AN4061
Application note

EEPROM emulation in STM32F0xx 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, an external EEPROM can be replaced by an on-chip Flash, with a specific software algorithm.

This application note describes the software solution for substituting a standalone EEPROM by emulating the EEPROM mechanism using the on-chip Flash of STM32F0xx devices.

The 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

Contents

List of tables

List of figures

1 Main differences between external and emulated EEPROM

The 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 STM32F0xx Flash memory may be used instead of an external EEPROM for simultaneous code and data storage.

Unlike Flash memory, however, the 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 an 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 an embedded Flash memory and an external serial EEPROM are the same for any microcontroller that uses the same Flash memory technology (it is not specific to the STM32F0xx family products). The major differences are summarized in *Table 1*.

Table 1. Differences between external and emulated EEPROM

Feature	External EEPROM (for example, M24C64: I²C 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 µs	Half-word program time: from 124 µs to 26 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 software 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 µs – Random word: 92 µs – Page: 22.5 µs per byte	Parallel: (at 48 MHz) the access time by half-word is from 3.8 µs to 110 µs (2)
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 STM32F051xx 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 a 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 software reset, ongoing page erase or mass erase operations on the STM32F0xx embedded Flash are not interrupted.

1.3 Similarity in writing method

One of the similarities between external EEPROM and emulated EEPROM with the STM32F0xx embedded Flash is the writing method.

- Standalone external EEPROM: once started by the CPU, the writing of a word cannot be interrupted by a software 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 software reset, ongoing word write operation on the STM32F0xx embedded Flash is 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

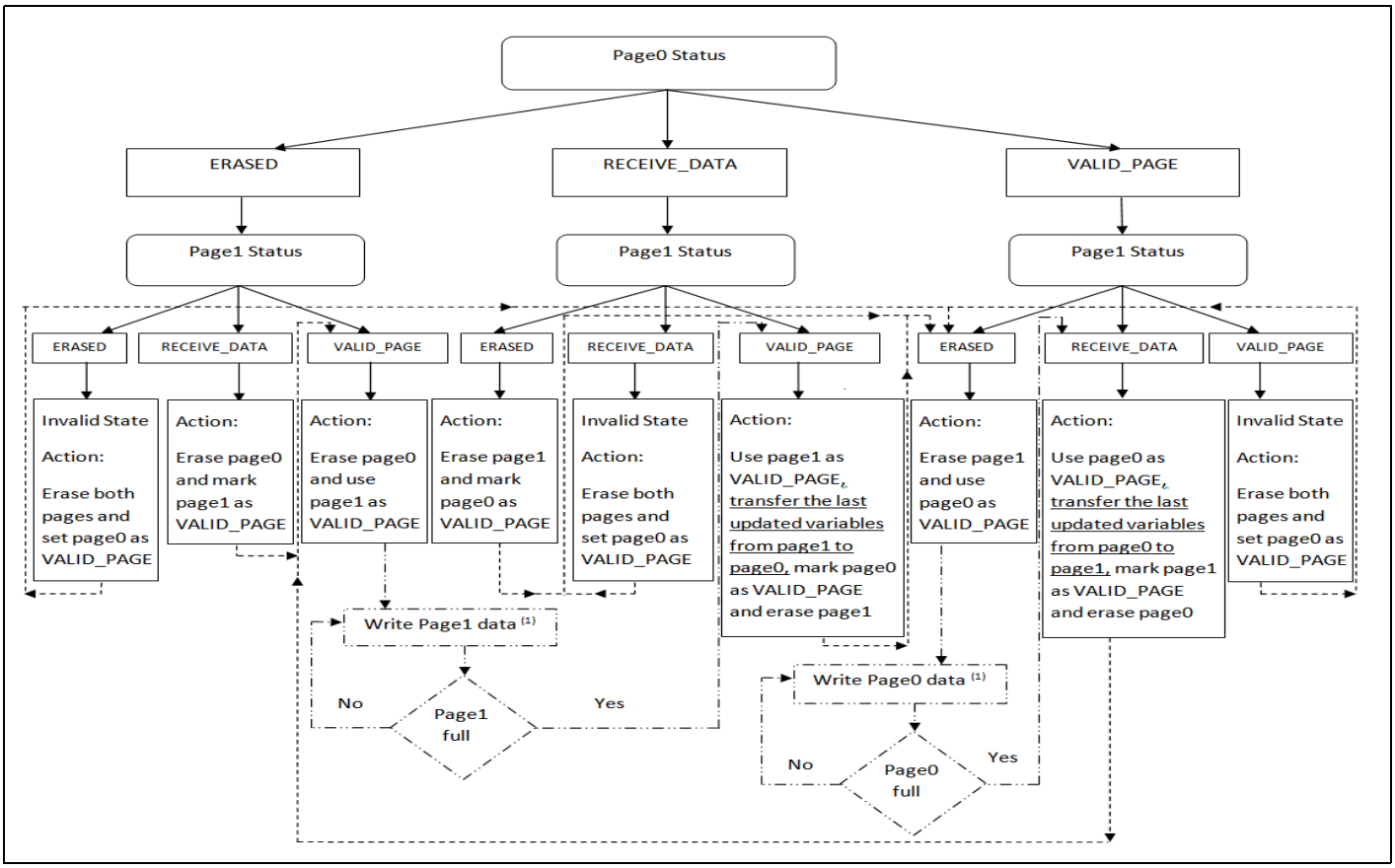
EEPROM emulation is performed in various ways, taking into consideration the Flash memory limitations and product requirements. The approach detailed below requires at least two Flash memory pages of an identical size allocated to non-volatile data: one that is initially erased, the other that is ready 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 t.he 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.

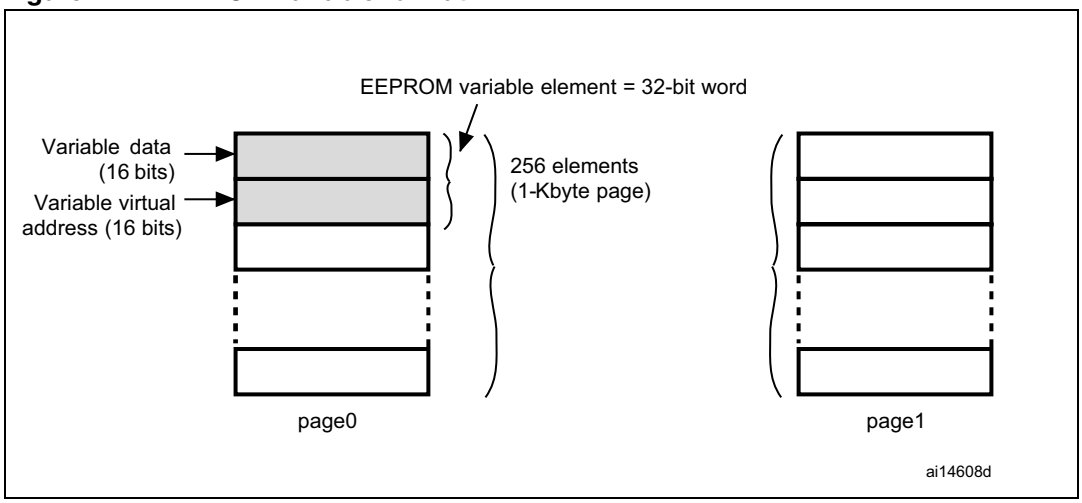
Figure 1 shows how the page status changes.

Figure 1. Header status switching between page0 and page1



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

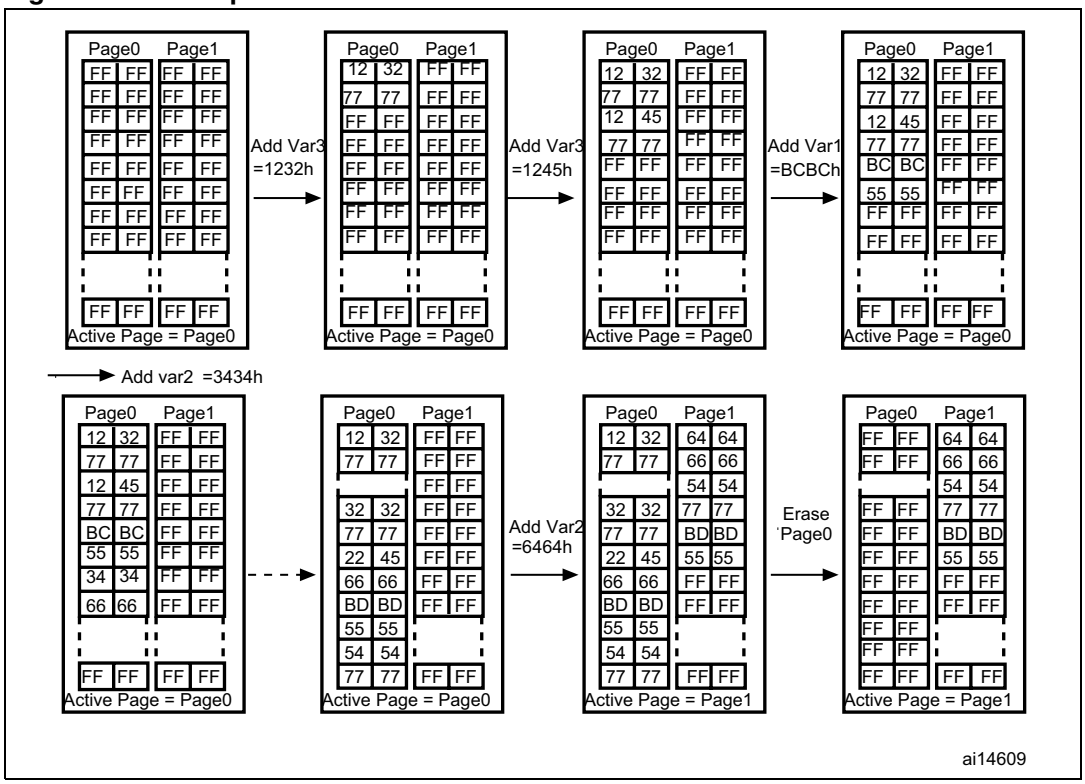
The following example shows the software management of three EEPROM variables (Var1, Var2 and Var3) with the following virtual addresses:

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 the EEPROM emulation using the STM32F0xx 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:

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

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

- ***eeprom.h***: contains the function prototypes and some declarations. You can use this file to adapt the following parameters to your application requirements:
  - 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 *eeprom.c* file, that are used for the EEPROM emulation, are described in the table below:

Table 2. 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.
`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()

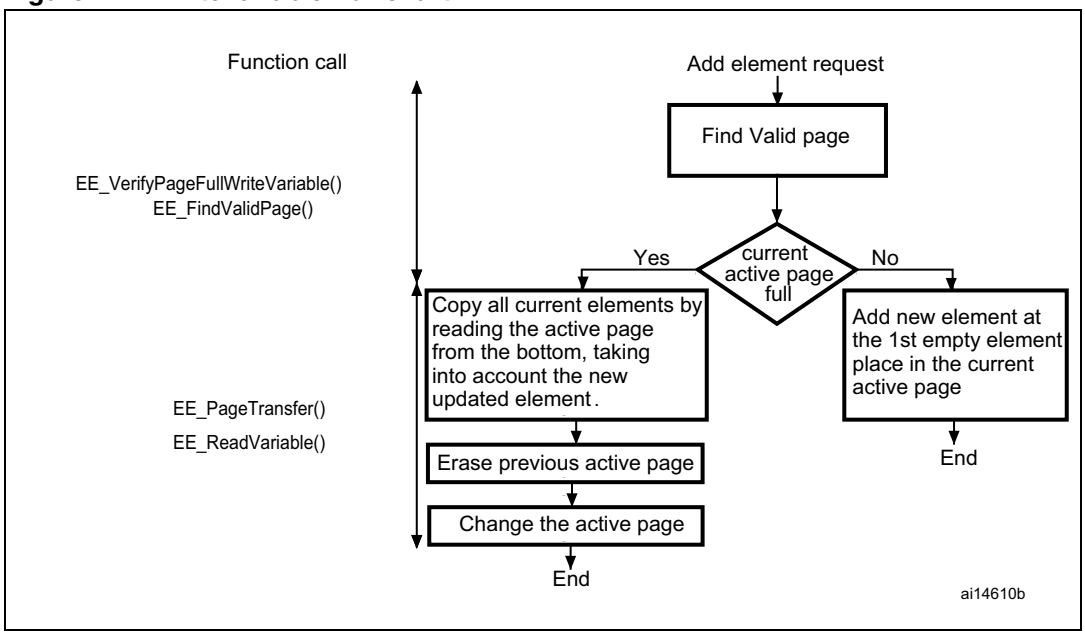
- EE_ReadVariable()

- 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 3 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 v6.30.7 tool with High Size optimization level.

Table 3. Memory footprint for EEPROM emulation mechanism

Mechanism	Minimum(1) required code size (bytes)	
	Flash	SRAM
EEPROM emulation software mechanism	1824	1046

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

2.5 EEPROM emulation timing

This section describes the timing parameters associated with the EEPROM emulation driver based on two 16-Kbyte EEPROM page sizes.

All timing measurements are performed:

- STM32F051RBT6
- System clock at 48 MHz, Flash prefetch and cache features enabled
- With execution from Flash
- At room temperature

Table 4 lists the timing values for EEPROM.

Table 4. EEPROM emulation timings with a 48 MHz system clock

Operation	EEPROM emulation timings		
	Minimum (µs)	Typical (ms)	Maximum (µs)
Typical (1) variable(2) Write operation in EEPROM	124	-	219
Variable Write operation with page swap (3) in EEPROM	-	26	-
Variable Read Operation from EEPROM (4)	3.8	-	110
EEPROM Initialization for the 1st time (5)	-	52.11	-
Typical EEPROM Initialization(6)	-	26	-

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. Page swapping is done when the valid page is full. It consists of transferring the last stored data for each variable to the other free page and erasing the full page.
4. The minimum value refers to a read operation of the 1st variable stored in the Flash page and the maximum value refers to a read operation of the last variable -1. 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 (see *Table 1: Header status switching between page0 and page1* for more details), 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

An 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 STM32F0xx on-chip Flash memory allows 16-bit or word programming:

Table 5. 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 STM32F0xx 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.

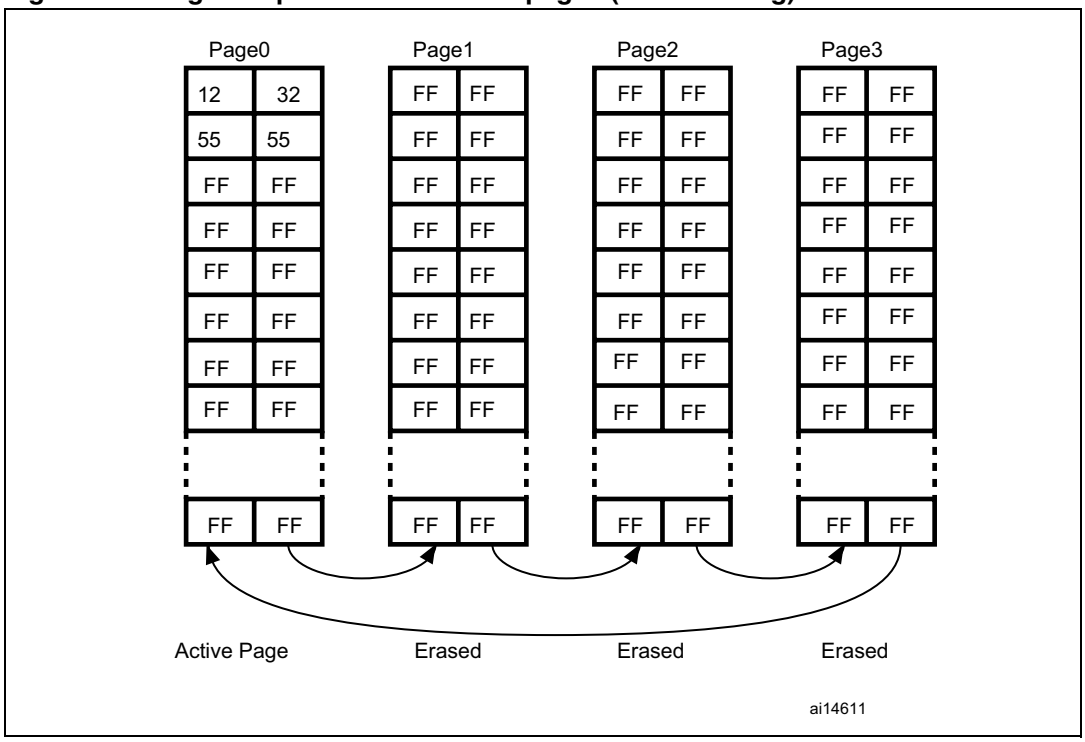
Note: The main memory block in the STM32F0xx Flash memory is divided as described in Table 3: Memory footprint for EEPROM emulation mechanism.

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 5*).

Figure 5. Page swap scheme with four pages (wear-leveling)



In the software, the wear-leveling algorithm can be implemented using the `EE_FindValidPage()` function (refer to *Table 2*).

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. *Figure 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 STM32F0xx's 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 1 Kbyte) are used and programmed with 16-bit data. Each variable corresponds to a 16-bit virtual address. That is, each variable occupies a word of storage space. A page can store 1 Kbyte multiplied by the Flash memory endurance of 10 000 cycles, giving a total of 10 000 Kbytes of data storage capacity for the lifetime of one page in the emulated EEPROM memory. Consequently, 20 000 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 1-Kbyte page with 10-Kbyte erase cycles can be used to write a maximum of 10 megabytes during the lifetime of the system.

Free variable space can be calculated as:

FreeVarSpace = (PageSize) / (VariableTotalSize) - [NbVar+1]

Where:

- **Page size**: page size in bytes (for example, 1 Kbyte)
- **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 follows:

Required page number:

NbPages = NbWrites / (PageEraseCycles * FreeVarSpace)

Where:

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

Note: If variables are 16-bit, each variable takes 32-bit (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 1-Kbyte page can take 256 variable writes before it is full.

Note: 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**) * NbVar = ~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:
  - Twenty one pages of 1 Kbyte
- 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:
  - Forty one pages of 1 Kbyte

Table 6. Application design

Variable size	Number of writes (NbWrites)	Total amount of data to write (in bytes)	Page size (Kbytes)	Page erase cycles	Number of pages needed	Number of pages used
16-bit	52 560 000	210 240 000	1	10000	20.5	21
32-bit	52 560 000	420 480 000	1	10000	41	41

3.5 Real-time consideration

The provided implementation of the EEPROM emulation firmware runs from the internal Flash, thus the access to the Flash 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 7. Document revision history

Date	Revision	Changes
11-May-2012	1	Initial release.