AN4894
Application note

EEPROM emulation techniques and software for STM32 microcontrollers

Introduction

EEPROMs (electrically erasable programmable read-only memories) are used for non-volatile storage of updatable application data, or to retain small amounts of data in the event of power failure in complex systems. To reduce cost, an external EEPROM can be replaced by on-chip Flash memory, provided that a specific software algorithm is used.

This application note describes the software solution (X-CUBE-EEPROM) for substituting a standalone EEPROM by emulating the EEPROM mechanism using the on-chip Flash memory available on STM32 Series listed in *Table 1: Applicable products*. X-CUBE-EEPROM also provides a firmware package including examples showing how to exploit this EEPROM emulation driver (see *Section 5: API and application examples*).

For STM32WB Series products only, an example that maintains a Bluetooth® Low-Energy connection and communication while performing EEPROM operations is provided.

The emulation method uses at least two Flash memory pages, between which the EEPROM emulation code swaps data as they become filled. This is transparent to the user. The EEPROM emulation driver supplied with this application note has the following features:

- Lightweight implementation and reduced footprint
- Simple API consisting of a few functions to format, initialize, read and write data, and clean up Flash memory pages
- At least two Flash memory pages to be used for internal data management
- Clean-up simplified for the user (background page erase)
- Wear-leveling algorithm to increase emulated EEPROM cycling capability
- Robust against asynchronous resets and power failures
- Optional protection implementation for Flash-memory sharing between cores in multi-core STM32 devices (for example STM32WB Series).

The EEPROM size to be emulated is flexible and only limited by the Flash memory size allocated to that purpose.

Table 1. Applicable products

Type	Series
Microcontrollers	STM32L4 Series, STM32L4+ Series, STM332L5 Series, STM32G0 Series, STM32G4 Series, STM32WB Series

Contents

List of tables

List of figures

1 General information

This document scopes STM32 microcontrollers that are based on an Arm®(a) core.

1.1 Reference documents

EEPROM emulation solutions and application notes are available for other STM32 Series as listed in reference *[1]* below.

Table 2. Related documents

Reference	Document name
[1]	Application notes: – STM32F0 Series: EEPROM emulation in STM32F0xx microcontrollers (AN4061) – STM32F1 Series: EEPROM emulation in STM32F10x microcontrollers (AN2594) – STM32F2 Series: EEPROM emulation in STM32F2xx microcontrollers (AN3390) – STM32F3 Series: EEPROM emulation in STM32F3xx microcontrollers (AN4046) / EEPROM emulation in STM32F30x/STM32F31x STM32F37x/STM32F38x microcontrollers (AN4056) – STM32F4 Series: EEPROM emulation in STM32F40x/STM32F41x microcontrollers (AN3969)
[2]	Building wireless applications with STM32WB Series microcontrollers, application note (AN5289)

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 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, half-word, or word granularity during run time. However, microcontrollers used in these systems are very often based on embedded Flash memory. To eliminate components, save PCB space and reduce system cost, the STM32 Flash memory may be used instead of external EEPROM to store not only code, but also data.

Special software management is required to store data in embedded Flash memory. The EEPROM emulation software scheme depends on many factors, including the required EEPROM reliability, the architecture of the Flash memory used, and the final product requirements, among other parameters.

The main differences between embedded Flash memory and external serial EEPROM are the same for any microcontroller that uses Flash memory technology (they are not specific to STM32 Series microcontrollers). One difference is that EEPROMs do not require an erase operation to free up space before data can be written again. Other major differences are summarized in *Table 3*.

Table 3. 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 in 4 ms. Word program time = 16 ms Page (32 bytes) Write 4 ms. Consecutive Words program time = 500 µs	Word program time: from 90 µs to 580 ms (1)
Erase time	N/A	2 Kbytes page-erase time: for instance, 22 ms(2)
Memory Size	From a few Kbytes to 256 Kbytes	Only limited by the size of Flash memory allowed for EEPROM emulation.
Read access	Serial: a hundred µs Random word: 92 µs Page: 22.5 µs per byte	Parallel: the access time is from 6 µs to 592 µs(1)
Endurance	4 million cycles at 25 °C 1.2 million cycles at 85 °C 600 kilocycles at 125 °C	10 kcycles per page @ 105 °C(2). Using multiple on-chip Flash memory pages is equivalent to increasing the number of write cycles. See *Section 4.4: Cycling capability: EEPROM endurance improvement*.
Retention(2)	50 years at 125 °C 100 years at 25 °C	7 years @ 125 °C 15 years @ 105 °C 30 years @ 85 °C

1. For further details, refer to *Chapter 5.3: EEPROM emulation timing*.
2. Example data for STM32L4 Series. Refer to the datasheet of your STM32 product.

2.1 Difference in write access time

Flash memories have a shorter write access time allowing critical parameters to be stored faster in the emulated EEPROM than in an external EEPROM in most cases. However, due to the data transfer mechanism, the emulated EEPROM write access time sometimes becomes significantly higher.

2.2 Programming and erase operations

Unlike Flash memories, EEPROMs do not require an erase operation to free up space before writing to a programmed address. This is a major difference between a standalone EEPROM and emulated EEPROM using embedded Flash memory.

- Emulated EEPROM using embedded Flash memory

  The Erase process management is fully handled by the EEPROM emulation software, but the Erase operation is left to application software management. This allows a reduction of the worst case write time, and also Flash Page Erase operations when the application execution time becomes less critical.

  Moreover, as the Flash memory programming and erase operations are quite long, power failures and other spurious events that might interrupt the erase process (such as resets) have been considered when designing the Flash memory management software. The EEPROM emulation software has been designed to be robust against power failures and fully asynchronous resets.

- Standalone external EEPROM

  Once started by the CPU, the writing of a word cannot be interrupted by a CPU reset. Only a supply failure may interrupt the write process, so power supply monitoring and properly sized decoupling capacitors are necessary to secure the complete writing process inside a standalone EEPROM.

3 Implementing EEPROM emulation

3.1 Principle

EEPROM emulation can be performed in various ways, taking into consideration the Flash memory characteristics and final product requirements. The approach detailed below requires two sets of Flash memory pages allocated to non-volatile data.

The first set of pages is initially erased and used to store new data and Flash programming operations are done sequentially in increasing order of Flash addresses. Once the first set of pages is full of data, it needs to be garbage-collected.

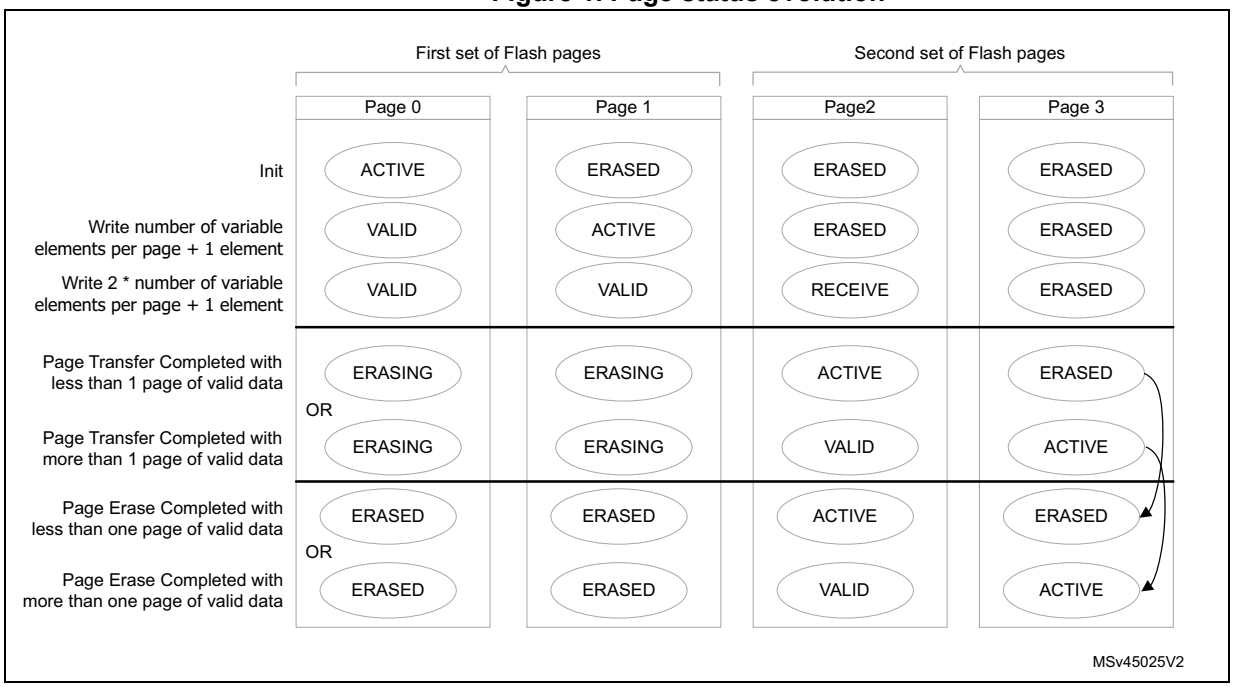
The second set of pages collects only the valid data from the first set of pages and the remaining area can be used to store new data. Once the transfer of valid data to the second set of pages is completed, the first set of pages can be erased.

Each set of pages can be made up of one or several Flash pages. A header field that occupies the first four 64-bit words (32 bytes) of each page indicates its status. Each page has five possible states:

- **ERASED**: the page is empty (initial state)
- **RECEIVE**: the page used during data transfer to receive data from other full pages.
- **ACTIVE**: the page is used to store new data
- **VALID**: the page is full. This state does not change until all valid data is completely transferred to the receiving page.
- **ERASING**: valid data in this page has been transferred. The page is ready to be erased.

Figure 1 shows the page status evolution in the case where each set of pages is made of two pages.

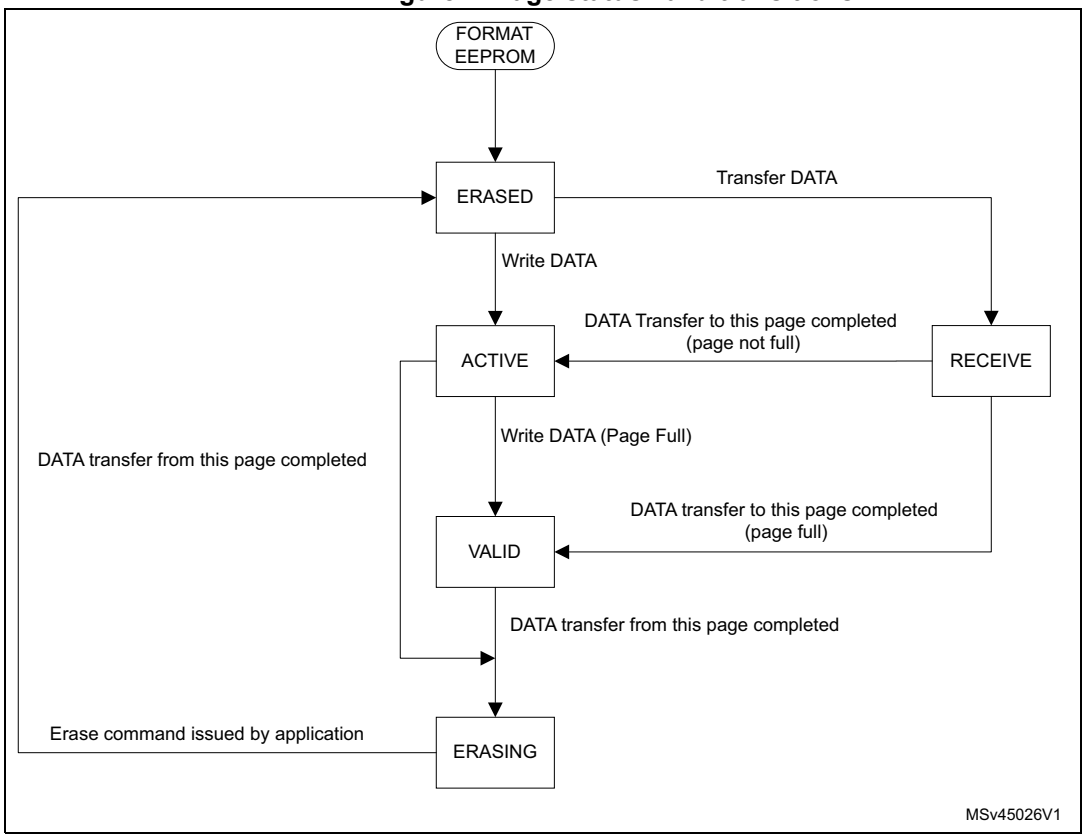
Figure 1. Page status evolution



3.2 Page status valid transitions

Information provided in this paragraph is only useful for users intending to modify the EEPROM emulation driver. It is not useful for a simple utilization of the driver.

Figure 2. Page status valid transitions



3.3 Page and variable format

Depending on the STM32 Series, Flash memory page sizes and composition can vary.

Moreover, for some STM32 Series, the EEPROM emulation driver supports either single bank mode, dual bank mode, or both. The right mode must be selected in the EEPROM emulation driver, according to the mode supported. See *Table 4* for this information.

Table 4. Flash memory properties

STM32 Series/Value Lines	Flash memory page sizes	Single bank mode support	Dual bank mode support
STM32L4+ Series	4 Kbytes (512 words of 64 bits for DBANK=1)	No(1)	Yes
STM32L41x/42x/43x 44x/45x/46x	2 Kbytes (256 words of 64 bits)	Yes	No(2)
STM32L47x/48x/49x/4Ax	2 Kbytes (256 words of 64 bits)	Yes	Yes
STM32L552 STM32L562	2 Kbytes (256 words of 64 bits) (for DBANK=1)	No (1)	Yes
STM32G0x0 STM32G0x1	2 Kbytes (256 words of 64 bits)	Yes	No(2)
STM32G4xx Series	2 Kbytes (256 words of 64 bits) (for DBANK=1)	No(1)	Yes
STM32WBxx Series	4 Kbyte (512 words of 64 bits)	Yes	No(2)

1. When in single bank mode, these devices operate 128 bits wide read accesses. However, the EEPROM emulation solution is designed for 64 bits wide read accesses.
2. These devices do not support the dual-bank feature.

The minimal write width in Flash memory is 64-bits due to its ECC (Error Correcting Code) that cannot be switched off; only zero (0x0000000000000000) can be written to an already programmed non-null Flash line. As the first four words are used by the header, a Flash page can store up to 252 variable elements when the page size is 2 Kbytes, and up to 508 variable elements when the page size is 4 Kbytes.

The possible states of a Flash page are coded by writing 0xAAAAAAAAAAAAAAAA into the page header. It is possible to determine the page state using the following procedure:

- The page is in ERASING state if its fourth line is not erased
- The page is in VALID state if the third line is not erased and the fourth line is erased
- The page is in ACTIVE state if the second line is not erased and the third and fourth lines are erased
- The page is in RECEIVE state if the first line is not erased and the second, third and fourth lines are erased
- The page is in ERASED state if the first four lines are erased.

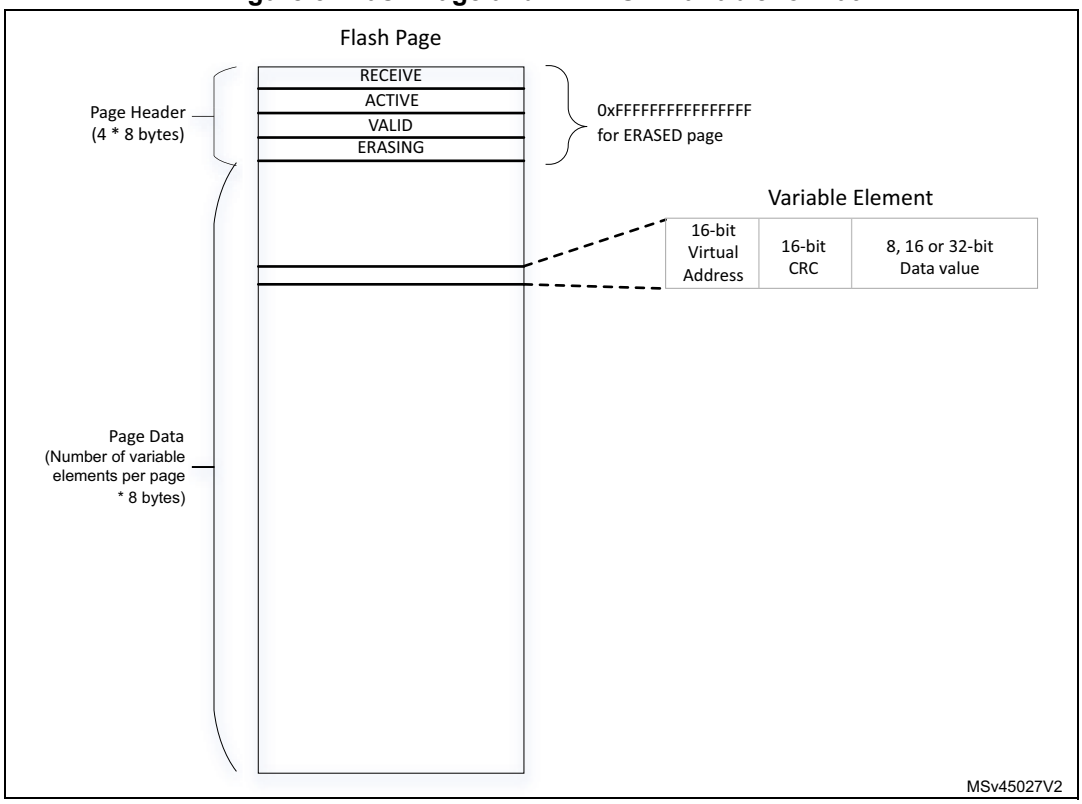
This algorithm allows the coding of all states and transitions described in *Section 3.2: Page status valid transitions*.

Each variable element is defined by a virtual address and a data value to be stored in Flash memory for subsequent retrieval or update. In the implemented software, the virtual address is 16 bits long and the data value is either 8 bits, 16 bits or 32 bits long. The driver requires

the virtual address values to be between 0x0001 (0x0000 corresponds to an EEPROM element invalidated by the driver), and the maximum number of EEPROM variables required. Also, since virtual addresses are 16-bits wide, the maximum number of EEPROM variables cannot exceed 0xFFFE (0xFFFF corresponds to an erased Flash line). Moreover, the number of variables is limited by the size of the product Flash memory (see *Section 4.5: Computing the required size of Flash for EEPROM emulation*).

Each element also contains a 16-bit CRC that is used to check the element integrity. When data is modified, the modified data associated with the same virtual address is stored in a new Flash memory location. Data retrieval returns the up-to-date data value.

Figure 3. Flash Page and EEPROM variable format



3.4 Simple use case

The following example shows the software management for three EEPROM variables with the virtual addresses shown in *Table 5*

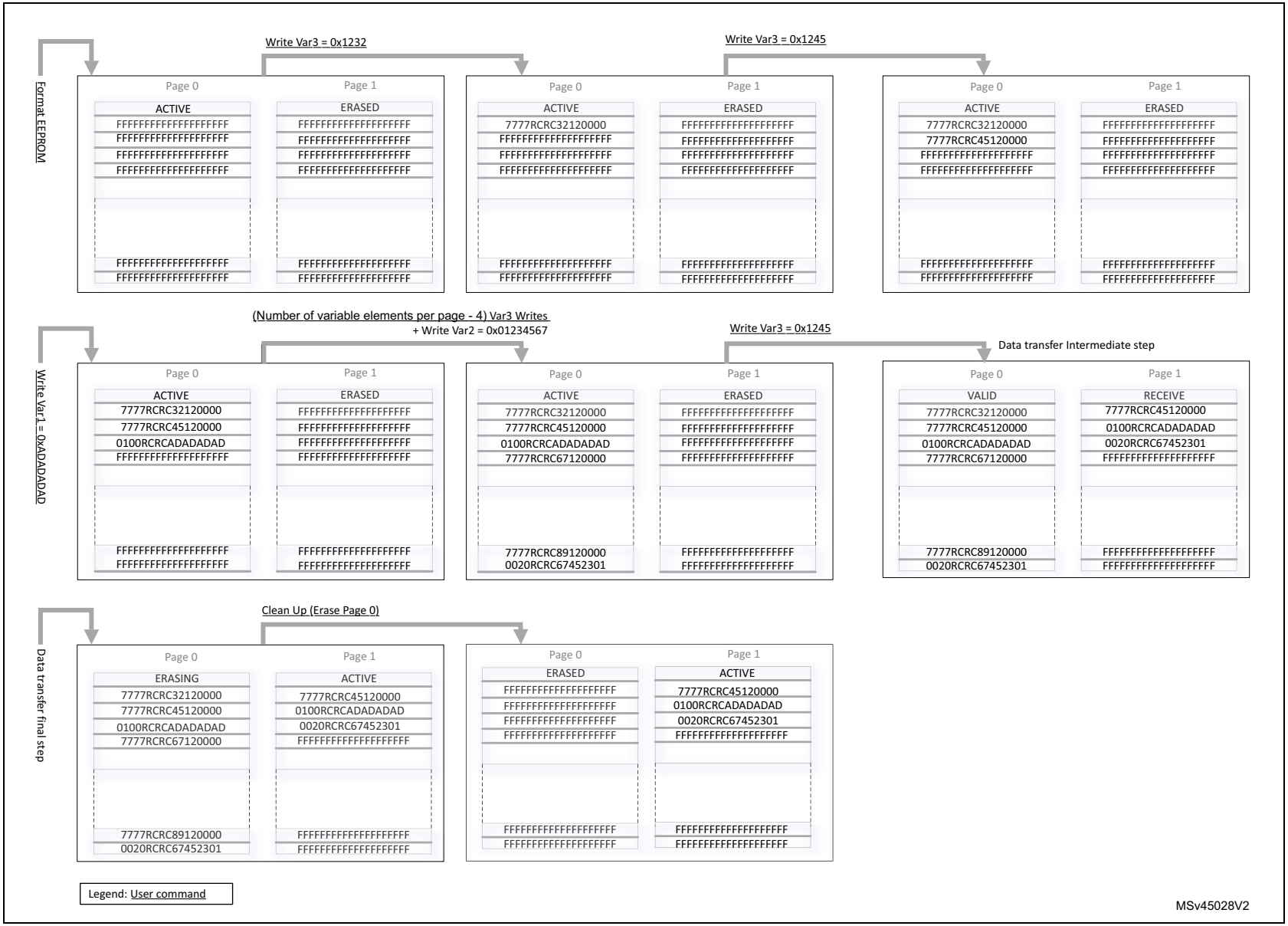
Table 5. Emulated EEPROM virtual addresses

Variable	Virtual address	Data width
Var1	0x0001	32-bit
Var2	0x2000	32-bit
Var3	0x7777	16-bit

For this example, only two Flash pages are necessary: Page 0 and Page 1. In *Figure 4: Data update flow* RCRC represents the 16-bit CRC value for this variable element (refer to *Section 4.6: EEPROM emulation robustness* for more details about CRC).

Figure 4: Data update flow shows only the write commands which are done sequentially in increasing order of Flash addresses.

Figure 4. Data update flow



3.5 Reading data

Read commands perform Flash reads from the highest to the lowest address in the ACTIVE or VALID page, and return only valid data. Data is considered valid if it is the latest to be written at a given virtual address, and the integrity check using CRC passes. Note also that only valid data is copied during the data transfer mechanism.

4 Advanced features

The EEPROM emulation firmware is designed to fulfill most of the requirements for an embedded application in terms of non-volatile storage. This section covers these requirements in detail. Other embedded application requirements may be needed in particular cases and are addressed in section *Section 6: Embedded application aspects*.

4.1 Data granularity management

Emulated EEPROM can be used in embedded applications where non-volatile storage of data updated with a byte, half-word, or word granularity is required. Data size generally depends on application requirements such as sensor or communication-interface data size.

The EEPROM emulation firmware is designed to support byte, 16-bit half-word and 32-bit word granularity. However, in order to optimize the Flash memory usage, the user application could gather all smaller sized data elements into 32-bit data elements before storing the content in emulated EEPROM. This would ensure an optimal use of the 64-bit Flash line by simultaneously writing the 16-bit virtual address, the 16-bit CRC and the 32-bit data value.

Note: The minimal write width in Flash memory is 64-bits due to its ECC (Error Correcting Code) that cannot be switched off.

4.2 Wear leveling algorithm and Flash page allocation

A wear leveling algorithm allows monitoring and even distribution of Flash Write and Erase operations between Flash pages. When no wear-leveling algorithm is used, the pages are not used at the same rate. For instance, pages with long-lived data do not endure as many Write and Erase 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 Flash page.

By design, the EEPROM emulation algorithm distributes evenly the Flash write and Erase operations between Flash pages. Flash writes are performed sequentially in increasing-address order whatever user variable is written. When one set of pages is full, the valid elements are copied to the other set of pages, and the first set of pages is fully erased.

4.3 Guard pages

In order to even further reduce wear on Flash pages, the user can decide to add an even number of Flash guard pages (1 guard page per set of pages by default). No guard page is necessary if the number of emulated variables leaves significant room in the ACTIVE page. Increasing this number (4,6,...) increases the Flash endurance beyond the guaranteed value. This feature is closely linked to the emulated EEPROM cycling capability - refer to paragraph *Section 4.4: Cycling capability: EEPROM endurance improvement*.

Taking an example based on the STM32L4, an emulation of 1000 EEPROM variables could be stored in two sets of 4 Flash pages (each page being able to store 252 elements). When all elements are written once (or after a page transfer), only 8 more elements can be written before a new page transfer is triggered. In this case, the addition of 2 guard pages is recommended (one per set of pages) so that 260 writes can be performed before a new page transfer is triggered.

4.4 Cycling capability: EEPROM endurance improvement

When EEPROM technology is used, each byte can be individually programmed a finite number of times (typically in the range of 1 million). When Flash technology is used, it is not possible to write to a non-erased address, and the minimum erase size is the Flash page size. Consequently, we need to define a program/erase cycle which consists of several Flash line write accesses followed by one Flash page erase operation.

Each STM32 device on-chip Flash memory page can be programmed and erased reliably a limited number of times. For write-intensive applications that need to update each variable more than this number, the wear leveling algorithm allows the endurance of the emulated EEPROM to be increased.

Table 6 shows the limit number of reliable programming and erasing operations for devices covered by this application note:

Table 6. Flash memory endurance

STM32 Series	Flash memory endurance
STM32L4 Series	10 kcycles
STM32L4+ Series	10 kcycles
STM32L5 Series	10 kcycles
STM32G0 Series	10 kcycles (1 kcycles for STM32G030x6/x8 and STM32G070CB/KB/RB)
STM32G4 Series	10 kcycles
STM32WB Series	10 kcycles

Knowing the requested size of emulated EEPROM and the targeted endurance, it is possible to compute the Flash memory size to be used for that purpose. The Flash memory size is also a function of the data width of stored variables.

Table 7. Flash usage for a 4000-byte emulated EEPROM (STM32L4/L4+)

Data width	Number of pages needed for 10 kcycles endurance	Flash size for 10 kcycles endurance	Number of pages needed for 100 kcycles endurance	Flash size for 100 kcycles endurance
STM32L4				
8-bit	34	68 Kbytes	322	644(1) Kbytes
16-bit	18	36 Kbytes	162	324 Kbytes
32-bit	10	20 Kbytes	82	164 Kbytes
STM32L4+				
8-bit	18	72 Kbytes	162	648 Kbytes
16-bit	10	40 Kbytes	82	328 Kbytes
32-bit	6	24 Kbytes	42	168 Kbytes

1. Not applicable for STM32L4: a maximum of 512 Kbytes can be allocated to EEPROM emulation. Please refer to the STM32 product datasheet for the size of each bank.

Note: To get the same information for other STM32 Series you can apply the formula presented in Section 4.5: Computing the required size of Flash for EEPROM emulation.

4.5 Computing the required size of Flash for EEPROM emulation

As first approximation, the required size of Flash memory is proportional to the emulated EEPROM size and to the cycling capability. *Table 7: Flash usage for a 4000-byte emulated EEPROM (STM32L4/L4+)* can be used to estimate the required size of Flash memory according to the data width of stored elements for STM32L4 Series and STM32L4+ Series MCUs.

For a precise Flash memory size requirement, whatever the STM32 Series, use the formula below:

$$\text{Flash size} = \left(\left\lceil \frac{\text{Number of EEPROM variables}}{\text{Variable elements per page}} \right\rceil \times \frac{\text{Number of cycles}}{\text{Flash memory endurance (kcycles)}} \times 2 + \text{Number of guard pages} \right) \times \text{Page size}$$

Where $\lceil \ \rceil$ represents the next higher integer

Note: Values of 'Flash memory endurance values (kcycles)' for applicable STM32 Series are listed in Table 6: Flash memory endurance.

The formula is decomposed into the following steps:

1. Knowing the number of elements a Flash page can store (refer to *Table 4: Flash memory properties*) compute the number of Flash pages needed to store the EEPROM elements. (The calculated Flash memory size is independent of the element data width).
2. Multiply the number of pages by the targeted integer cycling ratio (for instance 1 for the standard 10 kcycles endurance or 10 to achieve 100 kcycles endurance when the Flash memory endurance is 10 kcycles). The Flash-memory endurance of the product (indicated in *Table 6: Flash memory endurance*), must be taken into account.
3. Multiply the number of Flash pages by 2 in order to take into account the data transfer mechanism.
4. Add the number of guard pages (an even number) to get the final number of Flash pages.
5. Knowing the Flash page size for your product (refer to *Table 4: Flash memory properties*), compute the Flash memory size required by the EEPROM emulation.

STM32L4 calculation example

In order to store 4000 individual bytes, and knowing that each page can store up to 252 elements, a set of pages must comprise 16 Flash pages. A second set of pages, of the same size, is required to transfer data when the first one is full. If we assume that we use 2 guard pages, 34 Flash pages are necessary.

Note: This calculation is a slightly conservative estimation.

4.6 EEPROM emulation robustness

In an embedded application, it is possible that a power failure or asynchronous reset might occur while programming or erasing the Flash memory. In this case, the content of the Flash line (if programming) or the complete Flash page (if erasing) is unknown. There could be a single- or a multiple-bit error, with or without Error Correcting Code Detection (ECCD) or correction (ECCC).

Note: The hardware ECC is designed to detect and correct errors after Flash memory wear, but not to detect Flash program or erase operation interruptions reliably.

The EEPROM emulation software is designed and validated to be robust against power failures and asynchronous resets. This robustness relies on several features in the Flash interface and EEPROM emulation driver that are detailed in the following paragraphs. Refer to the product errata sheet for the Flash limitations and workarounds.

4.6.1 Data recovery

In order to detect corrupted data (virtual address and/or data value), a 16-bit CRC (cyclic redundancy check) has been implemented. It is based on the ANSI CRC-16 with the following polynomial: $x^{15} + x^2 + 1$ (represented as 0x8005). Even though this polynomial offers a very high detection rate, it can be modified by the user in the EEPROM configuration file.

When writing a variable element, it is stored with its corresponding CRC value. When reading or transferring a variable element, the CRC is computed and checked against the value stored in the variable element. If it matches, the variable element is considered as valid. In the case of a mismatch, the variable element is invalidated.

Note: The CRC peripheral is used by the EEPROM emulation software in order to accelerate the CRC computation.

The data recovery feature relies on the fact that a full zeros (and only a full 64-bit zeros) can be written to an already programmed non-null word; invalidating a variable element is done by writing zeros in the variable element. As a consequence, this precludes the use of 0x0000 as virtual address for the variable element.

Note: The virtual address 0xFFFF is also forbidden as it corresponds to an erased Flash line.

The data recovery mechanism also requires the user to write zeros in case an ECCD NMI is triggered on this Flash line. This is achieved by calling the `FI_DeleteCorruptedFlashAddress` function from the NMI service routine in case the ECC detects an error that it cannot correct (that is, the ECCD bit is set).

This mechanism is only applied during the cleanup phase operated in the `EE_Init` function. After this phase, it is assumed that there are no further NMI trigger events.

It is possible that, for some reason, the zero programming operation fails and that a Flash programming error flag (PROGERR, PGAERR or PGSERR) is raised (for example if the Flash line was already at 0, PROGERR is raised).The line then is considered invalid and is kept in its current state (the NMI service routine exits without doing anything).

If flags other than PROGERR, PGAERR or PGSERR are set, the NMI routine enters an infinite loop.

If, despite the cleanup phase, another NMI is triggered during the EEPROM emulation, nothing is done. The fact that the NMI is raised is considered to invalidate the line. The cleanup phase eliminates the majority of lines corrupted by an asynchronous reset or power down. Thus, after the cleanup phase, the number of remaining NMIs should, in most cases, be 0.

If, during the first write to a variable element with a given address, the write is interrupted by a power failure or an asynchronous reset, the driver considers that there is no data at this address. In all other cases the EEPROM emulation software always returns the latest valid data value by finding the previous value for this data stored in Flash memory.

4.6.2 Page header recovery

Similarly to data corruption, the page header can be corrupted in the case of power loss or asynchronous reset during a header update or a Flash page erase.

To detect this corruption and recover from it, the `EE_Init` routine is implemented. It should be called immediately after power-up. This routine checks the sequence of page statuses for integrity and performs a repair if necessary. This ensures that no data is lost. Refer to *Section 5.3: EEPROM emulation timing* for more details.

In order to avoid a possible failure scenario, erased pages are systematically erased again upon reset (in the `EE_Init` routine). This ensures a safe behavior but consumes cycling capability of the emulated EEPROM. This systematic erase can be avoided if the application is designed not to generate an asynchronous reset or power failure during Flash writes or erases. Refer to *Section 6.2: Detecting power failures* for more details.

Robustness is achieved at the expense of several checks and recovery mechanisms implemented in the `EE_Init` routine. This induces a slight code size increase and longer initialization time when compared with simpler EEPROM emulations. See *Section 5.3: EEPROM emulation timing* for further details.

4.7 Real-time considerations

The provided implementation of the EEPROM emulation firmware runs from internal Flash memory, thus accesses to the same Flash bank are stalled during Flash erase or programming operations (EEPROM initialization, variable update or page erase).

As a consequence, the application code is not executed and interrupts cannot be served during a maximum time equal to the longest page-erase operation duration for your STM32 product (see the applicable datasheet). This behavior may be acceptable for many applications, however for applications with real-time constraints, the user needs to take corrective action.

4.7.1 Devices embedding Flash memory with RWW (Read While Write) capability

With STM32 devices embedding a dual-bank Flash memory, it is recommended to put the critical routines (Vector table, critical interrupt service routines) in one Bank and the area used for EEPROM emulation in the other bank. The Flash area in the bank used for EEPROM emulation can still be used, but with execution delays during Flash Write and Erase operations.

Note: In order to know whether your STM32 device supports Flash Read While Write capability, please refer to the product datasheet.

4.7.2 Running the critical processes from the internal RAM

Another way to fulfill real-time constraints is to run the critical code from internal RAM:

1. Relocate the vector table into the internal RAM.
2. Execute all critical processes and interrupt service routines from the internal RAM. The compiler provides a keyword to locate the functions in RAM; the functions are copied from the Flash memory to the RAM at system startup just like any initialized variable.

Note: It is important to note that for a RAM function, all used data and called functions should be located within the RAM.

4.7.3 Dual core considerations

On some STM32 devices, two cores are available (refer to the applicable STM32 product datasheet and reference manual for more information). This is the case *for a majority* of the products from the STM32 Wireless Series for example.

STM32WB Series MCUs integrate a main core (CPU1, ARM Cortex-M4) alongside a second core (CPU2, ARM Cortex-M0+) that is dedicated to radio. Here we consider an example in a radio context, where CPU1 runs the EEPROM emulation and needs to process a Flash operation.

To achieve this while avoiding conflicts with CPU2, CPU1 must check that the radio application running on CPU2:

- is not currently accessing Flash memory
- does not urgently need to access Flash memory.

Note: Erase operations in particular take a relatively long time regarding the radio protocol specifications. Therefore, a mechanism must also be implemented to avoid the start of a long CPU1 operation (such as an erase) if CPU2 has already planned an urgent Flash operation.

Application developers adapting the EEPROM emulation application for a dual-core product may need to implement these verifications.

For instance, for STM32WB Series MCUs it is possible to use the following features:

- the hardware semaphore feature (HSEM)
- the CRPF bit from the PWR_EXTSCR register bit (set when a critical radio operation is ongoing).

Both are available on STM32WB Series MCUs.

Note: The same problematic applies to some dual-core applications other than radio.

For STM32WB Series products integrating a Bluetooth® Low-Energy (BLE) stack, a ‘Flash access management’ mechanism, based principally on the hardware semaphore feature, is described in an application note *[2]*.

For the P-NUCLEO-WB55 Nucleo board, two examples are provided in the X-CUBE-EEPROM firmware package (see *Section 5.1.2: STM32Cube expansion software (X-CUBE-EEPROM)*):

- An example that simply exploits the EEPROM emulation driver (similar to the examples for other Series).
- An example that maintains a BLE connection and communication while processing EEPROM operations. It follows the Flash access timing rules in AN5289 *[2]*, and can be used as a reference for implementing this mechanism.

To determine the availability of the above mechanisms for other ST MCUs with RF stacks, please refer to their datasheets and other related documentation.

4.8 Cleaning up the Flash memory in Interrupt or Polling mode

Once the valid elements are transferred to a new set of pages, the previous set of pages needs to be erased. The EEPROM emulation driver raises a flag requesting the main application to clean up the Flash memory. This clean up can be delayed by the application program until the real time constraints on the application are lowered (for instance, before going to low power mode).

This Flash clean up can be done in polling mode (by calling the `EE_CleanUp` function) meaning that the cleanup function does not return before one set of Flash pages used for EEPROM emulation is erased. It can also be done in interrupt mode (by calling the `EE_CleanUp_IT function`) meaning that the cleanup function returns immediately and following page erases are done by subsequent interrupt service routines.

5 API and application examples

5.1 EEPROM emulation software description

This section describes the driver implemented for EEPROM emulation using the STM32Cube firmware for your device Series, provided by STMicroelectronics.

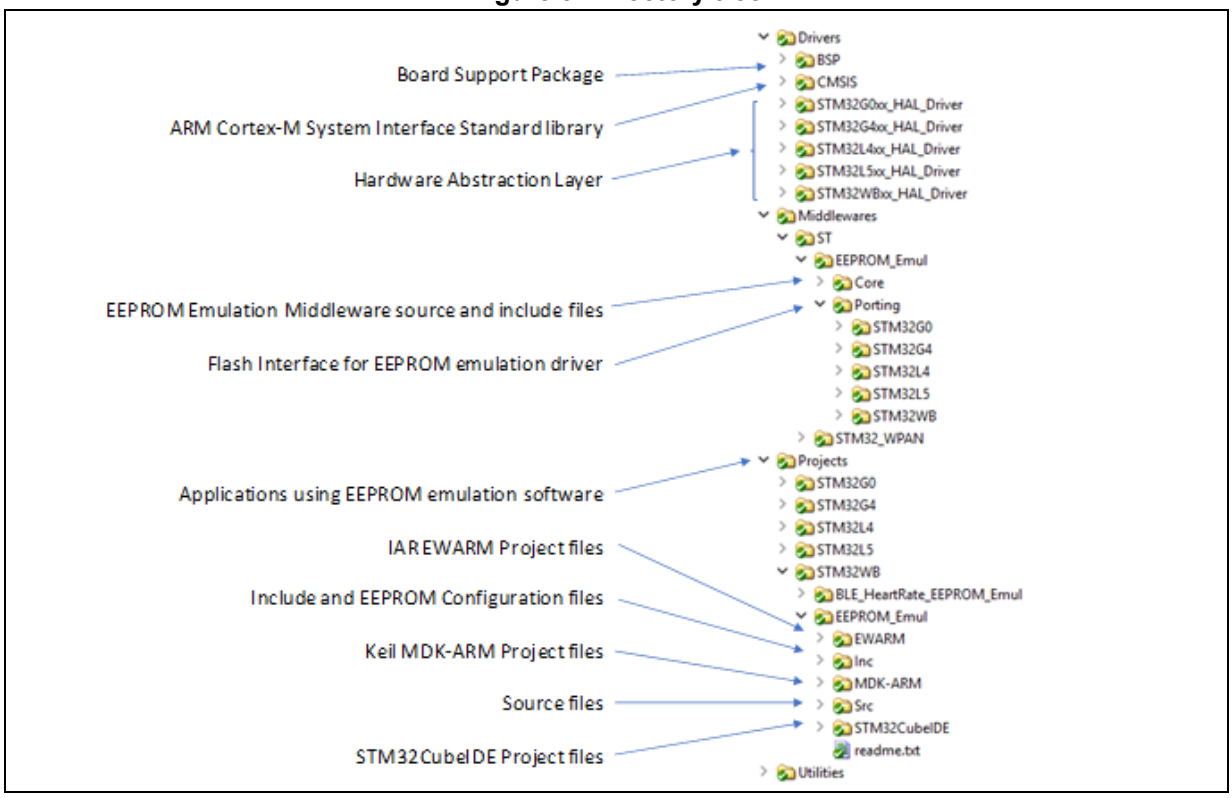
5.1.1 Key features

- user-configured EEPROM size
- supports 8-bit, 16-bit and 32-bit variables
- increased EEPROM memory endurance versus Flash memory endurance
- wear-leveling algorithm
- possible interrupt servicing during program and erase operations
- robust against asynchronous resets and power failures
- Optional protection implementation for Flash-memory sharing between cores in multicore STM32 devices (for example STM32WB Series products).

5.1.2 STM32Cube expansion software (X-CUBE-EEPROM)

The EEPROM emulation software is provided as a dedicated software package called X-CUBE-EEPROM. It is classified as a middleware utility in order to be compliant with all development boards and product Series.

Figure 5. Directory tree



A sample demonstration project using the EEPROM emulation driver is also supplied in order to demonstrate how to manage 1000 non-volatile variables. Sample projects are provided for the boards listed in *Table 8*. They are located in the *Projects\STM32xx\EEPROM_Emul* package folders, and can easily be tailored to other boards using the same STM32 MCU Series.

One more example is provided for the P-NUCLEO-WB55.Nucleo board. It is located at *Projects\STM32WB\BLE_HeartRate_EEPROM_Emul\P-NUCLEO-WB55.Nucleo*. The Nucleo board maintains a BLE connection and communication in addition to processing EEPROM operations. By using the DUALCORE_FLASH_SHARING preprocessor-defines symbol, this example follows the Flash sharing mechanism and timing rules described in *Section 4.7.3: Dual core considerations*.

Table 8. Boards covered

STM32 Series	Boards covered
STM32L4 Series	STM32L476G-Discovery
STM32L4+ Series	STM32L4R5ZI_Nucleo
STM32L5 Series	NUCLEO-L552ZE-Q
STM32G0 Series	NUCLEO-G071RB
STM32G4 Series	NUCLEO-G431RB
STM32WB Series	P-NUCLEO-WB55

The project contains four source files:

- ***eeprom_emul.c***: contains the EEPROM emulation firmware functions that can be called from the user program:

  ```
  EE_Format
  EE_Init
  EE_ReadVariable32bits
  EE_WriteVariable32bits
  EE_ReadVariable16bits
  EE_WriteVariable16bits
  EE_ReadVariable8bits
  EE_WriteVariable8bits
  EE_CleanUp
  EE_CleanUp_IT
  EE_DeleteCorruptedFlashAddress
  ```

- ***flash_interface.c***: contains the functions needed to handle the STM32 Flash specific features.
- ***main.c***: this application program is an example using the described routines in order to configure, write to and read from the emulated EEPROM.
- ***stm32xxxx_it.c***: shows an example of interrupt service file using the `EE_DeleteCorruptedFlashAddress` function in the NMI handler.

For the BLE STM32WB Series example, `BLE_HeartRate_EEPROM_Emul`, the code illustrating the driver exploitation can be found in the *Core/Src/app_entry.c* file. In particular, the functions `EEPROM_Emul_Init` and `EEPROM_Emul_Operation` are examples of sequencer tasks for EEPROM driver initialization and exploitation (read and write operations).

5.1.3 User defines

The EEPROM emulation algorithm parameters should be configured according to the application needs; they are located in the eeprom_emul_conf.h file:

- **NB_OF_VARIABLES** (default 1000): Number of non-volatile elements, each element value being 8-, 16- or 32-bit.
- **START_PAGE_ADDRESS**: Address of the first Flash page used for EEPROM emulation.
  For dual-bank capable devices, the first address of the second bank is a good example value for this parameter in order to benefit from this feature (see *Section 4.7.1: Devices embedding Flash memory with RWW (Read While Write) capability*).
- **CYCLES_NUMBER** (default 1): The number of kcycles, X, for the equivalent EEPROM endurance. If CYCLES_NUMBER equals 10, the emulated EEPROM has an equivalent endurance of 10 x X kcycles. X is the Flash endurance for the product used and listed in *Table 6: Flash memory endurance*.
- **GUARD_PAGES_NUMBER** (default 2): Number of guard pages used to reduce pressure on Flash memory. This number has to be even.
- **CRC_POLYNOMIAL_LENGTH** (default 16): No need to modify in most cases; a 16-bit CRC is optimized in terms of computational speed and Flash size, and offers a very good detection rate.
- **CRC_POLYNOMIAL_VALUE** (default 0x8005): No need to modify in most cases; this ANSI CRC is optimized in terms of computational speed and Flash size, and offers a very good detection rate.

To activate the dual-core Flash sharing mechanism in the context of the Bluetooth Low Energy stack of the STM32WB Series, DUALCORE_FLASH_SHARING must be defined as a preprocessor symbol in user software.

5.1.4 User API definition

The set of functions that can be called by the applicative program are described in *Table 9*. They are defined in the *eeprom_emul.c* module.

Table 9. API definition

Function name	Description
EE_Format	Erases all Flash pages used for EEPROM emulation and writes an ACTIVE header to the first page.
EE_Init	Configures the EEPROM emulation variables to their initial state and restores the Flash pages to a known good state in case of asynchronous reset or power loss during Flash write or erase operation. Erases the Flash pages that need to be erased (for instance pages not fully erased in ERASE state, pages in ERASING state). The EE_FORCE_ERASE parameter has to be used by default. If the application guarantees that no asynchronous reset nor power loss can happen during Flash write or erase operations, then the EE_CONDITIONAL_ERASE parameter can be used; refer to *Section 4.6.2: Page header recovery*. After reset, this function should be systematically called prior to accessing the emulated EEPROM.
EE_ReadVariableXXbits	This function updates the data value corresponding to the virtual address passed as a parameter. Only the last stored element is read. It returns a Status equal to EE_OK unless it cannot find any data at the given virtual address. Three functions are implemented to deal with all supported data sizes (XX = 8, 16 or 32).
EE_WriteVariableXXbits	This function updates the EEPROM at a given virtual address with the data value passed as parameters. If the set of pages used for EEPROM emulation is full, it triggers a Flash page transfer and returns a Status equal to EE_CLEANUP_REQUIRED. In the case of the BLE application for the NUCLEO-WB55 Nucleo board (DUALCORE_FLASH_SHARING is defined), it can return EE_FLASH_USED when the Flash is used by CPU2(1) Three functions are implemented to deal with all supported data sizes (XX = 8, 16 or 32).
EE_CleanUp	Erases the set of pages in ERASING state. The application program can call this function when real-time constraints are low. The Flash cleanup is done in polling mode so this function does not return before one set of Flash pages used for EEPROM emulation is erased.
EE_CleanUp_IT	Erases the set of pages in ERASING state. The Flash cleanup is started in interrupt mode so this function returns immediately and following page erase is done by subsequent interrupt service routines.
FI_DeleteCorruptedFlashAddress	This function can be called under NMI to delete a corrupted Flash Address found by the Flash ECCD in order to clear this Flash interface Fault *Section 4.6.1: Data recovery*.

1. EE_WriteVariableXX bits can return EE_FLASH_USED only in the case of direct EEPROM variable writing and not in the case of a page transfer, a page header update or the initialization process. Also, when this value is returned, the driver can activate the HSEM interrupt so that an interrupt is raised when the CPU2 releases the semaphore associated to the Flash device being used. The user program can then do something else while waiting for the semaphore release.

5.2 EEPROM emulation memory footprint

Table 10 details the footprint of the EEPROM emulation driver in terms of Flash memory, RAM and Stack size.

The figures shown have been determined using the IAR EWARM 8.42.2 tool with High Size optimization level. Note that this algorithm does not use Heap.

Table 10. Memory footprint for EEPROM emulation mechanism

STM32 Series	Required(1) code and data size in bytes	
	Flash	SRAM
STM32L4 Series	4256	12
STM32L4+ Series	4286	12
STM32L5 Series	4728	12
STM32G0 Series	3812	12
STM32G4 Series	4946	12
STM32WB Series	5392	16
STM32WB Series(2)	3328	14

1. Based on 1000 elements (16-bit address, 16-bit CRC and 32-bit data)
2. Example with BLE communication

Table 10 does not take into account the Flash size used for EEPROM emulation itself. Refer to the formula given in *Section 4.5: Computing the required size of Flash for EEPROM emulation* to calculate the corresponding Flash memory size.

Note also that the application program (*main.c*) implements the `VarDataTab[]` variable to store the EEPROM element values. Its size is 4000 bytes for 1000 32-bit elements.

For flexibility, this variable is placed in RAM in the provided example. Moreover `VarDataTab[]` can easily be removed as most user applications do not need to have a RAM copy of the elements stored in Flash memory.

5.3 EEPROM emulation timing

This section describes the timing parameters associated with the EEPROM emulation driver based on 1000 32-bit variables.
All timing measurements are performed with the following conditions:

- At VDD = 3.3 V
- With execution from Flash memory (from the bank not used for EEPROM variables, when dual bank is used)
- At room temperature
- Based on 100 operations for each types of operation.

Table 11 lists the timing values for the EEPROM emulation driver in these conditions. In other conditions (for instance for a different number of variables, cycles endurance or a different MCU frequency), these timings can differ significantly.

Table 11. EEPROM emulation timings for different targets

Target	Conditions	Operation	Mean	Min.	Max.
STM32L476G-DISCO	– System clock = 80 MHz – ART enabled – Flash prefetch disabled – Endurance = 100 kcycles	EEPROM Initialization with forced Erase(1)	1.2215 s	904.4 ms	2.0212 s
		EEPROM Initialization with conditional Erase(1)(2)	365.56 ms	6.1 ms	1.2499 s
		Write operation in EEPROM	See note(3)	90 µs	344.90 ms
		Read operation from EEPROM(4)	47.412 µs	9.3 µs	311.5 µs
		EEPROM Cleanup(1)	900.27 ms	900.13 ms	900.38 ms
		EEPROM Cleanup IT(1)	900.19 ms	900 ms	900.38 ms
NUCLEO-L4R5ZI	– System clock = 80MHz(5) – ART enabled – Flash prefetch disabled – Endurance = 100 kcycles	EEPROM Initialization with forced Erase(1)	741.69 ms	509.6 ms	884 ms
		EEPROM Initialization with forced Erase(1)(2)	466.91 ms	5.8 ms	776.9 ms
		Write operation in EEPROM	See note(3)	90 µs	307.20 ms
		Read operation from EEPROM(4)	75.639 µs	9 µs	314.3 µs
		EEPROM Cleanup(1)	461.90 ms	461.8 ms	462 ms
		EEPROM Cleanup IT(1)	461.86 ms	461.8 ms	462 ms

Table 11. EEPROM emulation timings for different targets

Target	Conditions	Operation	Mean	Min.	Max.
NUCLEO-L552ZE-Q	– System clock = 110 MHz – ICACHE disabled – Endurance = 10 kcycles	EEPROM Initialization with forced Erase(1)	365.10 ms	111.1 ms	670 ms
		EEPROM Initialization with conditional Erase(1)(2)	267.5 ms	1.5 ms	680 ms
		Write operation in EEPROM	See note(3)	90 µs	580.18 ms
		Read operation from EEPROM(4)	149.3 µs	10 µs	485.6 µs
		EEPROM Cleanup(1)	109.85 ms	109.80 ms	109.89 ms
		EEPROM Cleanup IT(1)	109.85 ms	109.81 ms	109.88 ms
NUCLEO-G071RB	– System clock = 64 MHz – Endurance = 10 kcycles	EEPROM Initialization with forced Erase(1)	251.17 ms	111.4 ms	814.4 ms
		EEPROM Initialization with conditional Erase(1)(2)	233.43 ms	1.47 ms	792.9 ms
		Write operation in EEPROM	See note(3)	90 µs	702.18 ms
		Read operation from EEPROM(1)	102.16 µs	14 µs	592 µs
		EEPROM Cleanup(1)	110.39 ms	110.37 ms	110.40 ms
		EEPROM Cleanup IT(1)	110.41 ms	110.40 ms	110.43 ms
NUCLEO-G431RB	– System clock = 170 MHz – Endurance = 10 kcycles	EEPROM Initialization with forced Erase(1)	179.33 ms	110.3 ms	349.4 ms
		EEPROM Initialization with conditional Erase(1)(2)	160.90 ms	600 µs	356.3 ms
		Write operation in EEPROM	See note(3)	90 µs	260.67 ms
		Read operation from EEPROM(4)	74.242 µs	6 µs	170 µs
		EEPROM Cleanup(1)	109.89 ms	109.86 ms	109.89 ms
		EEPROM Cleanup IT(1)	109.90 ms	109.89 ms	109.92 ms
P-NUCLEO-WB55.Nucleo(6)	– System clock = 64 MHz – Endurance = 10 kcycles	EEPROM Initialization with forced Erase(1)	276.83 ms	67.27 ms	561.32 ms
		EEPROM Initialization with conditional Erase(1)(2)	258.65 ms	1.86 ms	539.78 ms
		Write operation in EEPROM	See note(3)	90 µs	476.26 ms
		Read operation from EEPROM(4)	289.25 µs	11 µs	570 µs
		EEPROM Cleanup(1)	65.867 ms	65.82 ms	65.92 ms
		EEPROM Cleanup IT(1)	64.966 ms	64.90 ms	65.92 ms

1. The maximum initialization time and cleanup time depends on the number of Flash EEPROM pages used, and the time to erase one page. (See the section "Flash memory characteristics" in the STM32 product datasheet). The time taken to erase Flash EEPROM pages = number of pages * tERASE.
2. When the application ensures the EEPROM emulation cannot be interrupted by an asynchronous reset or power failure, a conditional Erase can be used. It is more efficient in terms of execution time and also in terms of EEPROM endurance (refer to *Section 4.6.2: Page header recovery* for more details).
3. The typical write time is close to the minimal write time as there is no data transfer in most cases. The maximum value refers to a write operation that generates a data transfer.
4. The minimum value refers to a read operation of the last element stored in the Flash memory and the maximum value refers to a read operation of first element stored in Flash memory.
5. Timing measurements for the STM32L4+ Series board (NUCLEO-L4R5ZI) has been done for a 80 MHz system clock so that a comparison can be done with the STM32L4 Series (NUCLEO-L476RG). However, the NUCLEO-L4R5ZI can be pushed up to 120 MHz while the NUCLEO-L476RG is limited at 80 MHz. The measurements for the other boards are done for the maximum system clock.
6. Data taken from example without BLE capabilities.

Note: These timing measurements are provided in order to give the reader an idea of the emulation firmware performance for different STM32 Series. They are based on 100 operations for each type of operation, thus they are not an absolute reference. It is possible to find different minimums, maximums and even means (especially if the measurement conditions are modified).

6 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.

6.1 Data retention

The Flash data retention is typically 30 years @ 85°C after cycling 1000 times (please refer to the STM32 product datasheet for more complete data). Nothing specific is done in the EEPROM emulation driver to increase the EEPROM data retention. However, when a page transfer is initiated, all data (including long-lived data) is copied to a new Flash page. This ensures an EEPROM data retention significantly higher than the original Flash data retention.

6.2 Detecting power failures

If no asynchronous reset can be generated by the application, the only way to corrupt the Flash programming or erasing operations is the case of power failure. The application can use the PVD in order to not generate a Flash write or erase operation during the VDD ramp down. An example of this PVD (Programmable Voltage Detector) usage is provided in the EEPROM example main application.

The decoupling capacitors or battery must be sized to provide enough power to the chip during the full write or erase operation and before the supply voltage goes below 1.7 V (minimal operating voltage) or before a BOR (Brown Out Reset) is generated. In this case, the `EE_CONDITIONAL_ERASE` parameter of the `EE_Init` routine can be used. In all other cases, the `EE_FORCE_ERASE` parameter has to be used.

6.3 Reducing worst case access times

An EEPROM read command consists of performing Flash reads from the highest to the lowest address in the ACTIVE and VALID pages. If the data to retrieve was the first to be written, this process can be quite long. Similarly, the page transfer searches for all virtual addresses used during EEPROM emulation and looks for corresponding data values. These read and transfer times can be quite long when the size of emulated EEPROM is large.

The worst-case read and write access times can be reduced by implementing a LUT (Look Up Table) storing the element values in RAM. Although implementing a LUT does not affect the typical access times and is quite costly in terms of RAM space, it could still be an option implemented in a future revision of the EEPROM emulation driver.

7 Conclusion

The STM32 devices' internal Flash memory can be used advantageously in many applications to emulate an EEPROM. Even though the memory characteristics are quite different, this application note has shown that emulated EEPROM competes with real EEPROM in terms of:

- Memory size
- Read and write access times
- Driver usage simplicity and flexibility
- Memory endurance
- Data retention
- Robustness to asynchronous resets and power failures
- Reliability

and mainly:

- cost by removing the need for an external component.

8 Revision history

Table 12. Document revision history

Date	Revision	Changes
11-Jul-2017	1	Initial release.
20-Sep-2018	2	Updated document to scope STM32L4 and STM32L4+ Series devices. Updated: – *Figure 1: Page status evolution* – *Figure 2: Page status valid transitions* – *Section 3.3: Page and variable format* – *Figure 3: Flash Page and EEPROM variable format* – *Section 4.4: Cycling capability: EEPROM endurance improvement* – *Table 7: Flash usage for a 4000-byte emulated EEPROM (STM32L4/L4+)* – *Section 4.7.1: Devices embedding Flash memory with RWW (Read While Write) capability* – *Section 5.1.3: User defines* – *Section 5.2: EEPROM emulation memory footprint* – *Section 5.3: EEPROM emulation timing* Added *Section 4.5: Computing the required size of Flash for EEPROM emulation*.
01-Jun-2020	3	Updated: – Document title – *Introduction* – *Section 3.3: Page and variable format* and *Table 4: Flash memory properties* – *Section 3.4: Simple use case* – *Section 4.4: Cycling capability: EEPROM endurance improvement* and *Table 6: Flash memory endurance* – *Section 4.5: Computing the required size of Flash for EEPROM emulation* – *Section 4.6.1: Data recovery* – *Section 4.7.3: Dual core considerations* – *Section 5.1.1: Key features* – *Section 5.1.2: STM32Cube expansion software (X-CUBE-EEPROM)*, *Figure 5: Directory tree*, and *Table 8: Boards covered* – *Section 5.1.3: User defines* – *Section 5.1.4: User API definition* and *Table 9: API definition* – *Section 5.2: EEPROM emulation memory footprint* and *Table 10: Memory footprint for EEPROM emulation mechanism* – *Section 5.3: EEPROM emulation timing* and *Table 11: EEPROM emulation timings for different targets*.