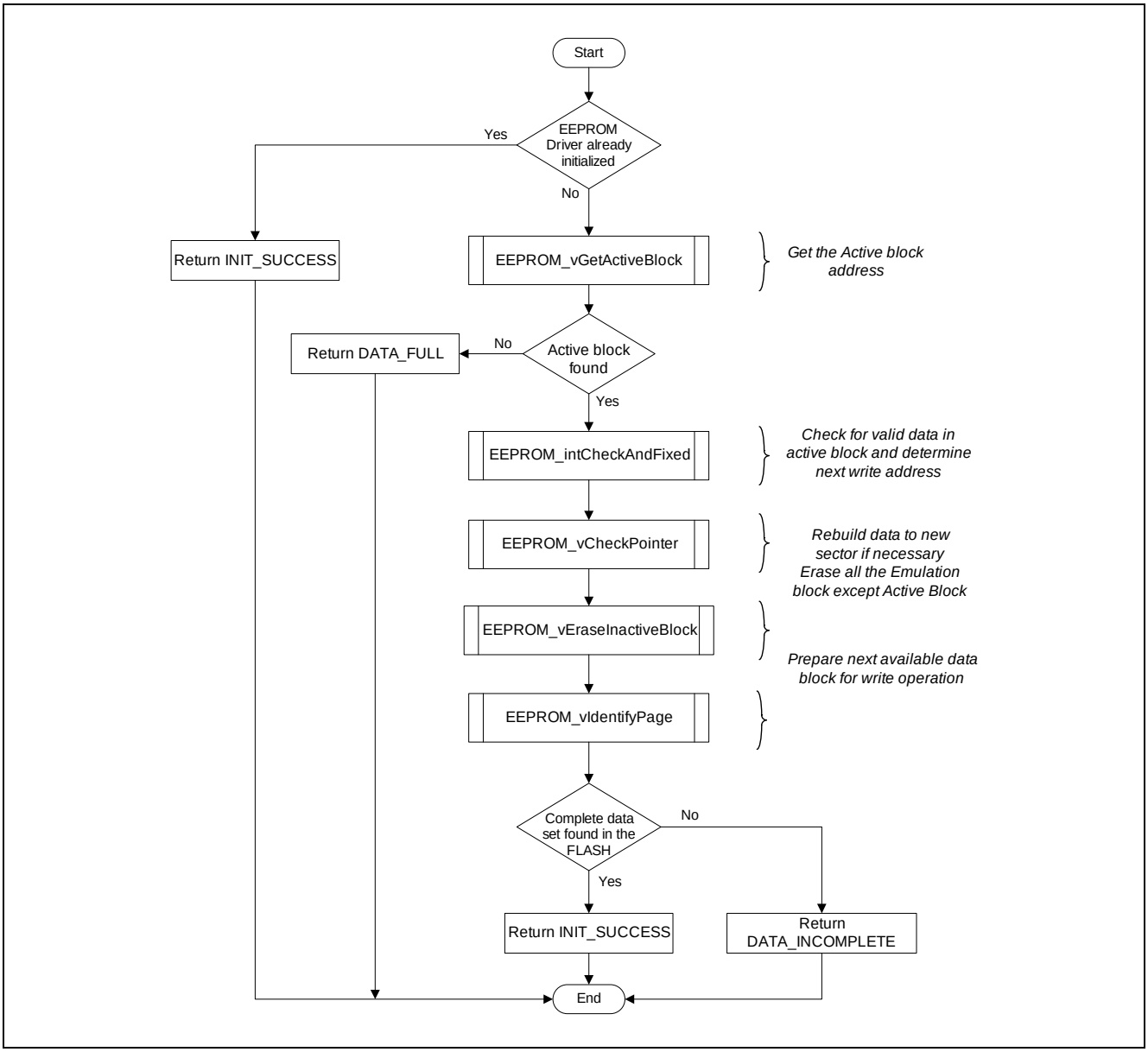

Figure 9 Flow Chart of EEPROM_Init Function

5.2.2 EEPROM_intWrite Function

This function writes data in to the emulated EEPROM. It must be called with a word-aligned data pointer pointing to the location of the data to be stored, together with the EEPROM logical address. This function will check whether the address provided is within the range of the NO_OF_PAGE in the user-configurable parameters.

The EEPROM_intWrite function also checks for the driver status and returns EEPROM_BUSY if the driver is busy. This is because a write process is not allowed when another EEPROM write is in progress.

If the driver is not busy, the data will be programmed into the emulated EEPROM. The EEPROM logical address will be masked and inserted into the EEPROM page as the header.

Table 6 EEPROM_intWrite Function

Function Name	EEPROM_intWrite
Function Prototype	int EEPROM_intWrite (uword *DataPtr, uword address)
Input Parameter	DataPtr: Pointer to the write data buffer in RAM address: The EEPROM logical address for the emulation data set
Return Value	00_H: DRIVER_NOT_INIT (EEPROM driver initialization is incomplete) 01_H: COMPLETE (Data is written to Flash memory) 02_H: CONFIG_ERROR (Invalid EEPROM logical address) 04_H: EEPROM_BUSY (Driver is currently busy with another process)

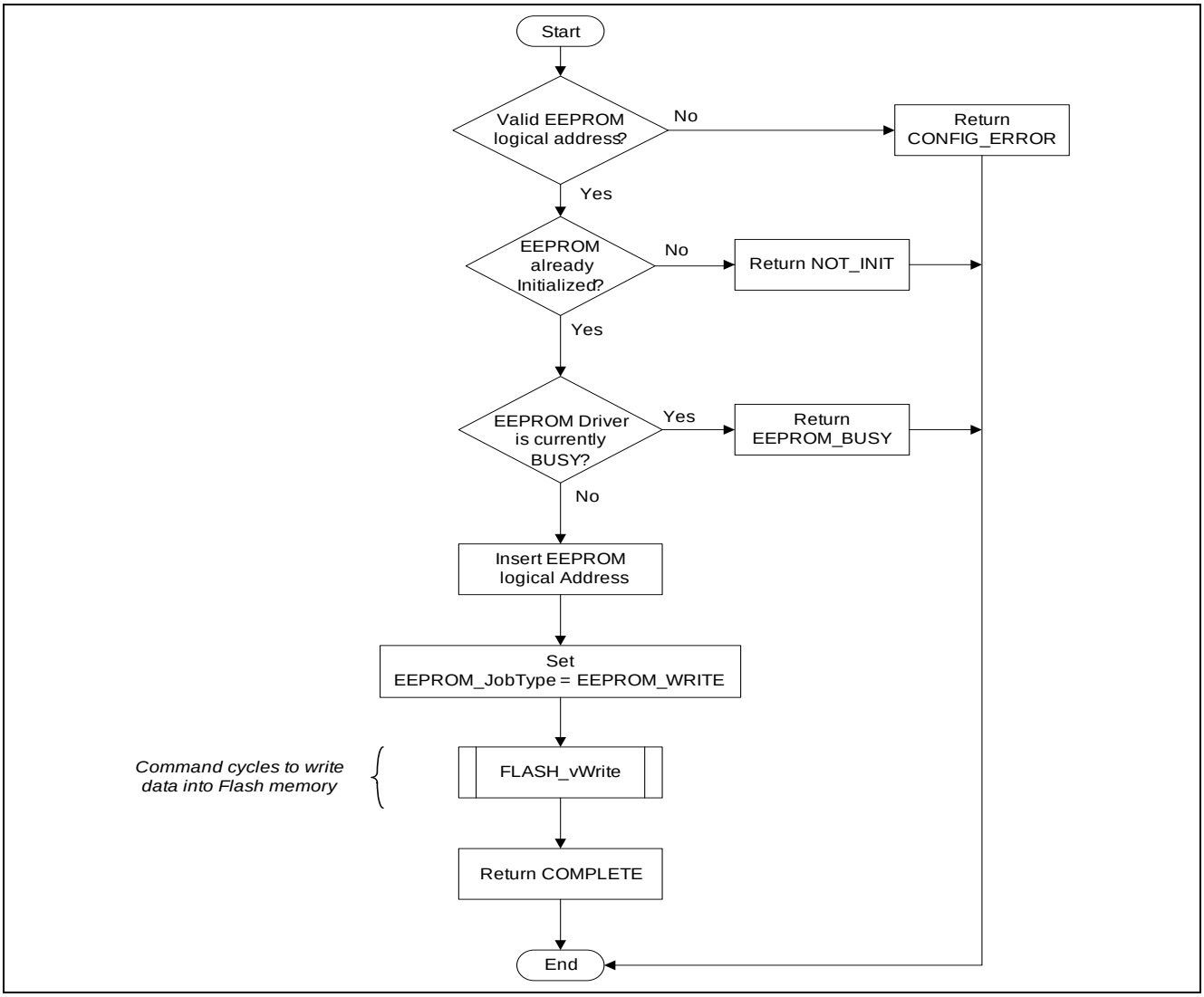

Figure 10 Flow Chart of EEPROM_intWrite Function

5.2.3 EEPROM_intRead Function

The EEPROM_intRead function loads the data with the supplied EEPROM logical address.

This function requires the user to provide a data pointer pointing to the buffer location when the data is fetched, and the EEPROM logical address. The function will check if the supplied EEPROM logical address is within the range of the NO_OF_PAGE set in the user-configurable parameters. It then locates the latest data with the EEPROM logical address provided in the Active Block. If the data with the address is available, it will load the data into the user's read buffer and return COMPLETE. If the data is not available, the function will return FAILED.

Table 7 EEPROM_intRead Function

Function Name	EEPROM_intRead
Function Prototype	int EEPROM_intRead (uword *buffer, uword address)
Input Parameter	buffer: Pointer to the user read buffer address: The page address of the emulation size
Return Value	00_H: DRIVER_NOT_INIT (EEPROM driver initialization is incomplete) 01_H: COMPLETE (Data is written to Flash memory) 02_H: CONFIG_ERROR (Invalid EEPROM logical address) 04_H: FAILED (Data with the supplied logical address is not found

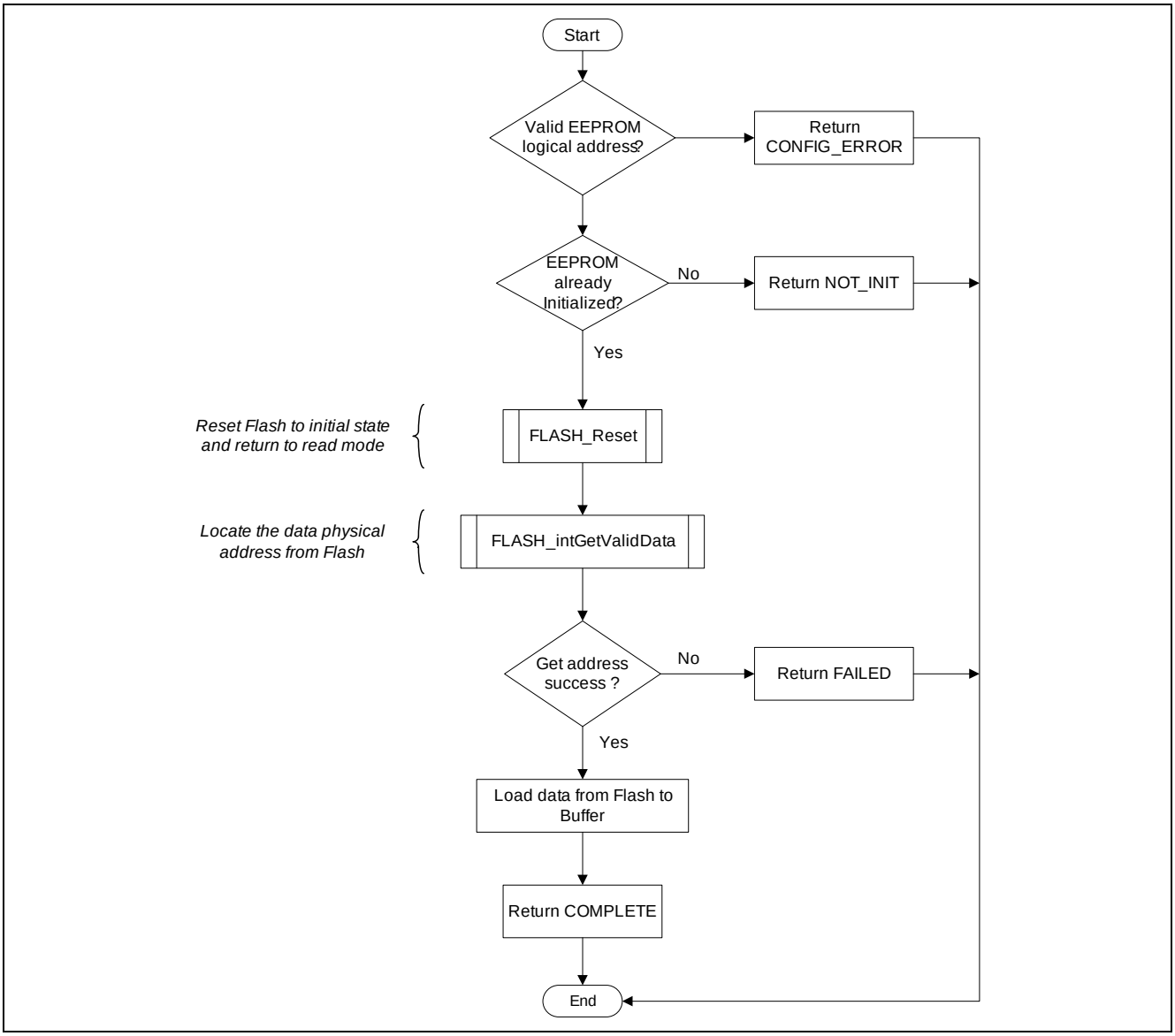

Figure 11 Flow Chart of EEPROM_intRead Function

5.2.4 EEPROM_vIntProcess Function

The user must call the EEPROM_vIntProcess function after every EEPROM write operation if ADVANCE_MODE is ON.

The EEPROM_vIntProcess performs the following actions:

- Updates the write pointer to the next physical write address of the Flash memory after the EEPROM write operation is complete
- Copies the emulation data set to a new data block when the active data block is full
- Updates Erase_Status when there are data blocks that need to be erased.

If the EEPROM_JobType status is not IDLE after calling the EEPROM_vIntProcess function for the first time, the function must be called again until the EEPROM_JobType status indicates IDLE.

Table 8 EEPROM_intCheckBusy Function

Function Name	EEPROM_vIntProcess
Function Prototype	void EEPROM_vIntProcess (void)
Input Parameter	None
Return Value	None

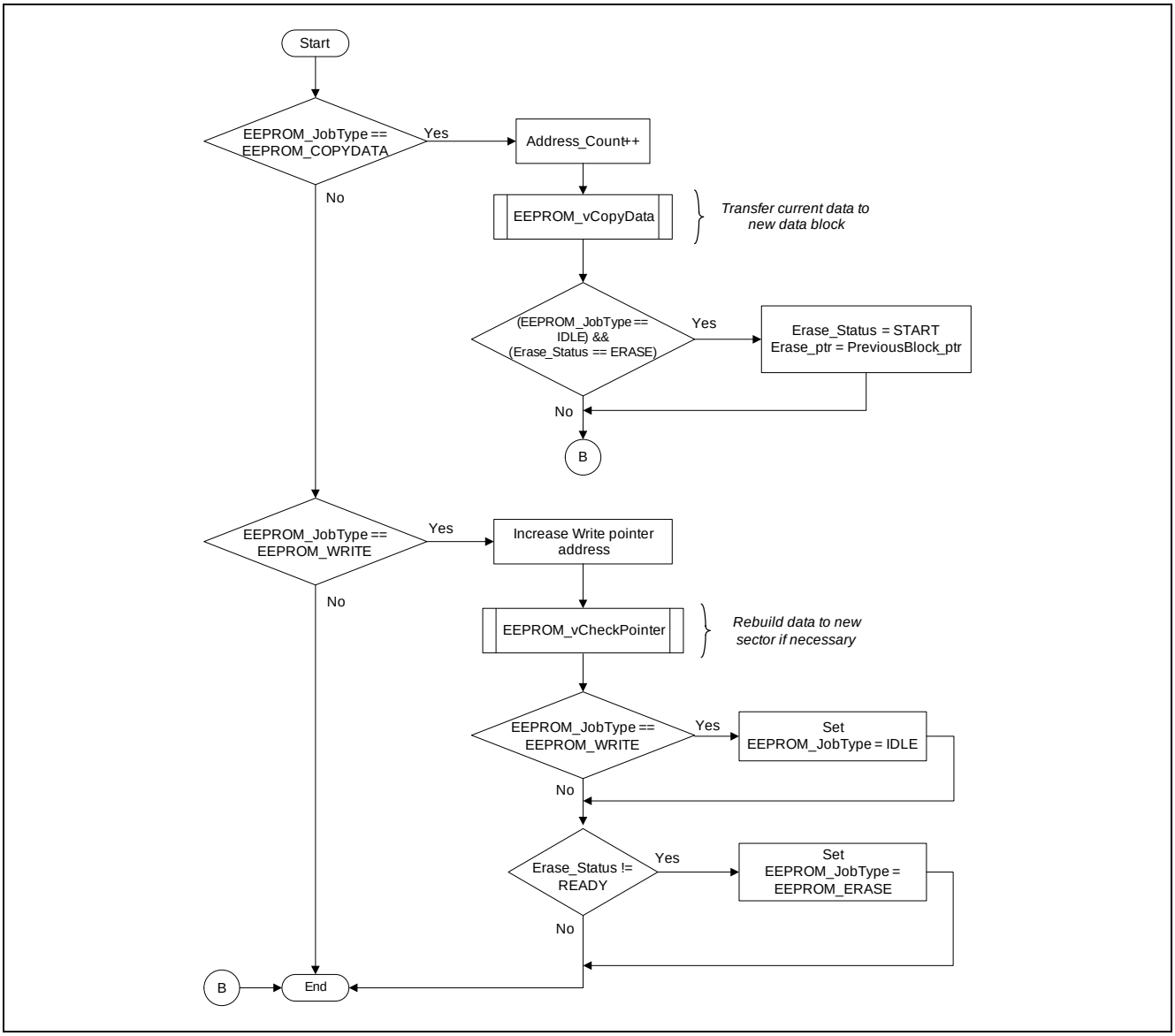

Figure 12 Flow Chart of EEPROM_vIntProcess Function

5.2.5 EEPROM_vEraseDataBlock Function

When ADVANCE_MODE is ON, the user application must erase full data blocks using the EEPROM_vEraseDataBlock function.

When a data block is full, the variable Erase_Status will be set to ERASE, indicating that there are data blocks to be erased. After calling the function, Erase_Status will change to START.

The function must be called again until Erase_Status indicates CLEAR.

After each call to the function it is necessary to poll the Flash 'busy' bit until the Flash is available again, before re-calling this function.

Table 9 EEPROM_vEraseDataBlock Function

Function Name	EEPROM_vEraseDataBlock
Function Prototype	void EEPROM_vEraseDataBlock (void)
Input Parameter	None
Return Value	None

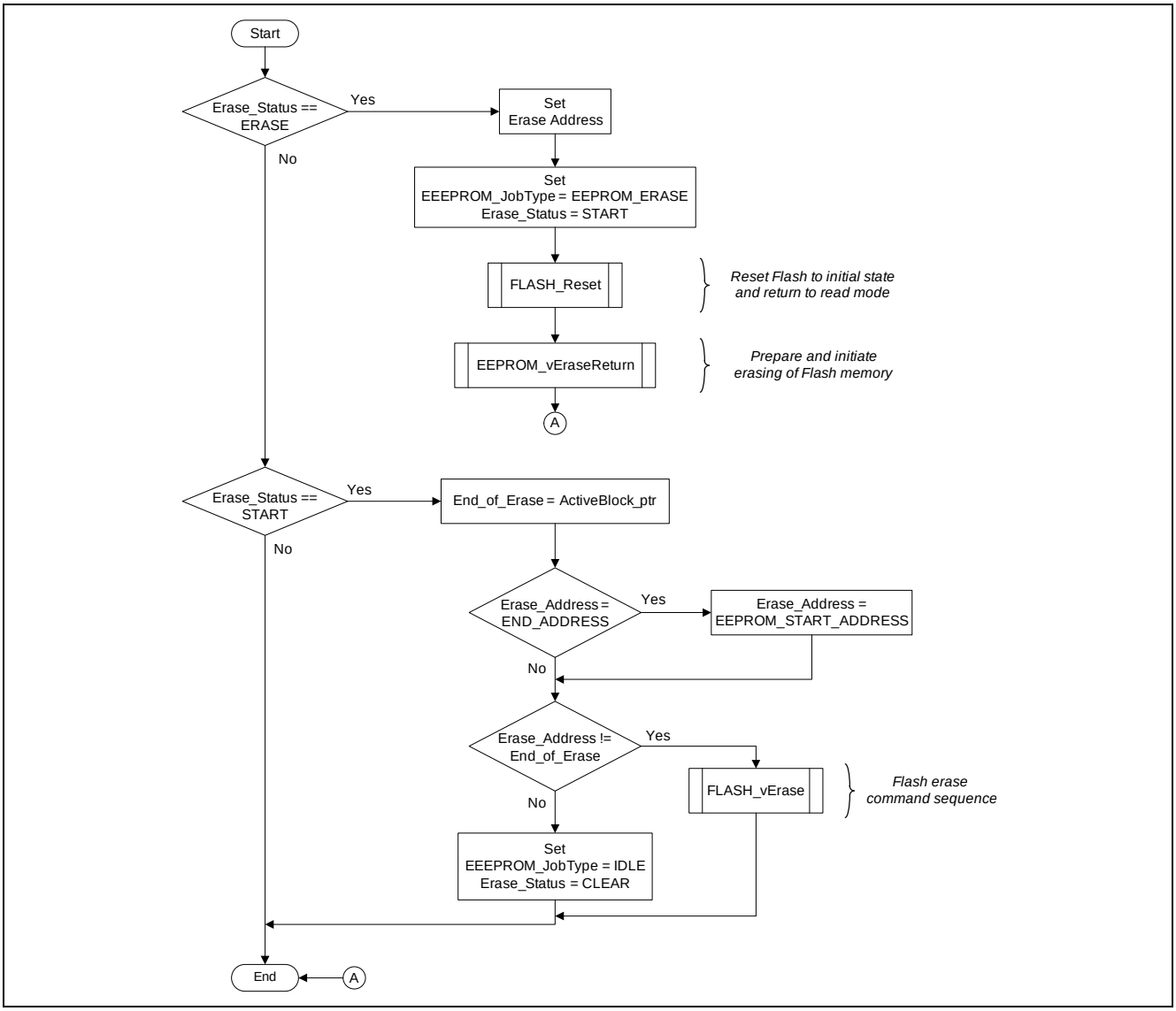

Figure 13 Flow Chart of EEPROM_vEraseDataBlock Function

5.3 Status Variable and Data Pointer

The following status variables and data pointers are applicable for the user application when ADVANCE_MODE is set to ON.

Table 10 Summary of EEPROM Emulation Driver Status Variables

Variable Name	Description	Link
EEPROM_Status	Initialization status of the EEPROM driver	Section 5.3.1
EEPROM_JobType	Current EEPROM driver job status	Section 5.3.2
Erase_Status	Status for erasing data blocks	Section 5.3.3
FLASH_BUSY_STATUS	Flash BUSY status	Section 5.3.4

Table 11 Summary of EEPROM Emulation Driver Data Pointer

Data Pointer	Description	Link
write_ptr	Initialization status of the EEPROM driver	Section 5.3.5
Erase_ptr	Current EEPROM driver job status	Section 5.3.6

5.3.1 EEPROM_Status

The variable EEPROM_Status indicates the EEPROM driver initialization status. This variable will be set to 1 only if the EEPROM driver is initialized.

Table 12 EEPROM_Status

Variable Name	EEPROM_Status
Return Value	00_H: EEPROM driver is not yet initialized 01_H: EEPROM driver is already initialized

5.3.2 EEPROM_JobType

The variable EEPROM_JobType indicates the current EEPROM driver job status.

The user must check this variable after calling the function EEPROM_vIntProcess following a EEPROM write request. If this variable is not set to IDLE, the EEPROM_vIntProcess function must be called and then the status of this variable must be re-checked. The process must continue until the variable indicates IDLE.

This variable also allows the user to monitor the current EEPROM driver job status because if the driver job status is EEPROM_WRITE or EEPROM_COPYDATA, it is not allowed to write to the emulated EEPROM.

Table 13 EEPROM_JobType

Variable Name	EEPROM_JobType
Return Value	00_H: IDLE 01_H: EEPROM_WRITE 02_H: EEPROM_ERASE 04_H: EEPROM_COPYDATA

5.3.3 Erase_Status

The variable Erase_Status indicates that there are data blocks to be erased when its value is ERASE. When the function to erase data blocks is called, the variable status will change to START. After all full data blocks are erased, it will read CLEAR.

Table 14 Erase_Status

Variable Name	Erase_Status
Return Value	00_H: CLEAR 01_H: START 02_H: ERASE

5.3.4 FLASH_BUSY_STATUS

The variable FLASH_BUSY_STATUS indicates the Flash array status.

When the Flash status is BUSY, no other Flash operation is allowed. Any Flash operation during this time will stall the CPU. The Flash will be in BUSY state while the Flash write or sector erase operation is in progress.

Table 15 FLASH_BUSY_STATUS

Variable Name	FLASH_BUSY_STATUS
Return Value	00_H: READY Other: BUSY

5.3.5 Write_ptr

Data pointer Write_ptr holds the physical address of the next write location.

By reading the value of Write_ptr, the physical address of the next write can be determined.

This data pointer is updated when EEPROM_vIntProcess is called after a write to the emulated EEPROM.

5.3.6 Erase_ptr

Data pointer Erase_ptr holds the physical address of the sector that will be erased when the function EEPROM_vEraseDataBlock is called. This allows the user to identify the sector to be erased and avoid writing data into data blocks that have not been erased.

6 EEPROM Emulation Examples

This chapter provides example code to illustrate the use of the EEPROM emulation driver.

6.1 Basic Mode

The driver operates in basic mode when the ADVANCE_MODE parameter is set to OFF. In basic mode, the call to the EEPROM_vIntProcess function after a write operation is handled by the driver, as is the erase of full data blocks. In this mode the user application only needs to call the EEPROM_intWrite function when it is required to store data in the emulated EEPROM. A write operation takes longer in this mode, so it is only recommended for applications that are not timing-critical.

6.1.1 EEPROM Driver Configuration for Basic Mode

Figure 14 shows an example configuration of the parameters in the **eeprom.h** file for basic mode:

```
/*****************************************************************************
    User-Configurable Parameters
*****************************************************************************/

#define EEPROM_START_ADDRESS        (0x0c40000)

#define NO_OF_PAGE                  (5)

#define EEPROM_BLOCK_SIZE           (0x2000)

#define TOTAL_EMULATION_SIZE        (0x6000)

#define FLASH_MODULE                (1)

#define MASKING_VALUE               (0x00FF)

#define VALIDITY_BIT                (4)

#define ADVANCE_MODE                (OFF)
```

Figure 14 User-Configuration Parameters for Basic Mode

Table 16 Example Parameters for Basic Mode

Parameter Name	Value	Description
EEPROM_START_ADDRESS	0x0c40000	EEPROM emulation will start from the address 0x0c40000.
NO_OF_PAGE	5	There will be a total of 5 EEPROM pages in the emulation data set. The EEPROM logical address for 5 EEPROM pages will be from 0x00 to 0x04.
EEPROM_BLOCK_SIZE	0x2000	The size of each data block is 8 Kbytes.
TOTAL_EMULATION_SIZE	0x6000	A total of 24 Kbytes will be used for EEPROM emulation. There will be total of 3 data blocks in this configuration.
FLASH_MODULE	1	Flash array 1 is used for the EEPROM emulation.
MASKING_VALUE	0x00FF	1byte will be used for the header.
VALIDITY_BIT	4	4 bits of 0x05 will be used; i.e. in the header, the lower 4 bits will be 0101_B and the upper 4 bits will be the EEPROM logical address.
ADVANCE_MODE	OFF	This means the EEPROM driver is running in Basic Mode and will call the EEPROM_vIntProcess function and erase data blocks as necessary.

6.1.2 Write to EEPROM for Basic Mode

To write data into the emulated EEPROM a data pointer is required, pointing to the first address of the data to write into the emulated EEPROM. This can be achieved by declaring a 128-byte data array and a data pointer as shown:

```
//*****************************************************************************
//  Variables Declaration
//*****************************************************************************
uword *buffptr;
uword wordBuffer [64];

buffptr = &wordBuffer;
```

Figure 15 Declaration of Data Pointer and Data Array

To perform the write, the EEPROM_intWrite function must be called. This function requires the data pointer and the EEPROM logical address. When the function returns COMPLETE, this indicates that the data is programmed into the emulated EEPROM and is ready for the next write operation.

```
int RetStatus1;

  /* Write data from the buffer (wordBuffer)     */
  /* to the EEPROM logical address (address1)    */
  RetStatus1 = EEPROM_intWrite (buffptr, address1);
```

Figure 16 Write to Emulated EEPROM

The RetStatus1 in Figure 16 refers to the variable for the return status of the EEPROM_intWrite function. This variable indicates the result of the write operation.

6.2 Advance Mode

To enable Advance Mode, set ADVANCE_MODE to ON in the user-configurable parameters in the eeprom.h file.

In advance mode, the user application must handle the write operation sequence, but is then able to control the timing of the write operation.

The complete sequence of the write operation is:

- Call the EEPROM_intWrite function to write data into the emulated EEPROM.
- Wait for the Flash to be available and call the EEPROM_vIntProcess function until the EEPROM_JobType indicates IDLE, to update the EEPROM driver.
- Check for the Erase_Status and call the EEPROM_vEraseDataBlock function if Erase_Status is set to ERASE.

In between these functions the user application can perform other critical tasks, but the user always needs to be aware of the following:

- After calling EEPROM_intWrite, and before calling the EEPROM_vIntProcess function, another write to the emulated EEPROM is not allowed until EEPROM_JobType indicates IDLE. If not in IDLE and the EEPROM_intWrite function is called, the write will be blocked by the driver and return EEPROM_BUSY. The erase option, called with the EEPROM_vEraseDataBlock function, is also blocked if not in IDLE. Only the read operation is possible, but this will only start when the Flash is not in a busy state. After calling the EEPROM_vIntProcess function and after updating the EEPROM driver, the driver is ready for the next write or erase operation.
- Full data blocks can be erased at any time if the EEPROM_JobType is IDLE, but the erase operation moves the Flash into a busy state and stalls the CPU from accessing the Flash memory. Therefore the user application can do this when the CPU is not performing any critical tasks, but it must ensure that the next data block after the active block is empty. To check this, examine the Write_ptr and Erase_ptr pointer addresses.

In Advance Mode, one of two approaches can be used for checking the state of read/write operations:

- **“Polling Method” on Page 30**
- **“Interrupt Method” on Page 31**

6.2.1 Polling Method

Using a polling method, the user application must poll for the Flash busy flag to determine when the write or erase operation is complete.

6.2.1.1 Write to EEPROM (Polling Method)

To write data into the emulated EEPROM using the polling method, first call EEPROM_intWrite with the data pointer and the EEPROM logical address. After calling the function, poll for the Flash busy bit until it is available. Once available, call the EEPROM_vIntProcess function. If the EEPROM_JobType is not IDLE, the EEPROM_vIntProcess function needs to be called again.

```
/* Write data to emulated EEPROM */
RetStatus1 = EEPROM_intWrite (buffptr, address1);

while(EEPROM_JobType != IDLE)
{
  while(FLASH_BUSY_STATUS != 0); /* Polls for Flash BUSY bit */
  EEPROM_vIntProcess();
}
```

Figure 17 Write to Emulated EEPROM (Polling Method)

6.2.1.2 Erase Data Blocks (Polling Method)

When the variable Erase_Status is ERASE or 02_H, it means that there are data block(s) to be erased. The user application therefore has to call the EEPROM_vEraseDataBlock function to erase the full data blocks.

Each call of the function erases 1 sector. If the size of the filled data blocks is more than 1 sector, the function needs to be called multiple times to completely erase all the full data blocks. However it is not necessary to erase all full data blocks in one go, so the user application can call the EEPROM_vEraseDataBlock function when the CPU is not handling any other critical tasks. No more blocks need to be erased once the Erase_Status indicates CLEAR or 00_H.

```
/* Erase filled data block if any */
if (Erase_Status == ERASE)
{
  while (Erase_Status!= CLEAR)
  {
    EEPROM_vEraseDataBlock();
    while(FLASH_BUSY_STATUS != 0); /* Polls for Flash BUSY bit */
  }
}
```

Figure 18 Erase Full Data Blocks (Polling Method)

6.2.2 Interrupt Method

In the interrupt based approach, the SCU interrupt must be enabled.

The SCU interrupt receives the interrupt request from the Program Flash Interrupt when the Flash busy bit changes from ‘1’ to ‘0’.

The SCU interrupt must be initialized in MAIN_vInit with the function shown in figure 19.

```
void uSCU_vInit (void)
{
  SCU_INTDIS    &= 0x1FFF;      /* Enabled Program Flash Interrupt      */
  SCU_0IC       |= 0x48;        /* Setting the Priority Group and Level */
  IMB_INTCTR    |= 0x01;        /* Enable Flash busy to ready interrupt */
}
```

Figure 19 SCU Program Flash Interrupt Initialization

6.2.2.1 Write to EEPROM (Interrupt Method)

When using the Interrupt based approach, the EEPROM_vIntProcess function should be called in the Interrupt Service Routine (ISR) of the SCU interrupt.

After calling the EEPROM_intWrite function, the user application can proceed to work on other tasks while the write to Flash process is performed in the background.

When the SCU Interrupt generates interrupts, the EEPROM_vIntProcess function will be called and updates any variables and pointers for the EEPROM driver as necessary.

```
    /* Write data to emulated EEPROM */
  RetStatus1 = EEPROM_intWrite (buffptr, address1);

  while(EEPROM_JobType != IDLE)     /* Proceed to work on other */
  {                                 /* things during writing to */
    TOGGLEIO_P10_0;                 /* Flash operation          */
  }
```

Figure 20 Write to Emulated EEPROM (Interrupt Method)

```
void __interrupt (0x6C) SCU0_ISR (void)
{
  if (((IMB_FSR_OP & 0x01) == 1) && (EEPROM_JobType != IDLE))
  {
      FLASH_ClearStatus();  /* Clear the interrupt Flag */
      EEPROM_vIntProcess();
  }
}
```

Figure 21 Interrupt Service Routine for SCU Interrupt

6.2.2.2 Erase Data Blocks (Interrupt Method)

To erase full data blocks in the interrupt method, call the EEPROM_vEraseDataBlock function when the Erase_Status is set to ERASE. This function will start the erase from the first sector of the filled data block.

Once the erase is complete, the SCU Interrupt is generated.

To erase all full data blocks in one go, the EEPROM_vEraseDataBlock function needs to be called again in the SCU Interrupt Service Routine. As the SCU interrupt is a shared interrupt, it is necessary to ensure that the generated interrupt is a result of the Flash operation of EEPROM driver and not some other process. This is done

by checking the register IMB_FSR_OP and the EEPROM_JobType, as shown in figure 22.

```
void __interrupt (0x6C) SCU0_ISR (void)
{
  /* Erase operation */
  if(((IMB_FSR_OP & 0x02) != 0) && (EEPROM_JobType == EEPROM_ERASE))
    {
        FLASH_ClearStatus();  /* Clear the interrupt Flag */
        EEPROM_vEraseDataBlock();
    }
  /* Write operation */
   else if (((IMB_FSR_OP & 0x01) == 1) && (EEPROM_JobType != IDLE))
    {
        FLASH_ClearStatus();  /* Clear the interrupt Flag */
        EEPROM_vIntProcess();
    }
}
```

Figure 22 Interrupt Service Routine for Erasing Data Blocks in Program Flash Interrupt

6.3 Read from EEPROM

To read out data from the emulated EEPROM, the EEPROM_intRead function is called. This function requires a data pointer and the EEPROM logical address.

The data pointer needs to point to the buffer location after the data is fetched. The buffer is created by declaring a data array of 128 bytes and pointing the data pointer to the first data array address. The function EEPROM_intRead can be called to run at any time. If the Flash is busy, it will wait until the Flash is available and proceed to fetch the data.

```
/*****************************************************************
    Variable Declaration
*****************************************************************/
    uword flashword [64];
    uword *Readbuff;
```

Figure 23 Declaration of Data Array and Data Pointer for Read from EEPROM

```
/* Point the pointer to the buffer */
Readbuff = &flashword[0];
/* Fetch EEPROM data with the EEPROM logical address */
RetStatus = EEPROM_intRead(Readbuff,address);
```

Figure 24 Read Data from Emulated EEPROM

The RetStatus is the variable that holds the return status of the EEPROM_intRead function. This variable is used to check whether the data fetch from the emulated EEPROM is successful or not.

The “address” in the function EEPROM_intRead is the variable that keeps the EEPROM logical address. For the example above, after data is fetched from the emulated EEPROM it is stored in the array Flash word.

6.4 Demonstration Program

The demonstration program available with this application note provides examples of how the EEPROM emulation driver functions can be called.

The demonstration program was developed and verified using the following hardware and software components:

- XC2000M Easykit board
- DAVE™ 2.0 for code generation
- TASKING VX-Toolset v2.5r1 for code compilation and debugging
- MTTTY as the terminal program on the PC host

For this demo, the driver is configured with the user-configurable parameters as shown in **Figure 14** expect with ADVANCE_MODE turned ON.

To work with the demo program HyperTerminal or a similar program, must be configured to 19.2 Kbaud on the ASC connection.

The General Purpose Timer (GPT) is used in the demonstration to generate an interrupt when the timer overflows. The Interrupt Service Routine then calls the write function to write the dummy data in the buffer into the emulated EEPROM.

When the demo program is started, a demo program menu is shown.

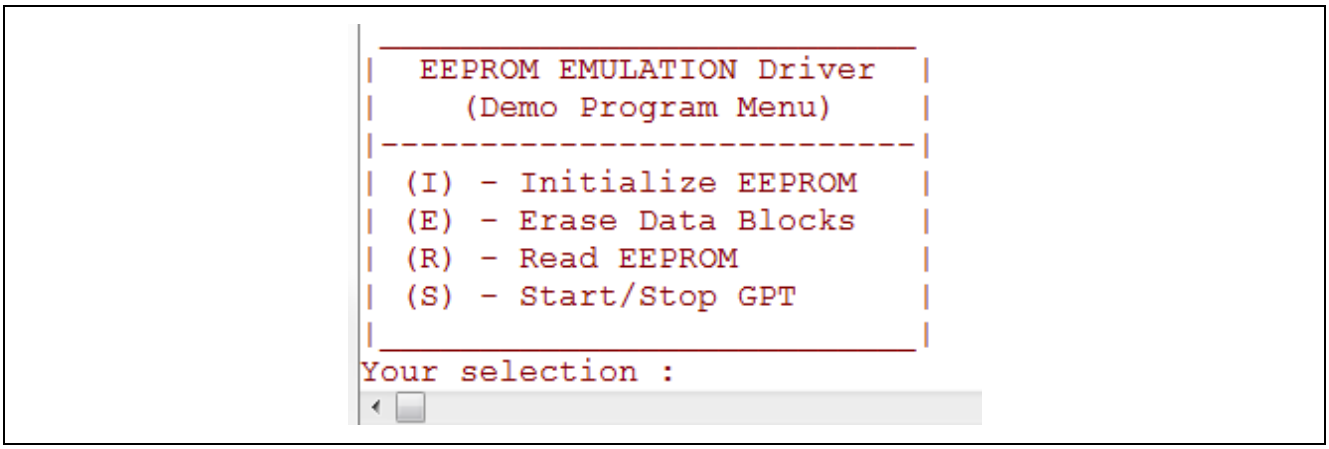

Figure 25 Demo Program Menu

6.4.1 EEPROM Driver Initialization

The EEPROM driver can be initialized with the “I” command on the demo program menu.

The function can be called as shown on figure 21. The return value from the initialization is used to determine the status of the EEPROM data in the Flash memory.

```
switch (select)
{
/*************************  EEPROM Initialize  ****************************/
case 'I':
case 'i':

 RetStatus = EEPROM_Init ();
 if (RetStatus == COMPLETE)
 {
   ASC_printf("\nEEPROM Initialization Success\n\r");
 }
else if (RetStatus == CONFIG_ERROR)
{
    ASC_printf("\nEEPROM Initialization Failed\n");
    ASC_printf("Configuration Error Occurs\n");
}
else if (RetStatus == DATA_INCOMPLETE)
{
    ASC_printf("\nEEPROM Initialization Success\n");
    ASC_printf("Data in Flash Incomplete!!\n");
}
else if (RetStatus == DATA_EMPTY)
{
    ASC_printf("\nEEPROM Initialization Success\n");
    ASC_printf("No data found in Flash!!\n");
}
else if (RetStatus == DATA_FULL)
{
    ASC_printf("\nEEPROM Initialization Failed\n");
    ASC_printf("No empty space found in Flash!!\n");
}
else
{
    ASC_printf("\nUnknown Error occurs\n");
}
break;
```

Figure 26 EEPROM Initialization

6.4.2 Write to EEPROM and Erase Data Blocks

To write data into the emulated EEPROM, use the “S” command on the demo program menu to start the GPT.

When the GPT is interrupted it will first call the EEPROM_intWrite function to write data into emulated EEPROM. It then proceeds to call EEPROM_vIntProcess until EEPROM_JobType indicates IDLE.

Next, it will check if there are data blocks that need to be erased by checking the Erase_Status. If the Erase_Status is ERASE, it will call the EEPROM_vEraseDataBlock function until the Erase_Status is CLEAR. It will then check for the return status from EEPROM_intWrite and prompt “Write success” when the return status is COMPLETE.

```
_interrupt(T3INT)  void GPT1_viTmr3(void)
{
  // USER CODE BEGIN (Tmr3,2)
    int RetStatus1;

      /* Write data from the buffer (wordBuffer)     */
      /* to the EEPROM logical address (address1)    */
     RetStatus1 = EEPROM_intWrite (buffptr, address1);

       while(EEPROM_JobType != IDLE)
       {
         while(FLASH_BUSY_STATUS != 0); /* Polls for Flash BUSY bit */
         EEPROM_vIntProcess();
       }
       /* Erase filled data block if any */
       if (Erase_Status == ERASE)
       {
         while (Erase_Status!= CLEAR)
         {
           EEPROM_vEraseDataBlock();
           while(FLASH_BUSY_STATUS != 0); /* Polls for Flash BUSY bit */
         }
       }
       /* Check write to EEPROM status*/
       if (RetStatus1 == COMPLETE)
       {
         ASC_printf("\nWrite success\n");
         address1++;
       }
       else if (RetStatus1 == EEPROM_BUSY)
       {
         ASC_printf("\nEEPROM driver is BUSY\n");
       }
       else
       {
         ASC_printf("\nWrite Failed: %d\n",RetStatus1);
       }
```

Figure 27 Call Routine for Writing Data to Emulated EEPROM on GPT Interrupt Service Routine

6.4.3 Manually Erasing Data Block(s)

To manually erase full data blocks to simulate the application which would erase data blocks separately, by-pass the check of the Erase_Status on the GPT Interrupt Service Routine.

This Erase_Status is still set to ERASE, but the erase of data blocks is only performed when users enter the “E” command on the demo program menu. Each “E” command will erase operations on one sector, therefore several erase operations are required to completely erase the full data blocks.

Attention: To manually erase or delay the erase of data blocks to a time when the CPU is not performing critical tasks, ensure that the Write_ptr address does not overlap with the Erase_ptr address.

```
_interrupt(T3INT)  void GPT1_viTmr3(void)
{
  // USER CODE BEGIN (Tmr3,2)
    int RetStatus1;

      /* Write data from the buffer (wordBuffer)     */
      /* to the EEPROM logical address (address1)    */
     RetStatus1 = EEPROM_intWrite (buffptr, address1);

        while(EEPROM_JobType != IDLE)
        {
          while(FLASH_BUSY_STATUS != 0); /* Polls for Flash BUSY bit */
          EEPROM_vIntProcess();
        }

        /* Erase filled data block if any
        if (Erase_Status == ERASE)
        {
          while (Erase_Status!= CLEAR)
          {
            EEPROM_vEraseDataBlock();
            while(FLASH_BUSY_STATUS != 0);
          }
        }*/

        /* Check write to EEPROM status*/
        if (RetStatus1 == COMPLETE)
        {
          ASC_printf("\nWrite success\n");
          address1++;
        }
        else if (RetStatus1 == EEPROM_BUSY)
        {
          ASC_printf("\nEEPROM driver is BUSY\n");
        }
        else
        {
          ASC_printf("\nWrite Failed: %d\n",RetStatus1);
        }
```

By-passed the checking of Erase_Status

Figure 28 By-pass check of Erase_Status

```
/*************************  Erase EEPROM  *****************************/
case 'E':
case 'e':

if (Erase_Status != CLEAR)
{
    EEPROM_vEraseDataBlock();
    while(FLASH_BUSY_STATUS != 0);
}
break;
```

Figure 29 Erase Data Blocks

6.4.4 Read from EEPROM

To read data from the emulated EEPROM use command “R” on the demo program menu.

The program will prompt the user for the EEPROM logical address.

EEPROM_intRead is then called with the address of the data buffer and the user submitted EEPROM logical address.

If the return value from the read function is COMPLETE, the program will read out the data from the data buffer.

```
/*************************  Read FLASH Value  *****************************/
case 'R':
case 'r':

ASC_printf("\rPlease input address:\n");
address = getMessage();
ASC_printf("%c\n", address);
address = address - 0x30;

/* Point the pointer to the buffer */
Readbuff = &flashword[0];
/* Fetch EEPROM data with the EEPROM logical address */
RetStatus = EEPROM_intRead(Readbuff,address);

if (RetStatus == COMPLETE)
{
  ASC_printf("\nRead success\n");

  for(count=0; count < 63; count++)
  {
     ASC_printf("%c", flashword[count]);
  }
}
else
```

Figure 30 Read from EEPROM