



EEPROM emulation in STM32F30x/STM32F31x STM32F37x/STM32F38x microcontrollers

Introduction

EEPROMs (Electrically Erasable Programmable Read-Only Memory) are often used in industrial applications to store updateable data. An EEPROM is a type of permanent (non-volatile) memory storage system used in complex systems (such as computers) and other electronic devices to store and retain small amounts of data in the event of power failure.

For low-cost purposes, external EEPROM can be replaced by on-chip Flash, with specific software algorithm.

This application note describes the software solution for substituting standalone EEPROM by emulating the EEPROM mechanism using the on-chip Flash of STM32F3xx^(a) devices.

Emulation is achieved by employing at least two pages in the Flash. The EEPROM emulation code swaps data between the pages as they become filled, in a manner that is transparent to the user.

The EEPROM emulation driver supplied with this application note meets the following requirements:

- Lightweight implementations offering a simple API that consists of three functions for initialization, read data and write data, and reduced footprint.
- Simple and easily updateable code model
- Clean-up and internal data management transparent to the user
- Background page erase
- At least two Flash memory pages to be used, more if possible for wear-leveling

Table 1 lists the microcontrollers concerned by this application note.

Table 1. Applicable products

Type	Applicable products
Microcontrollers	STM32F30x STM32F31x STM32F37x STM32F38x

a. STM32F3xx devices refer to STM32F30x/STM32F31x/STM32F37x/STM32F38x devices throughout the document.

Contents

- 1 Main differences between external and emulated EEPROM 5**
 - 1.1 Difference in write access time 6
 - 1.2 Difference in erase time 6
 - 1.3 Similarity in writing method 6

- 2 Implementing EEPROM emulation 7**
 - 2.1 Principle 7
 - 2.2 Case of use: application example 8
 - 2.3 EEPROM emulation software description 9
 - 2.4 EEPROM emulation memory footprint 12
 - 2.5 EEPROM emulation timing 13

- 3 Embedded application aspects 14**
 - 3.1 Data granularity management 14
 - 3.2 Wear-leveling: Flash memory endurance improvement 14
 - 3.2.1 Wear-leveling implementation example 14
 - 3.3 Page header recovery in case of power loss 15
 - 3.4 Cycling capability and page allocation 15
 - 3.4.1 Cycling capability 15
 - 3.4.2 Flash page allocation 16
 - 3.5 Real-time consideration 17

- 4 Revision history 18**

List of tables

Table 1.	Applicable products	1
Table 2.	Differences between external and emulated EEPROM	5
Table 3.	API definition.	10
Table 4.	Memory footprint for EEPROM emulation mechanism	12
Table 5.	EEPROM emulation timings with a 72 MHz system clock	13
Table 6.	Flash program functions	14
Table 7.	Application design.	17
Table 8.	Document revision history	18

List of figures

Figure 1.	Header status switching between page0 and page1	7
Figure 2.	EEPROM variable format	8
Figure 3.	Data update flow	9
Figure 4.	WriteVariable flowchart.	11
Figure 5.	Flash memory footprint for EEPROM emulation (mechanism and storage)	12
Figure 6.	Page swap scheme with four pages (wear-leveling).	15

1 Main differences between external and emulated EEPROM

EEPROM is a key component of many embedded applications that require non-volatile storage of data updated with byte or word granularity during run time.

Microcontrollers used in these systems are more often based on embedded Flash memory. To eliminate components, save PCB space and reduce system cost, the STM32F3xx Flash memory may be used instead of external EEPROM for simultaneous code and data storage.

Unlike Flash memory, however, external EEPROM does not require an erase operation to free up space before data can be rewritten. Special software management is required to store data in embedded Flash memory.

The emulation software scheme depends on many factors, including the EEPROM reliability, the architecture of the Flash memory used, and the product requirements.

The main differences between embedded Flash memory and external serial EEPROM are the same for any microcontroller that uses the same Flash memory technology (it is not specific to the STM32F3xx family products). The major differences are summarized in [Table 2](#).

Table 2. Differences between external and emulated EEPROM

Feature	External EEPROM (for example, M24C64: I ² C serial access EEPROM)	Emulated EEPROM using on-chip Flash memory
Write time	Random byte Write within 5 ms Word program time = 20 ms Page (32 bytes) Write within 5 ms Word program time = 625 μ s	Half-word program time: from 142 μ s to 41 ms ⁽¹⁾
Erase time	N/A	Page Erase time: from 20 ms to 40 ms ⁽²⁾
Write method	Once started, is not CPU-dependent. Only needs proper supply.	Once started, is CPU-dependent. If a Write operation is interrupted by CPU reset, the EEPROM Emulation algorithm is stopped, but current Flash write operation is not interrupted by a software reset. Can be accessed as half words (16 bits) or full words (32 bits).
Read access	Serial: a hundred μ s Random word: 92 μ s Page: 22.5 μ s per byte	Parallel: (@72 MHz) the access time by half-word is from 5 μ s to 150 μ s ⁽¹⁾
Write/Erase cycles	1 million Write cycles	10 kilocycles by page. Using multiple on-chip Flash memory pages is equivalent to increasing the number of write cycles. See Section 3.4: Cycling capability and page allocation .

1. For further detail, refer to [Chapter 2.5: EEPROM emulation timing](#).

2. For further detail, refer to "Memory characteristics" in STM32F3xx datasheet.

1.1 Difference in write access time

Because Flash memories have a shorter write access time, critical parameters can be stored faster in the emulated EEPROM than in an external serial EEPROM, thereby improving data storage.

1.2 Difference in erase time

The difference in erase time is the other major difference between a standalone EEPROM and an emulated EEPROM using embedded Flash memory. Unlike Flash memories, EEPROMs do not require an erase operation to free up space before writing to them. This means that some form of software management is required to store data in the Flash memory. Moreover, as the erase process of a block in the Flash memory does not take long, power shutdown and other spurious events that might interrupt the erase process (a reset, for example) should be considered when designing the Flash memory management software. To design robust Flash memory management software, a thorough understanding of the Flash memory erase process is necessary.

Note: In case of a CPU reset, ongoing page erase operations on the STM32F3xx embedded Flash are not interrupted.

1.3 Similarity in writing method

One of the similarities between external EEPROM and emulated EEPROM with the STM32F3xx embedded Flash is the writing method.

- Standalone external EEPROM: once started by the CPU, the writing of a word cannot be interrupted by a CPU reset. Only a supply failure will interrupt the write process, so properly sizing the decoupling capacitors can secure the complete writing process inside a standalone EEPROM.
- Emulated EEPROM using embedded Flash memory: once started by the CPU, the write process can be interrupted by a power failure. In case of a CPU reset, ongoing word write operations on the STM32F3xx embedded Flash are not interrupted. The EEPROM algorithm is stopped, but the current Flash word write operation is not interrupted by a software reset.

2 Implementing EEPROM emulation

2.1 Principle

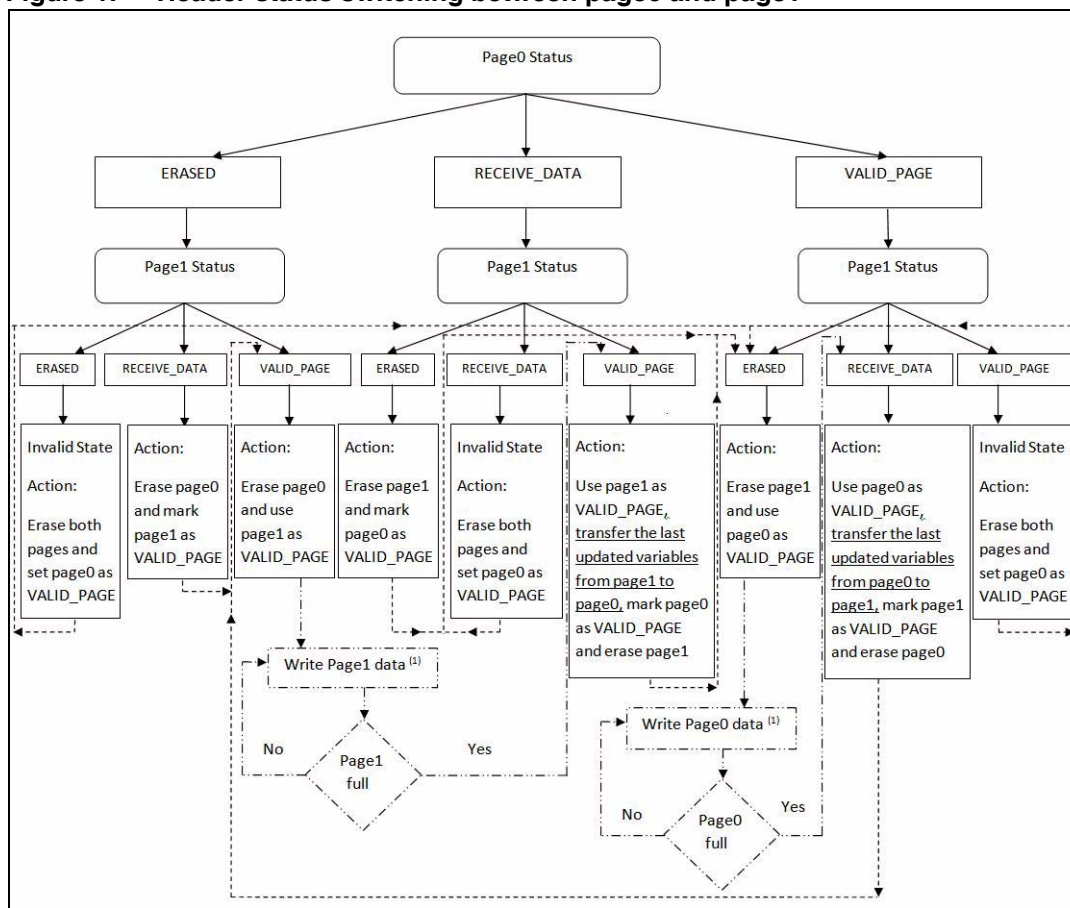
EEPROM emulation is performed in various ways, taking into consideration the Flash memory limitations and product requirements. The approach detailed below requires at least two Flash memory pages of identical size allocated to non-volatile data: one that is initially erased, the other that is ready to take over when the former page needs to be garbage-collected. A header field that occupies the first half word (16-bit) of each page indicates the page status. Each of these pages is called Page0 and Page1 in the rest of this document.

Each page has three possible states:

- **ERASED**: the page is empty.
- **RECEIVE_DATA**: the page is receiving data from the other full page.
- **VALID_PAGE**: the page contains valid data and this state does not change until all valid data is completely transferred to the erased page.

Figure 1 shows how the page status changes.

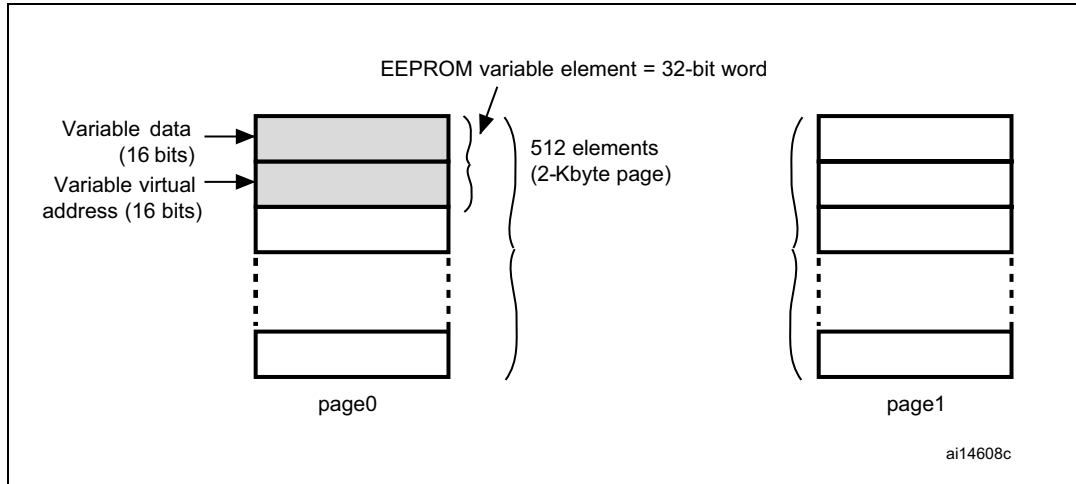
Figure 1. Header status switching between page0 and page1



1. To write data, EE_WriteVariable() function is used in the application program.

Each variable element is defined by a virtual address and a value to be stored in the Flash memory for subsequent retrieval or update (in the implemented software, both virtual address and data are 16 bits long). When data is modified, the modified data associated with the earlier virtual address is stored into a new Flash memory location. Data retrieval returns the up-to-date data value.

Figure 2. EEPROM variable format

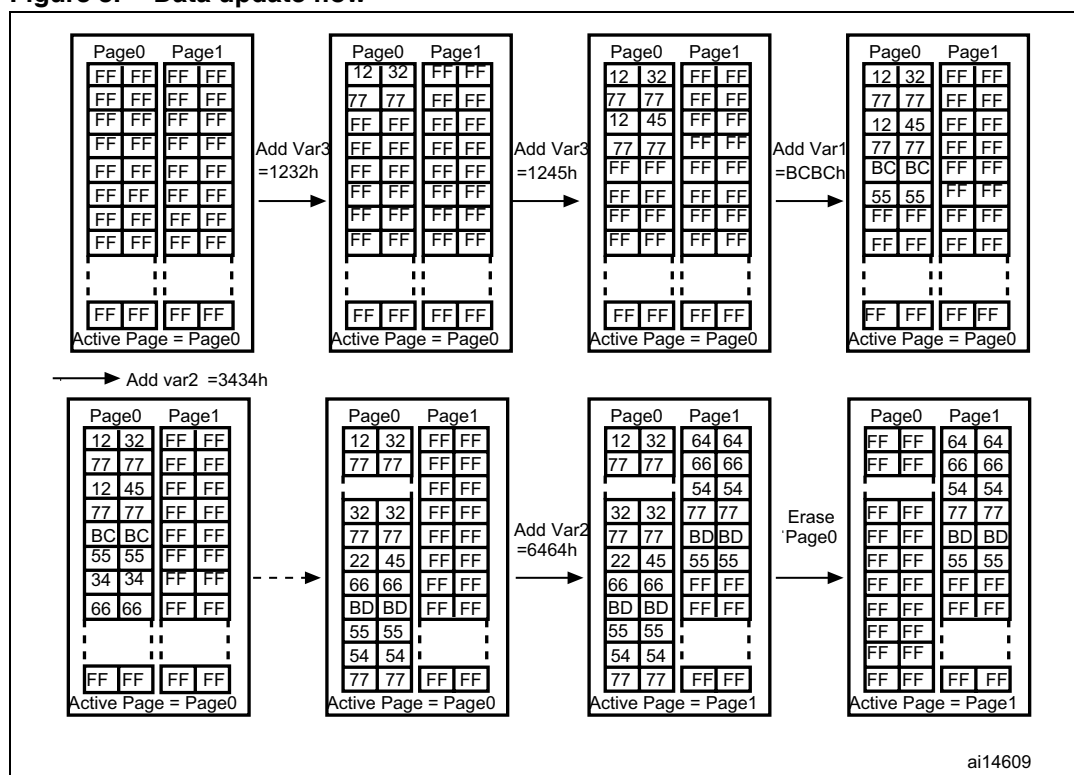


2.2 Case of use: application example

The following example shows the software management of three EEPROM variables (Var1, Var2 and Var3) with the following virtual addresses:

- Var1 virtual address 5555h
- Var2 virtual address 6666h
- Var3 virtual address 7777h

Figure 3. Data update flow



2.3 EEPROM emulation software description

This section describes the driver implemented for EEPROM emulation using the STM32F3xx Flash memory driver provided by STMicroelectronics.

A sample demonstration program is also supplied to demonstrate and test the EEPROM emulation driver using the three variables Var1, Var2 and Var3 defined in the *VirtAddVarTab[]* table declared in the software *main.c* file.

The project contains three source files in addition to the Flash memory library source files:

- ***eeeprom.c***: contains the EEPROM emulation firmware functions:

```
EE_Init()
EE_Format()
EE_FindValidPage()
EE_VerifyPageFullWriteVariable()
EE_ReadVariable()
EE_PageTransfer()
EE_WriteVariable()
```

- ***eeeprom.h***: contains the functions prototypes and some declarations. You can use this file to adapt the following parameters to your application requirements.
 - In the EEPROM emulation firmware, the `FLASH_ProgramHalfWord()` function is used to program the memory.
 - Number of data variables to be used (default: 3)

- **main.c:** this application program is an example using the described routines in order to write to and read from the EEPROM.

User API definition

The set of functions contained in the *eprom.c* file, that are used for EEPROM emulation, are described in the table below:

Table 3. API definition

Function name	Description
EE_Init()	Page header corruption is possible in the event of power loss during data update or page erase / transfer. In this case, the EE_Init() function will attempt to restore the emulated EEPROM to a known good state. This function should be called prior to accessing the emulated EEPROM after each power-down. It accepts no parameters. The process is described in Table 1: Header status switching between page0 and page1 .
EE_Format()	This function erases page0 and page1 and writes a VALID_PAGE header to page0.
EE_FindValidPage()	This function reads both page headers and returns the valid page number. The passed parameter indicates if the valid page is sought for a write or read operation (READ_FROM_VALID_PAGE or WRITE_IN_VALID_PAGE).
EE_VerifyPageFullWriteVariable()	It implements the write process that must either update or create the first instance of a variable. It consists in finding the first empty location on the active page, starting from the end, and filling it with the passed virtual address and data of the variable. In case the active page is full, the PAGE_FULL value is returned. This routine uses the parameters below: Virtual address: may be any of the three declared variables' virtual addresses (Var1, Var2 or Var3) Data: the value of the variable to be stored This function returns FLASH_COMPLETE on success, PAGE_FULL if there is not enough memory for a variable update, or a Flash memory error code to indicate an operation failure (erase or program).
EE_ReadVariable()	This function returns the data corresponding to the virtual address passed as a parameter. Only the last update is read. The function enters in a loop in which it reads the variable entries until the last one. If no occurrence of the variable is found, the ReadStatus variable is returned with the value "1", otherwise it is reset to indicate that the variable has been found and the variable value is returned on the Read_data variable.
EE_PageTransfer()	It transfers the latest value of all variables (data with associated virtual address) from the current page to the new active page. At the beginning, it determines the active page, which is the page the data is to be transferred from. The new page header field is defined and written (new page status is RECEIVE_DATA given that it is in the process of receiving data). When the data transfer is complete, the new page header is VALID_PAGE, the old page is erased and its header becomes ERASED.
EE_WriteVariable()	This function is called by the user application to update a variable. It uses the EE_VerifyPageFullWriteVariable() and EE_PageTransfer() routines that have already been described.

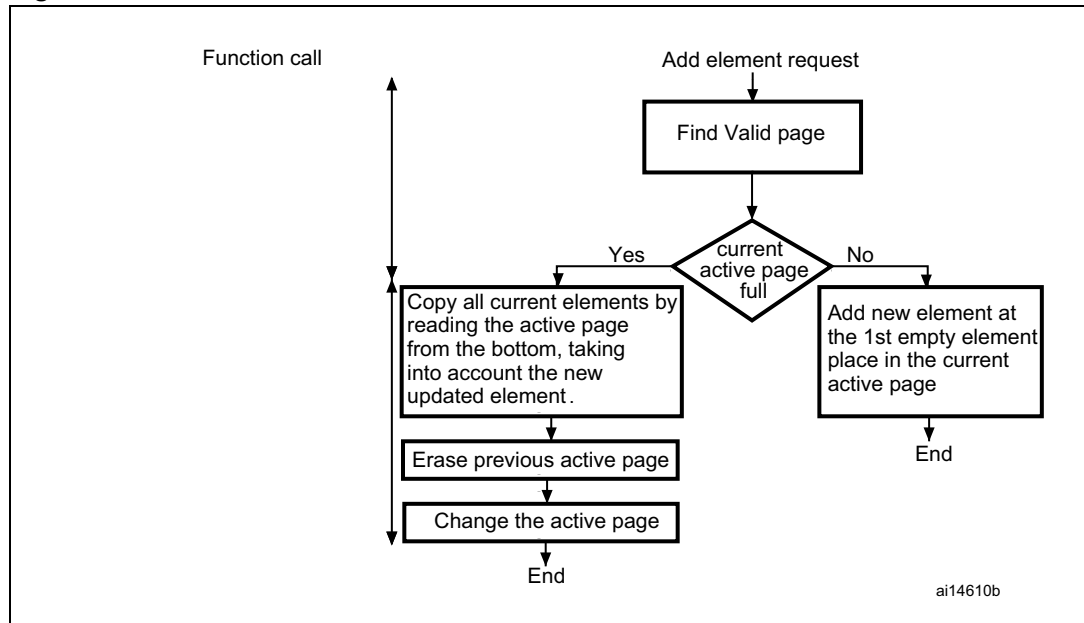
Note: The following functions can be used to access the emulated EEPROM:

- EE_Init()
- EE_ReadVariable()
- EE_WriteVariable()

These functions are used in the application code delivered with this application note.

Figure 4 shows the procedure for updating a variable entry in the EEPROM.

Figure 4. WriteVariable flowchart



Key features

- User-configured emulated EEPROM size
- Increased Flash memory endurance: page erased only when it is full
- Non-volatile data variables can be updated infrequently
- Interrupt servicing during program/erase is possible

2.4 EEPROM emulation memory footprint

Table 4 details the footprint of the EEPROM emulation driver in terms of Flash size and RAM size.

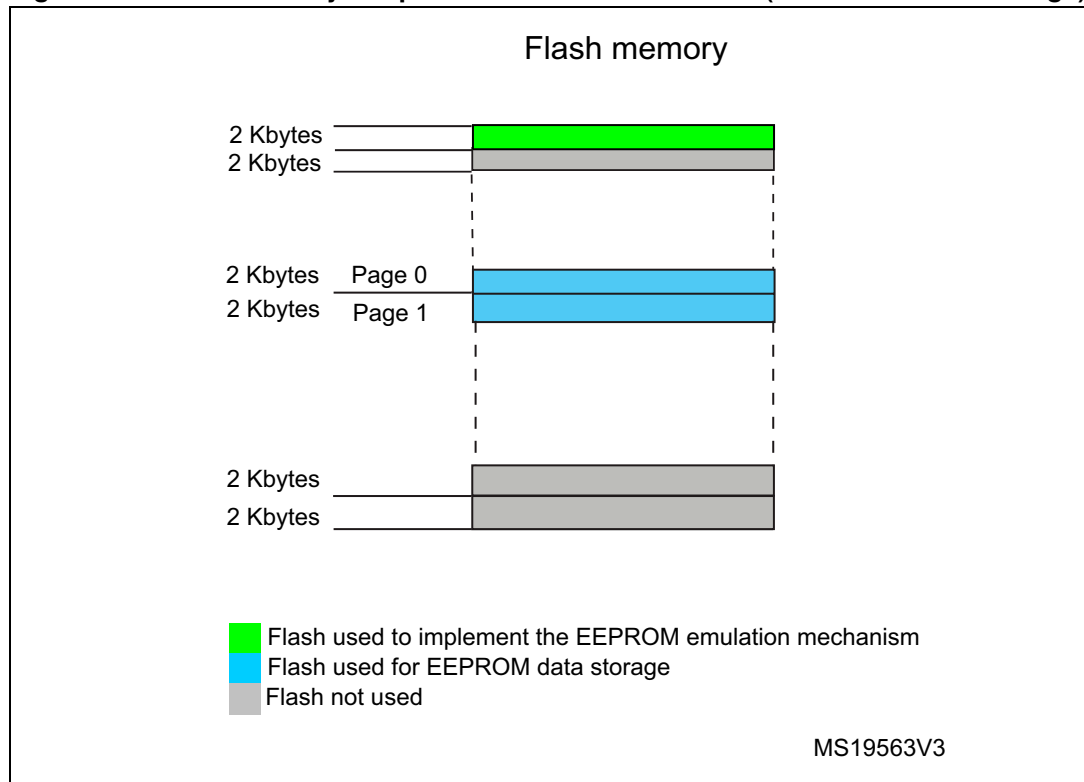
The table and figure below have been determined using the IAR EWARM 6.30 tool with High Size optimization level.

Table 4. Memory footprint for EEPROM emulation mechanism

Mechanism	Minimum ⁽¹⁾ required code size (bytes)	
	Flash	SRAM
EEPROM emulation software mechanism	2260	2074

1. Based on one 32-bit variable (16-bit for address and 16-bit for data). The SRAM memory used increases depending on the number of variables used.

Figure 5. Flash memory footprint for EEPROM emulation (mechanism and storage)



2.5 EEPROM emulation timing

This section describes the timing parameters associated with the EEPROM emulation driver based on two 2-Kbyte EEPROM page sizes.

All timing measurements are performed:

- STM32F373VCT6 Revision A
- System clock at 72 MHz, Flash prefetch disabled and cache features enabled
- With execution from Flash
- At room temperature

[Table 5](#) lists the timing values for EEPROM.

Table 5. EEPROM emulation timings with a 72 MHz system clock

Operation	EEPROM emulation timings		
	Minimum	Typical	Maximum
Typical ⁽¹⁾ write operation of a variable ⁽²⁾ in EEPROM	142 μ s	-	290 μ s
Write operation of a variable with page swap ⁽³⁾ in EEPROM	-	41 ms	-
Read operation of a variable from EEPROM ⁽⁴⁾	5 μ s	-	150 μ s
EEPROM Initialization for the 1st time ⁽⁵⁾	-	80 ms	-
Typical EEPROM Initialization ⁽⁶⁾	-	40 ms	-

1. Write with no page swap. The minimum value refers to a write operation of a variable in the beginning of the Flash page and the maximum value refers to a write operation in the end of the Flash page. The difference between the minimum and maximum values is due to the time taken to find a free Flash address to store the new data.
2. The variable size used is 32-bit (16-bit for the virtual address and 16-bit for the data).
3. The page swapping is done when the valid page is full. It consists of transferring the last data stored for each variable to the other free page and erasing the full page.
4. The minimum value refers to a read operation of the last variable stored in the Flash page and the maximum value refers to a read operation of the first variable. The difference between the minimum and maximum values is due to the time taken to find the last variable data stored.
5. When the EEPROM mechanism is run for the 1st time or for an invalid status (refer to [Table 1: Header status switching between page0 and page1](#) for more detail), the two pages are erased and the page used for storage is marked as VALID_PAGE.
6. A typical EEPROM initialization is performed when a valid page exists (the EEPROM has been initialized at least once). During a typical EEPROM initialization, one of the two pages is erased (see [Table 1: Header status switching between page0 and page1](#) for more details).

3 Embedded application aspects

This section provides advice on how to overcome software limitations in embedded applications and how to fulfill the needs of different applications.

3.1 Data granularity management

Emulated EEPROM can be used in embedded applications where non-volatile storage of data updated with a half-word or word granularity is required. It generally depends on the user requirements and Flash memory architecture (for example, stored data length, write access).

The STM32F3xx on-chip Flash memory allows 16-bit or word programming used as described in [Table 6](#).

Table 6. Flash program functions

Data granularity	Function name
by word (32-bit)	FLASH_ProgramWord
by half-word (16-bit)	FLASH_ProgramHalfWord

3.2 Wear-leveling: Flash memory endurance improvement

In the STM32F3xx on-chip Flash memory, each page can be programmed or erased reliably around 10,000 times.

For write-intensive applications that use more than two pages for the emulated EEPROM, it is recommended to implement a wear-leveling algorithm to monitor and distribute the number of write cycles among the pages.

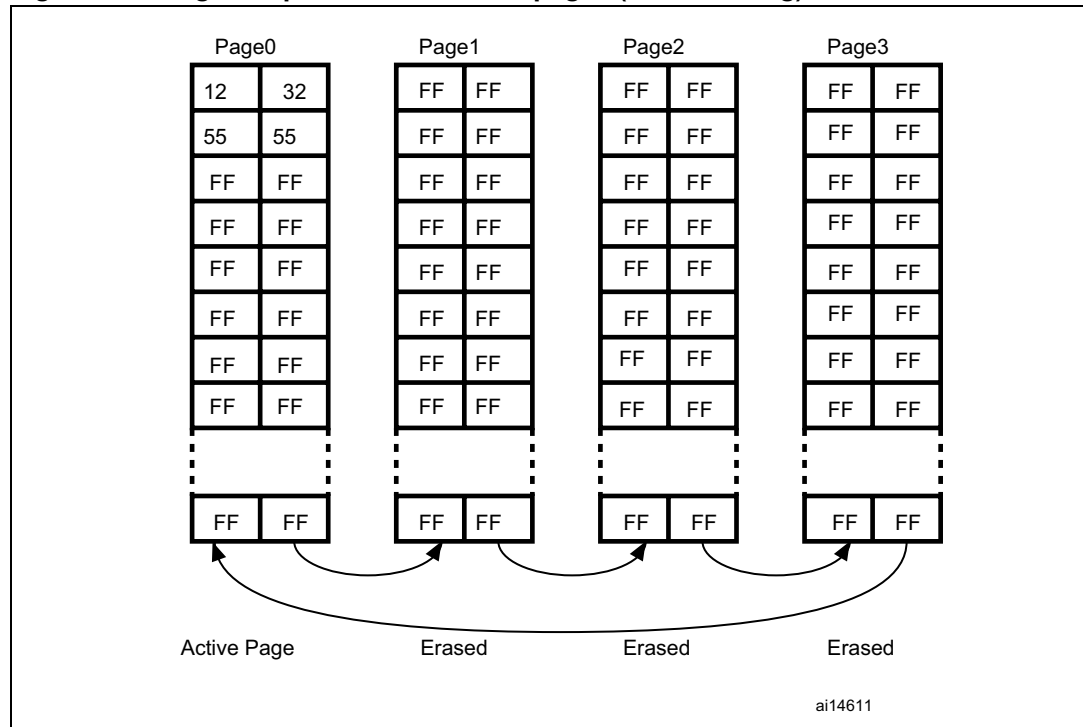
When no wear-leveling algorithm is used, the pages are not used at the same rate. Pages with long-lived data do not endure as many write cycles as pages that contain frequently updated data. The wear-leveling algorithm ensures that equal use is made of all the available write cycles for each page.

3.2.1 Wear-leveling implementation example

In this example, in order to enhance the emulated EEPROM capacity, four pages will be used (Page0, Page1, Page2 and Page3).

The wear-leveling algorithm is implemented as follows: when page n is full, the device switches to page $n+1$. Page n is garbage-collected and then erased. When it is the turn of Page3 to be full, the device goes back to Page0, Page3 is garbage-collected then erased and so on (refer to [Figure 6](#)).

Figure 6. Page swap scheme with four pages (wear-leveling)



In the software, the wear-leveling algorithm can be implemented using the `EE_FindValidPage()` function.

3.3 Page header recovery in case of power loss

Data or page header corruption is possible in case of a power loss during a variable update, page erase or transfer.

To detect this corruption and recover from it, the `EE_Init()` routine is implemented. It should be called immediately after power-up. The principle of the routine is described in this application note. The routine uses the page status to check for integrity and perform repair if necessary.

After power loss, the `EE_Init()` routine is used to check the page header status. There are 9 possible status combinations, three of which are invalid. [Table 1: Header status switching between page0 and page1](#) shows the actions that should be taken based on the page statuses upon power-up.

3.4 Cycling capability and page allocation

3.4.1 Cycling capability

A program/erase cycle consists of one or more write accesses and one page erase operation.

When the EEPROM technology is used, each byte can be programmed and erased a finite number of times, typically in the range of 10 000 to 100 000.

However, in an embedded Flash memory, the minimum erase size is the page and the number of program/erase cycles applied to a page is the number of possible erase cycles. The STM32F3xx electrical characteristics guarantee 10 000 program/erase cycles per page. The maximum lifetime of the emulated EEPROM is thereby limited by the update rate of the most frequently written parameter.

The cycling capability depends on the amount/size of data that the user wants to handle.

In this example, two pages (of 2 Kbytes) are used and programmed with 16-bit data. Each variable corresponds with a 16-bit virtual address. That is, each variable occupies a word of storage space. A page can store 2 Kbytes multiplied by the Flash memory endurance of 10 000 cycles, giving a total of 20 000 Kbytes of data storage capacity for the lifetime of one page in the emulated EEPROM memory. Consequently, 40 000 Kbytes can be stored in the emulated EEPROM, provided that two pages are used in the emulation process. If more than two pages are used, this number is multiplied accordingly.

Knowing the data width of a stored variable, it is possible to calculate the total number of variables that can be stored in the emulated EEPROM area during its lifetime.

3.4.2 Flash page allocation

The page size and the page number needed for an EEPROM emulation application can be chosen according to the amount of data written during the lifetime of the system.

For example, a 2-Kbyte page with 10-Kbyte erase cycles can be used to write a maximum of 20 megabytes during the lifetime of the system.

Free variable space can be calculated as:

$$\text{FreeVarSpace} = (\text{PageSize}) / (\text{VariableTotalSize}) - [\text{NbVar} + 1]$$

Where:

- **Page size:** page size in bytes (for example, 2 Kbytes)
- **NbVar:** number of variables in use (for example, 10 variables)
- **VariableTotalSize:** number of bytes used to store a variable (address and data)
 - VariableTotalSize = 8 for 32-bit variables with (32-bit data + 32-bit virtual address)
 - VariableTotalSize = 4 for 16-bit variables with (16-bit data + 16-bit virtual address)

An estimation of the pages needed to guarantee predictable Flash operation during the application lifetime can be calculated as following:

Required page number:

$$\text{NbPages} = \text{NbWrites} / (\text{PageEraseCycles} * \text{FreeVarSpace})$$

Where:

- **NbPages** = number of pages needed
- **NbWrites** = total number of variable writes
- **PageEraseCycles** = number of erase cycles of a page

- Note:*
- 1 *If variables are 16-bit, each variable takes 32 bits (16-bit data with a 16-bit virtual address), which means that each variable uses 4 bytes of Flash memory each time new data is written. Each 2-Kbyte page can take 512 variable writes before it is full.*
 - 2 *This calculation is a slightly conservative estimation.*

Case of use example

To design an application that updates **20** different variables, every **2** minutes for **10** years:

- NbVar = **20**
- NbWrites = $10 * 365 * 24 * (60/2) * \text{NbVar} = \sim 52$ million writes
- If variables are 16-bit data, with a 16-bit virtual address, the number of pages needed to guarantee non-volatile storage of all this data is two pages of 16 Kbytes.
- If variables are 32-bit data with 32-bit virtual address, the number of pages needed to guarantee non-volatile storage of all this data is three pages of 16 Kbytes.

Table 7. Application design

Variable size	Number of writes (NbWrites)	Total amount of data to write (bytes)	Page size (Kbytes)	Page erase cycles	Number of pages needed	Number of pages used (1)
16-bit	52 560 000	210 240 000	2	10000	10.5	11
32-bit	52 560 000	420 480 000	2	10000	21.1	22

1. A minimum of two pages is needed to emulate the EEPROM

3.5 Real-time consideration

The provided implementation of the EEPROM emulation firmware runs from the internal Flash, thus access to the Flash memory will be stalled during operations requiring Flash erase or programming (EEPROM initialization, variable update or page erase). As a consequence, the application code is not executed and the interrupt cannot be serviced.

This behavior may be acceptable for many applications, however, for applications with real-time constraints, you need to run the critical processes from the internal RAM.

In this case:

1. Relocate the vector table in the internal RAM.
2. Execute all critical code and interrupt service routines from the internal RAM. The compiler provides a keyword to declare functions as a RAM function; the function is copied from the Flash to the RAM at system startup just like any initialized variable. It is important to note that, for a RAM function, all used variable(s) and called function(s) should be within the RAM.

4 Revision history

Table 8. Document revision history

Date	Revision	Changes
24-Sep-2012	1	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY TWO AUTHORIZED ST REPRESENTATIVES, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2012 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com

