
**EEPROM emulation techniques and software for
STM32L4 and STM32L4+ Series microcontrollers**

Introduction

EEPROMs (electrically erasable programmable read-only memories) are often used in industrial and consumer 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.

In order 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 STM32L4xxxx on-chip Flash memory.

Emulation is achieved by employing at least two pages in the Flash memory. 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 has the following features:

- Lightweight implementation and reduced footprint
- Simple API that consists 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.

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



Contents

1	Main differences between external and emulated EEPROM	6
1.1	Difference in write access time	7
1.2	Programming and erase operations	7
2	Implementing EEPROM emulation	8
2.1	Principle	8
2.2	Page status valid transitions	9
2.3	Page and variable format	10
2.4	Simple use case	11
2.5	Reading data	13
3	Advanced features	14
3.1	Data granularity management	14
3.2	Wear leveling algorithm and Flash page allocation	14
3.3	Guard pages	14
3.4	Cycling capability: EEPROM endurance improvement	15
3.5	Computing the required size of Flash for EEPROM emulation	16
3.6	EEPROM emulation robustness	17
3.6.1	Data recovery	17
3.6.2	Page header recovery	17
3.7	Real-time considerations	18
3.7.1	Devices embedding Flash memory with RWW (Read While Write) capability	18
3.7.2	Running the critical processes from the internal RAM	18
3.8	Cleaning up the Flash memory in Interrupt or Polling mode	19
4	STM32L4/L4+ API and application examples	20
4.1	EEPROM emulation software description	20
4.1.1	Key features	20
4.1.2	STM32Cube expansion software (X-CUBE-EEPROM)	20
4.1.3	User defines	22
4.1.4	User API definition	23

4.2	EEPROM emulation memory footprint	24
4.3	EEPROM emulation timing	25
5	Embedded application aspects	27
5.1	Data retention	27
5.2	Detecting power failures	27
5.3	Reducing worst case access times	27
6	Conclusion	28
7	Revision history	29

List of tables

Table 1.	Differences between external and emulated EEPROM	6
Table 2.	Emulated EEPROM virtual addresses	11
Table 3.	Flash usage for a 4000-byte emulated EEPROM (STM32L4/L4+).	15
Table 4.	API definition.	23
Table 5.	Memory footprint for EEPROM emulation mechanism	24
Table 6.	STM32L4 EEPROM emulation timings with an 80 MHz system clock	26
Table 7.	STM32L4+ EEPROM emulation timings with an 80 MHz system clock	26
Table 8.	Document revision history	29

List of figures

Figure 1.	Page status evolution	8
Figure 2.	Page status valid transitions	9
Figure 3.	Flash Page and EEPROM variable format	11
Figure 4.	Data update flow	12
Figure 5.	Directory tree	20

1 Main differences between external and emulated EEPROM

STM32L4-Series microcontrollers are based on an Arm^{®(a)} core. 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 STM32L4 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 STM32L4-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 1](#).

Table 1. 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 in 4 ms. Word program time = 16 ms Page (32 bytes) Write 4 ms. Consecutive Words program time = 500 μ s	Word program time: from 97 μ s to 376 ms ⁽¹⁾
Erase time	N/A	2 Kbytes page Erase time: 22 ms
Memory Size	From a few Kbytes to 256 Kbytes	Only limited by the size of Flash memory allowed for EEPROM emulation (up to 512 Kbytes)
Read access	Serial: a hundred μ s Random word: 92 μ s Page: 22.5 μ s per byte	Parallel: (@ 80 MHz) the access time is from 8.9 μ s to 331 μ s ⁽¹⁾
Endurance	4 million cycles at 25 °C 1.2 million cycles at 85 °C 600 kilocycles at 125 °C	10 kilocycles per page @ 105°C. Using multiple on-chip Flash memory pages is equivalent to increasing the number of write cycles. See Section 3.4: Cycling capability: EEPROM endurance improvement .
Retention	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 4.3: EEPROM emulation timing](#).

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

1.1 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.

1.2 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.

2 Implementing EEPROM emulation

2.1 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.

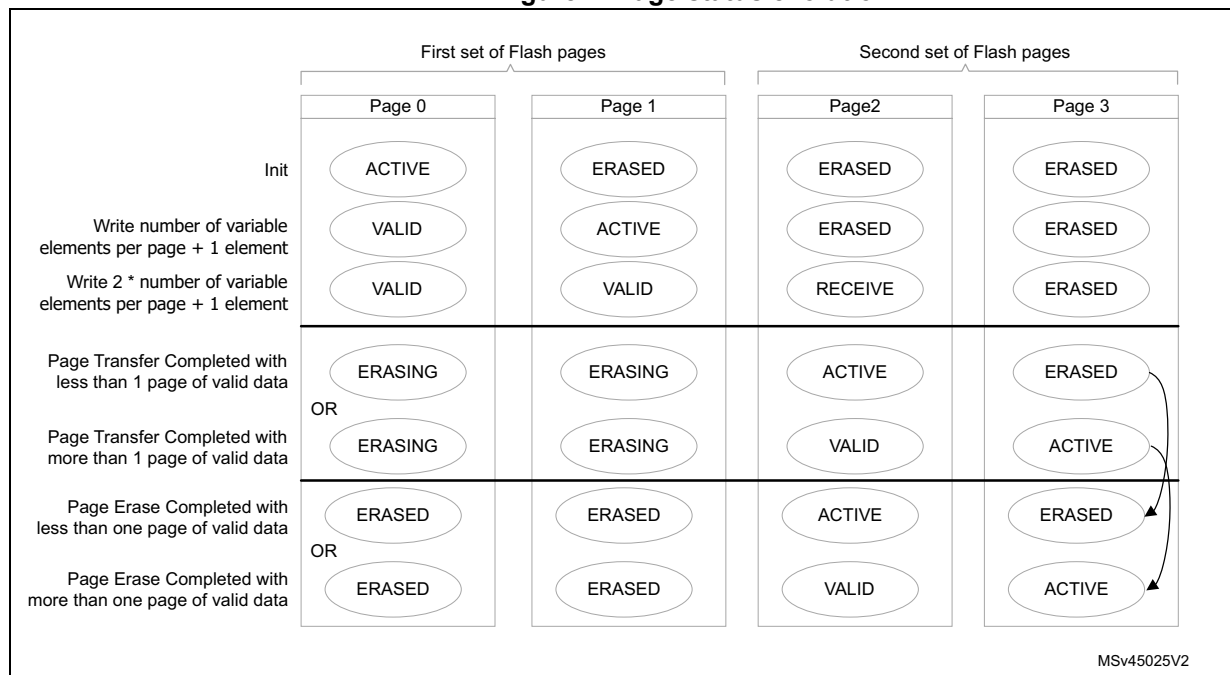
The second set of pages collects only the valid data from the first set of pages and the remaining area can be used to store new data. Once the transfer of valid data to the second set of pages is completed, the first set of pages can be erased.

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.

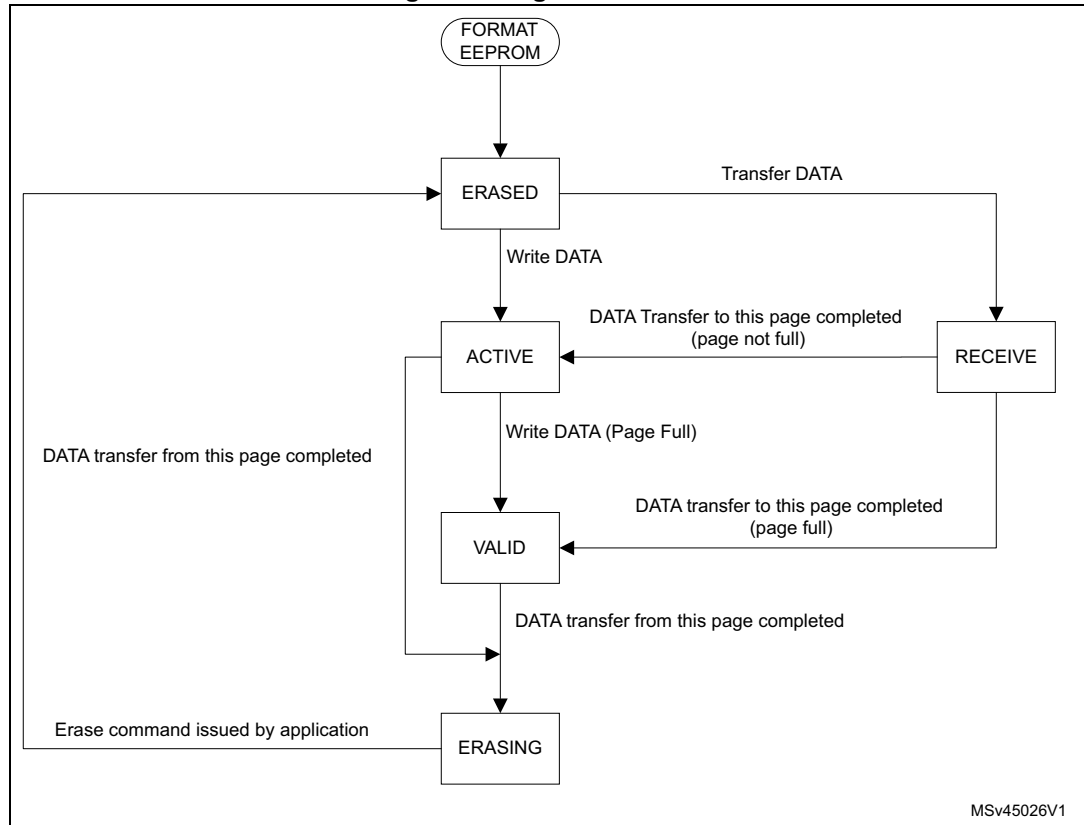
Figure 1. Page status evolution



2.2 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



2.3 Page and variable format

For the STM32L4 each 2 Kbyte page comprises 256 words of 64 bits. For the STM32L4+ each 4 Kbyte page comprise 512 words of 64 bits. For the STM32L4+ the single bank mode (DBANK=0) is not supported by the EEPROM emulation driver. The dual bank mode must be selected (DBANK = 1). 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 for the STM32L4, and up to 508 variable elements for the STM32L4+.

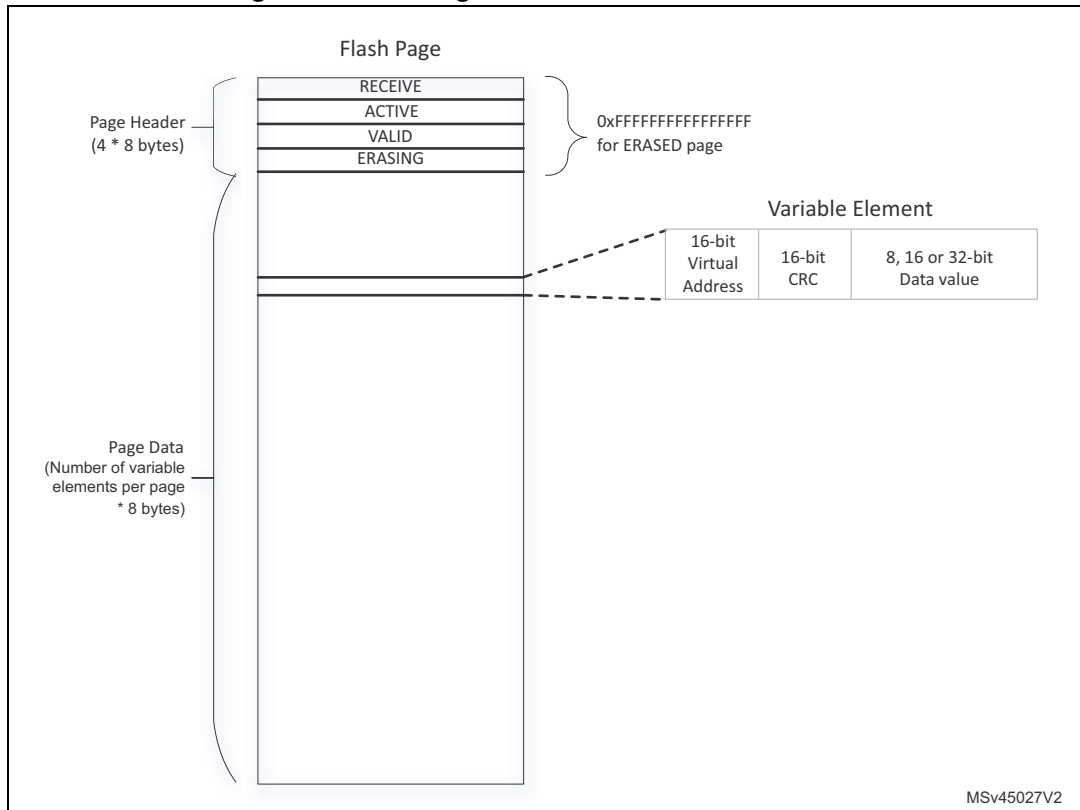
The possible states of a Flash page are coded by writing 0xAAAAAAAAAAAAAAAA into the page header. It is possible to determine the page state using the following procedure:

- 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 2.2: 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. 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



2.4 Simple use case

The following example shows the software management for three EEPROM variables with the virtual addresses shown in [Table 2](#):

Table 2. Emulated EEPROM virtual addresses

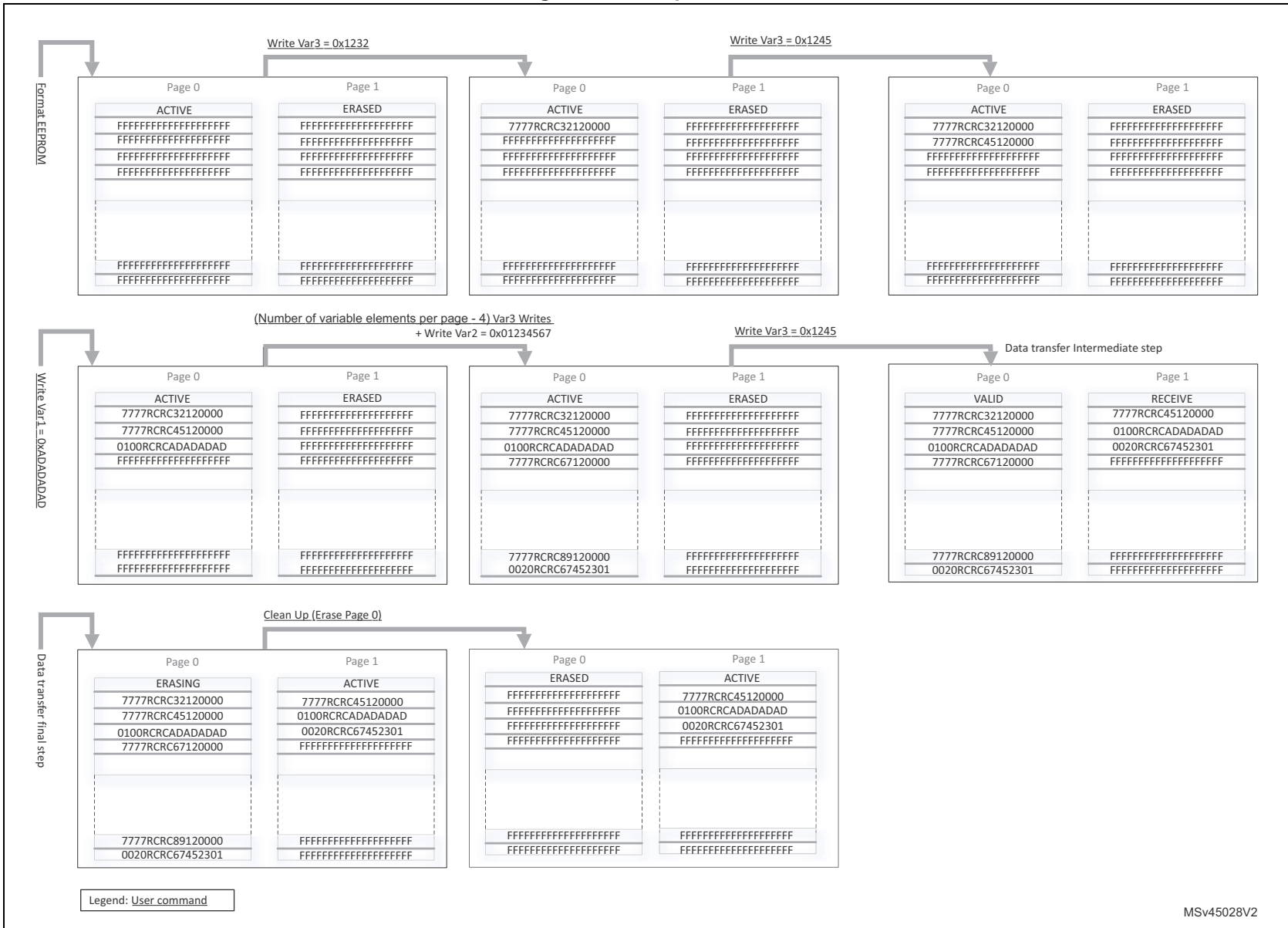
Variable	Virtual address	Data width
Var1	0x0001	32-bit
Var2	0x2000	32-bit
Var3	0x7777	16-bit

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 3.6: EEPROM emulation robustness](#) for more details about CRC).

[Figure 4](#) shows only the Write commands which are done sequentially in increasing order of Flash addresses.



Figure 4. Data update flow



2.5 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.

3 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 5: Embedded application aspects](#).

3.1 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.

3.2 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.

3.3 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 3.4: 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.

3.4 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 STM32L4 on-chip Flash memory page can be programmed and erased reliably 10 000 times. For write-intensive applications that need to update each variable more than 10 000 times, the wear leveling algorithm allows the endurance of the emulated EEPROM to be increased.

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 3. 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 product datasheet for the size of each bank.

3.5 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 3: 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 a precise Flash memory size requirement, use the formula below:

$$\text{Flash Size} = \left\lceil \frac{\text{Number of EEPROM variables}}{\text{Variable elements per page}} \right\rceil \times \left(\frac{\text{Number of cycles}}{10 \text{ kcycles}} \times 2 + \text{Number of Guard pages} \right) \times \text{Page size}$$

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

This formula is decomposed in the following steps:

1. Knowing that each Flash page can store 252 variable elements for STM32L4 (508 for STM32L4+) (whatever their data width), compute the number of Flash pages needed to store the EEPROM elements.
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)
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 that each Flash page size is 2 Kbytes for the STM32L4 (4 Kbytes for the STM32L4+), 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.

3.6 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.

3.6.1 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 `EE_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).

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.

3.6.2 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 4.3: 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 5.2: 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 4.3: EEPROM emulation timing](#) for further details.

3.7 Real-time considerations

The provided implementation of the EEPROM emulation firmware runs from the 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 of 24.5 ms (maximum Flash page erase time). This behavior may be acceptable for many applications, however for applications with real-time constraints, the user needs to take corrective actions.

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

With STM32L4 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 STM32L4/L4+ device supports Flash Read While Write capability, please refer to the product datasheet.

3.7.2 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.

3.8 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.

4 STM32L4/L4+ API and application examples

4.1 EEPROM emulation software description

This section describes the driver implemented for EEPROM emulation using the STM32CubeL4 firmware provided by STMicroelectronics.

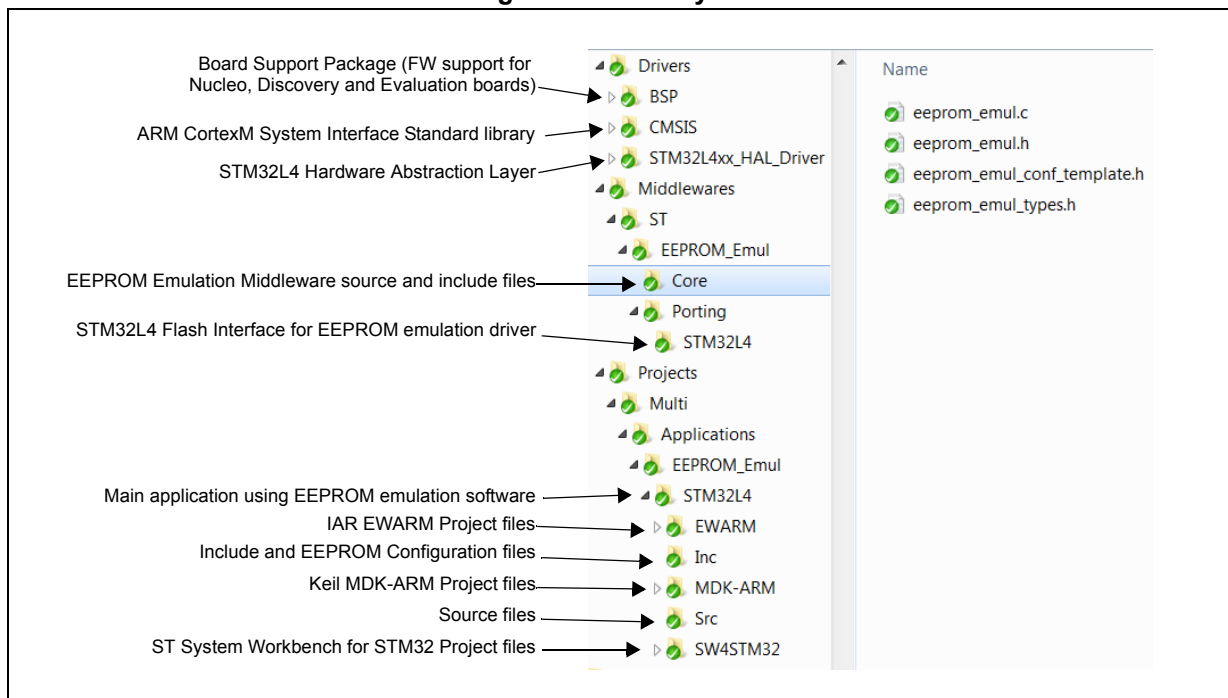
4.1.1 Key features

- 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

4.1.2 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 middleware 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. The sample project is provided for the STM32L476 Discovery board and STM32L4R5ZI Nucleo 144 board, and can easily be tailored to any other STM32L4xxxx board.

The project contains four source files:

- ***eeeprom_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 STM32L4/L4+ 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.
stm32l4xx_it.c: shows an example of interrupt service file using the `EE_DeleteCorruptedFlashAddress` function.

4.1.3 User defines

The EEPROM emulation algorithm parameters should be configured according to the application needs; they are located in the `eprom_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 STM32L4, the default address (0x08080000) is selected in order to benefit from the STM32L4 dual-bank feature (read while write). On devices embedding less than 1 Mbyte of Flash memory, this address has to be modified by the user.
 - For STM32L4+, the default address (0x08100000) is selected to benefit from the STM32L4+ dual bank feature (read while write). On devices embedding less than 2 Mbytes of Flash memory, this address has to be modified by the user.
- **CYCLES_NUMBER** (default 1): Number of 10 kcycles for the equivalent EEPROM endurance. If **CYCLES_NUMBER** equals 10, the emulated EEPROM has an equivalent endurance of 100 kcycles.
- **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.

4.1.4 User API definition

The set of functions that can be called by the applicative program are described in [Table 4](#). They are defined the `eeprom_emul.c` module.

Table 4. API definition

Function name	Description
<code>EE_Format</code>	Erases all Flash pages used for EEPROM emulation and writes an ACTIVE header to the first page.
<code>EE_Init</code>	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 <code>EE_FORCE_ERASE</code> 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 <code>EE_CONDITIONAL_ERASE</code> parameter can be used; refer to Section 3.6.2: Page header recovery . After reset, this function should be systematically called prior to accessing the emulated EEPROM.
<code>EE_ReadVariableXXbits</code>	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 <code>EE_OK</code> 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).
<code>EE_WriteVariableXXbits</code>	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 <code>EE_CLEANUP_REQUIRED</code> . Three functions are implemented to deal with all supported data sizes (XX = 8, 16 or 32).
<code>EE_CleanUp</code>	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.
<code>EE_CleanUp_IT</code>	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.
<code>EE_DeleteCorruptedFlashAddress</code>	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.

4.2 EEPROM emulation memory footprint

[Table 5](#) 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 8.22.1 tool with High Size optimization level. Note that this algorithm does not use Heap.

Table 5. Memory footprint for EEPROM emulation mechanism

Size in bytes	Required ⁽¹⁾ code and data size		
	Flash	SRAM	Stack
EEPROM emulation software mechanism	4030	12	72

1. Based on 1000 elements (16-bit address, 16-bit CRC and 32-bit data)

[Table 5](#) does not take into account the Flash size used for EEPROM emulation itself. Please refer to [Table 3: Flash usage for a 4000-byte emulated EEPROM \(STM32L4/L4+\)](#) to calculate the corresponding Flash memory size.

Note also that the application program (main.c) implements two variables:

- VarDataTab[] to store the EEPROM element values. Its size equals 4000 bytes for 1000 32-bit elements.
- VirtAddVarTab[] to store the EEPROM element virtual addresses. Its size equals 2000 bytes for 1000 elements.

For flexibility, both variables are placed in RAM in the provided example. VarDataTab[] can easily be removed as most user applications do not need to have a RAM copy of the elements stored in Flash memory. However, the VirtAddVarTab[] table is needed by the EEPROM emulation driver. If RAM size is a concern, it is recommended to place VirtAddVarTab[] in Flash memory rather than in RAM assuming that the Virtual addresses in Emulated EEPROM are known at compile time.

4.3 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:

- on STM32L476VGT6 Revision 4 (typical process)
- on STM32L4R5ZIT6U (typical process)
- at VDD = 3.3 V
- 100 kcycles endurance (82 pages for STM32L4 and 42 pages for STM32L4R)
- System clock at 80 MHz with ART enabled and Flash prefetch disabled,
- With execution from Flash memory (from the bank not used for EEPROM variables)
- At room temperature.

[Table 6](#) 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 6. STM32L4 EEPROM emulation timings with an 80 MHz system clock

Operation	Min	Max
EEPROM Initialization with forced Erase ⁽¹⁾	8 ms	2.1 s
EEPROM Initialization with conditional Erase ⁽¹⁾⁽²⁾	6 ms	1.2 s
Write operation in EEPROM ⁽³⁾	97 μs	376 ms
Read operation from EEPROM ⁽⁴⁾	8.9 μs	331 μs
EEPROM Cleanup ⁽¹⁾	903 ms	904 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 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 3.6.2: 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.

Table 7. STM32L4+ EEPROM emulation timings with an 80 MHz system clock

Operation	Min	Max
EEPROM Initialization with forced Erase ⁽¹⁾	7.2 ms	1.12 s
EEPROM Initialization with conditional Erase ⁽¹⁾⁽²⁾	6.3 ms	680 ms
Write operation in EEPROM ⁽³⁾	99.7 μs	308 ms
Read operation from EEPROM ⁽⁴⁾	8.3 μs	278.5 μs
EEPROM Cleanup ⁽¹⁾	462 ms	463 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 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 3.6.2: 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 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.

5 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.

5.1 Data retention

The Flash data retention is typically 30 years @ 85°C after cycling 1000 times (please refer to the product datasheet for more complete data). There is nothing specific 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.

5.2 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.

5.3 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.

6 Conclusion

The internal STM32L4/L4+ 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.

7 Revision history

Table 8. Document revision history

Date	Revision	Changes
11-Jul-2017	1	Initial release.
20-Sep-2018	2	<p>Updated document to scope STM32L4 and STM32L4+ Series devices.</p> <p>Updated:</p> <ul style="list-style-type: none"> – Figure 1: Page status evolution – Figure 2: Page status valid transitions – Section 2.3: Page and variable format – Figure 3: Flash Page and EEPROM variable format – Section 3.4: Cycling capability: EEPROM endurance improvement – Table 3: Flash usage for a 4000-byte emulated EEPROM (STM32L4/L4+) – Section 3.7.1: Devices embedding Flash memory with RWW (Read While Write) capability – Section 4.1.3: User defines – Section 4.2: EEPROM emulation memory footprint – Section 4.3: EEPROM emulation timing <p>Added Section 3.5: Computing the required size of Flash for EEPROM emulation.</p>

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2018 STMicroelectronics – All rights reserved