EFM32

... the world's most energy friendly microcontrollers

EEPROM Emulation

AN0019 - Application Note

Introduction

This application note demonstrates a way to use the flash memory of the EFM32 to emulate single variable rewritable EEPROM memory through software. The example API provided enables reading and writing of single variables to non-volatile flash memory. The erase-rewrite algorithm distributes page erases and thereby doing wear leveling.

This application note includes:

- This PDF document
- Source files (zip)
  - Example C-code
  - Multiple IDE projects

ENERGY micro

www.energymicro.com

1 General Theory

1.1 EEPROM and Flash Based Memory

EEPROM stands for Electrically Erasable Programmable Read-Only Memory and is a type of non-volatile memory that is byte erasable and therefore often used to store small amounts of data that must be saved when power is removed. The EFM32 microcontrollers do not include an embedded EEPROM module for byte erasable non-volatile storage, but all EFM32s do provide flash memory for non-volatile data storage. The main difference between flash memory and EEPROM is the erasable unit size. Flash memory is block-erasable which means that bytes cannot be erased individually, instead a block consisting of several bytes need to be erased at the same time. Through software however, it is possible to emulate individually erasable rewritable byte memory using block-erasable flash memory.

To provide EEPROM functionality for the EFM32s in an application, there are at least two options available. The first one is to include an external EEPROM module when designing the hardware layout of the application. The other is to use the on-chip flash memory and emulate EEPROM functionality through a software API. There are however some key differences between these two methods.

- First, the write access time for flash memory is shorter than for external EEPROM. This means that writing to emulated EEPROM is faster than writing to an external EEPROM.
- Second, while a standalone EEPROM will be able to complete a write operation even if the system is reset, the emulated EEPROM will need the CPU to be active throughout the entire flash operation. This is an important difference. The consequences of an aborted flash operation should be taken into account when designing an application. The flash based EEPROM emulation could use checksums and logging to ensure the integrity of written data.
- Lastly the emulated EEPROM will regularly need to erase pages in flash, to free space and be able to write to the same page more than once. On a standalone EEPROM there is no need for a dedicated erase operation, since all bytes can be rewritten independently. Here it is also important to notice that the flash erase operation will need the CPU to be running safely for the entire operation.

In addition to the risk of power failure or system reset aborting flash write or erase operations, one must also handle internal sources like interrupts trying to execute code from flash. When writing to or erasing flash, any concurrent attempts to access the flash will result in a hard fault exception and core lock up. Any interrupts triggering while the CPU is performing operations on flash must therefore be executed from RAM. This includes the interrupt vector, the interrupt vector table also needs to be relocated to RAM. An alternative is just to disable any interrupts while performing the flash operations. This can be done in software. Any interrupt that would have triggered during the flash operation, can be configured to execute when the flash operation is complete, and interrupts again are enabled.

It should also be mentioned that in order to avoid any bus conflicts, all flash operations must be performed with code from RAM. Attempts to initiate a flash operation from the flash itself, will result in the same exception as the case with interrupts.

1.2 Flash Limitations

Flash memory is limited to a finite number of program-erase cycles. This means that the embedded flash memory of the EFM32 can be erased only a certain number of times, before the wear will begin to affect the integrity of the storage. This deterioration applies to all kinds of flash memory. The amount of on-chip flash memory found in the different EFM32 microcontrollers, varies depending on the specific part. All flash memory is divided into 512 byte pages that each must be erased as single units. Flash pages and words in each page are illustrated in Figure 1.1 (p. 3). Because the erase operation erases only whole pages, it is important to write as much data as possible to a single page of flash, before erasing the page. On the EFM32 MCUs, the flash memory is guaranteed to withstand a minimum of 20k erase cycles.

Figure 1.1. Flash Pages and Words.

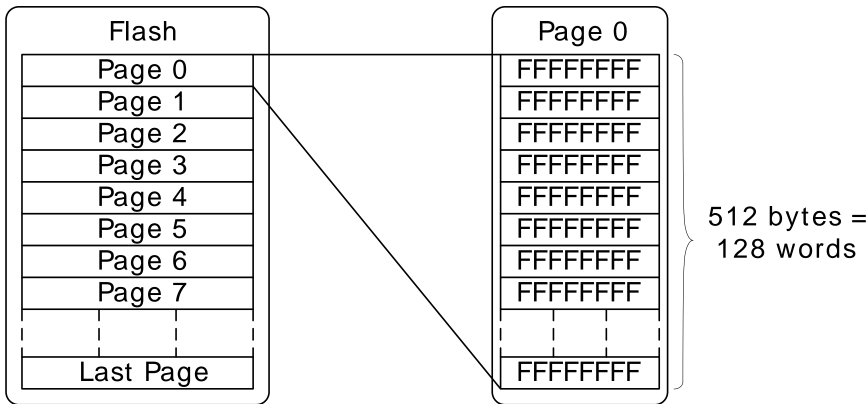


Figure 1.1 (p. 3) shows the flash, and how it is divided in pages that in turn are divided in 32 bit words.

The EFM32 does not support reading from flash while writing or erasing flash. This means that, in addition to power failure or system reset aborting flash write or erase operations, one must also handle internal sources like interrupts trying to execute code from flash. When writing to or erasing flash, any concurrent attempts to access the flash will result in a hard fault exception and core lock up. Any interrupts triggering while the CPU is performing operations on flash must therefore be executed from RAM. This includes the interrupt vector, the interrupt vector table also needs to be relocated to RAM. An alternative is just to disable any interrupts while performing the flash operations. This can be done in software. Any interrupt that would have triggered during the flash operation, can be configured to execute when the flash operation is complete, and interrupts again are enabled.

It should also be mentioned that in order to avoid any bus conflicts, all flash operations must be performed with code from RAM. Attempts to initiate a flash operation from the flash itself, will result in the same exception as the case with interrupts.

2 EEPROM Emulator Realization

2.1 Implementation

There are different ways to implement an EEPROM Emulator using flash memory. The idea behind the example attached in this application note, is to allocate a certain number of pages of flash memory for the entire lifetime of the application. The wear is then leveled among these by alternating which pages are used. The number of pages allocated must reflect the amount of data that will be written throughout the application lifetime.

2.2 Pages and Their States

In every page allocated to the EEPROM Emulator provided, the first word is reserved for page head data. This head word contains the status of the page and the erase count.

The erase count is the number of times all pages have been erased. The erase count is incremented each time the last page is erased and valid variables are transferred back to page 0. This completes one erase cycle on all the flash pages allocated.

Each page will always be in one of three different states. Active, Receiving or Erased.

- After a page is erased, all bits in the entire page are 1's.
- When a page is receiving, it means that a transfer of variables from a full active page is in progress. After the transfer is complete, the receiving page is made the new active one, and the old active page is erased.
- The active page is the page that contains the currently valid data. All read and write operations are made on the active page. There should never be more than one active or receiving page at any time in this implementation.

Figure 2.1. EEPROM Emulation Page Status Flow.

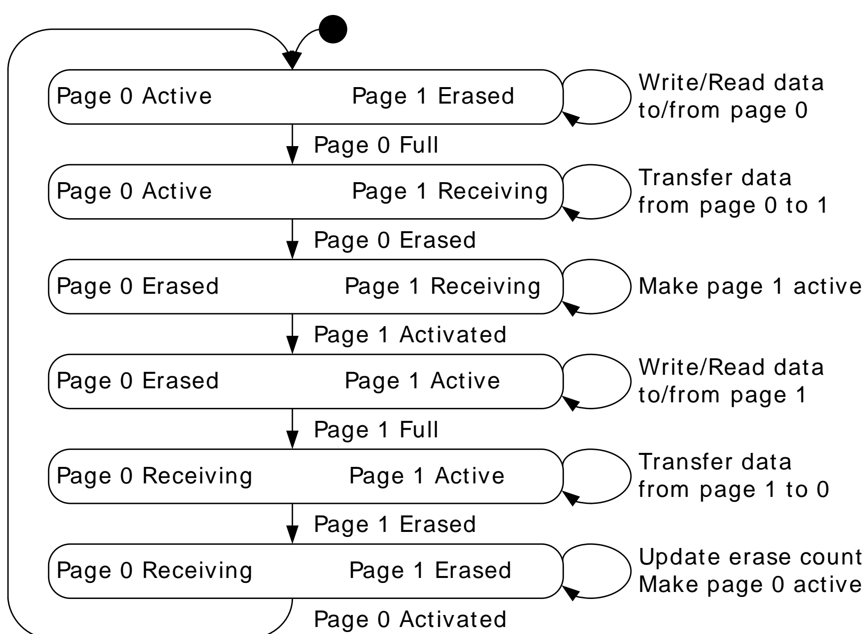


Figure 2.1 (p. 4) shows the state flow for a realization using 2 pages. The flow would be similar if more pages are allocated, only more pages would be in the erased state simultaneously.

During initialization, the page status for the selected number of pages are checked, to ensure that a legal set of page-states are obtained. If there are more than one of either the active or receiving pages,

or if all pages are erased, all pages are reset to a valid configuration and page 0 is the new active page. Page 0 will be the last page in the flash memory, page 1 will be the second last etc. This minimizes the probability for data collisions with program instructions, which are located at the base of the flash. In applications where much of the available flash memory is in use, it can be critical to know where all data is stored in the flash to avoid collisions. The ability to lock the flash from being written to can be a nice feature in this context. Locking individual pages is done by clearing bits in the Lock Bit Page, for further information see the reference manual for the device. Alternatively a memory protection algorithm can be implemented in software.

The remaining words of each page, after the first Page Status Word, are free to be used for data storage. Each data storage word is divided into two parts; one virtual address field and one data field. Each of them are 16 bits wide in this example. Whenever a page is full, a page transfer is initiated. This operation consists of several steps to always ensure that all variables are non-volatile in case of an external event. First, an erased page is located to store the valid data present in the full page. This page is marked as “receiving”. Next, the most recent data associated with each variable is transferred to the top of the new page. When this is done, the old active page is erased, the erase count is written to the receiving page header, lastly it is labeled as the new active page. This process is illustrated in Figure 2.2 (p. 5) .

Figure 2.2. EEPROM Emulation Variable Flow.

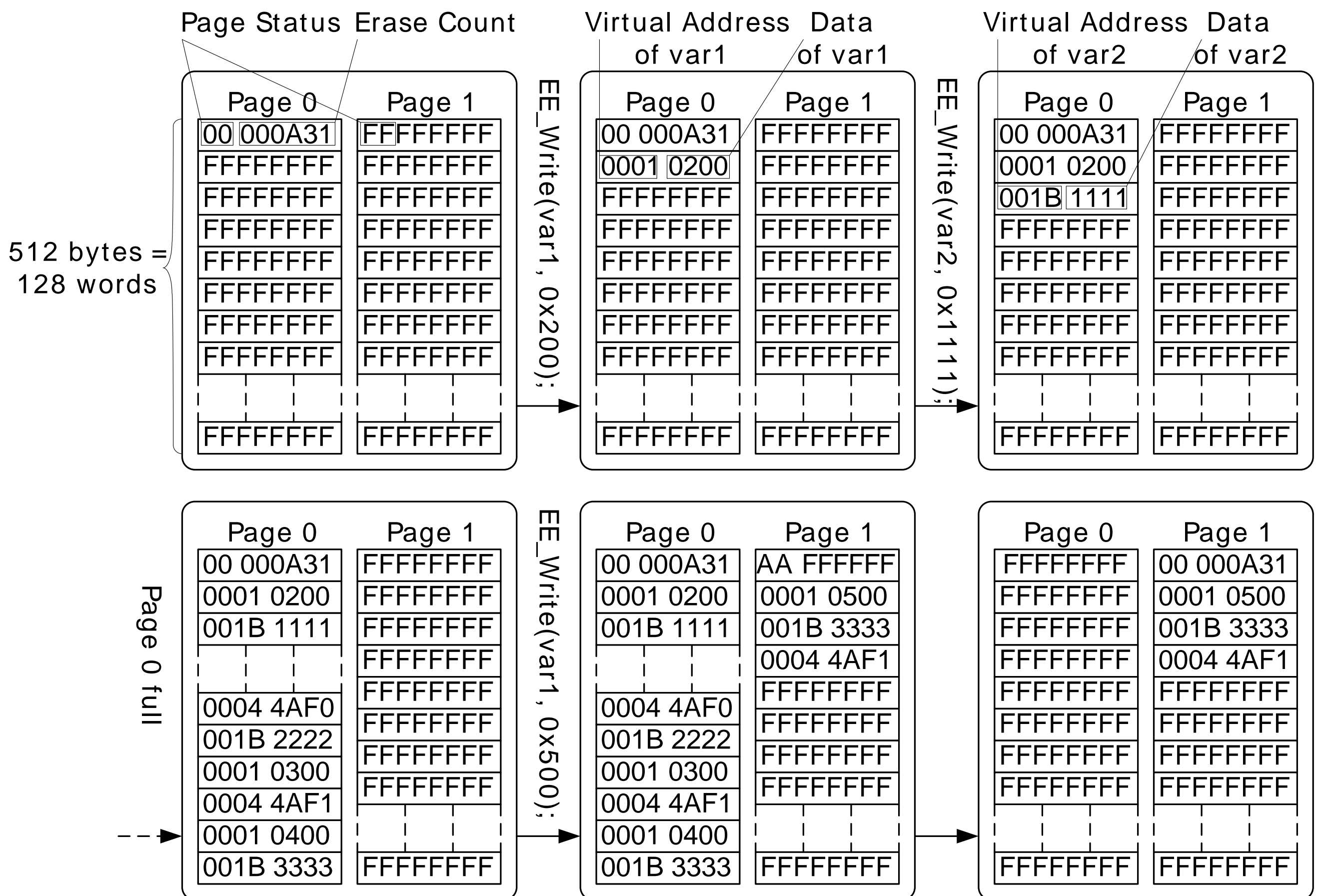


Figure 2.2 (p. 5) shows how the data is written to the pages, and how a the latest valid data is transferred to an erased page when the active page is filled up.

2.3 Read and Write

Whenever data is to be written to a new variable, it should first be declared to give it a unique virtual address. The virtual addresses makes the read operation able to determine what data belongs to which variable. This is necessary since the variables are not stored on a constant absolute address, but are moved around in flash memory to minimize wear. Reading consists of an iteration through the

active page, starting from the bottom. When the correct virtual address is found for the first time, the corresponding data is returned as the currently valid data.

The write operation consists of a similar iteration, only that it is starting at the top of the active page. When an empty word is found, the correct virtual address and data is written, and the function returns. If no empty word is found, and the end of the page is reached, the page is considered full.

2.3.1 Duplicate Data

Since the lifetime of the system is limited by the number of erase cycles available on the flash memory, some functionality is added to avoid unnecessary extra writes to the flash. If the data to be written is identical to what is already written, the entire write operation is skipped. Also, since the flash memory can only change 1 bits to 0's individually through a write, a write to the flash is like a "bitwise and"-operation with what is already written to it. This means that if the new value of a variable leaves all 0's untouched, the old value can safely be overwritten with the new one. This is illustrated in Figure 2.3 (p. 6) . In this way the flash lifetime can be extended a little further.

Figure 2.3. Flash Writes Without Erase

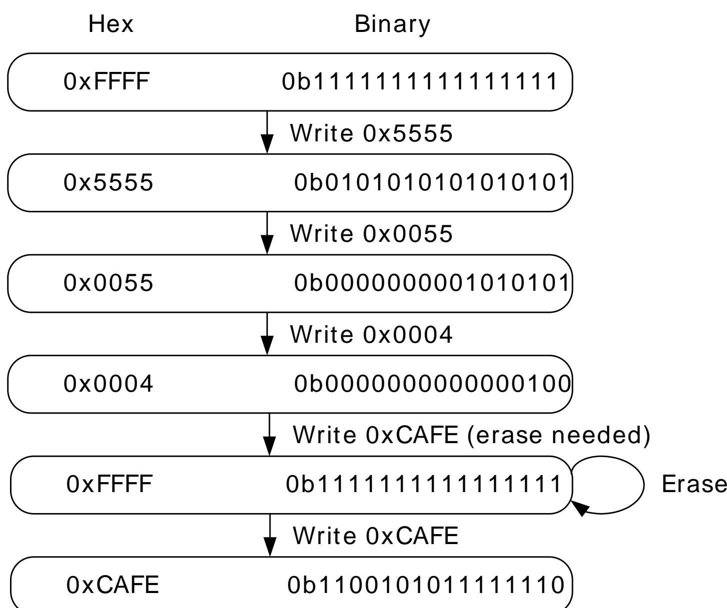


Figure 2.3 (p. 6) shows how an erase is avoided as long as the 1's in the new data written is a subset of the 1's in the previously written data.

2.4 Flash Wear and Page Allocation

For many applications, the flash wear limitation is not an important system restriction. However, for any application that depends on regularly writing data to the non-volatile flash memory, this can be an important lifetime limiting factor.

A 512 byte page with 20k erase cycles, can be used to write a maximum of 10 megabytes throughout the lifetime of the system. It is in this context important to have a good understanding of how much data that will be written in the lifetime of the system to decide how many pages are needed.

In the software example attached, all variables are 16 bit data, with a 16 bit virtual address. This means that all variables uses 4 bytes of flash memory each time new data is written, see Figure 2.2 (p. 5). Accordingly each page can take 128 variable writes, before it is full. Each new page is initially filled with the latest value of all currently valid variables and a head, before it can be written to. This means that before any new value can be written to a variable on a new page, the page will already have filled a space

equal to the number of variables in use. An estimate on the pages needed to guarantee predictable flash operation throughout the application lifetime can be calculated as:

Page number calculations

$$\text{Writes} = 20k * (128 - \text{Variables}) * \text{Pages} \quad (2.1)$$

Where "Writes" is the number of total variable writes, "Variables" is the number of variables in use, and "Pages", is the number of pages needed.

Say for instance that a certain application is designed to update 7 different variables, every 5 minutes for 8 years. This implies 7 * 8 * 365 * 24 * 12 = ~5.9 million writes of 32 bits. The number of pages needed to guarantee non-volatile storage of all this data, is:

Pages = Writes / (20k * (2^7 - Variables)) = 5.9 * 10^6 / (2 * 10^4 * (2^7 - 7)) = 590 / (2 * 121) = 2.44 = ~3 pages

This calculation should give a good estimate of the number of pages needed. However, the fact that some writes will be duplicates and overwrites is not taken into account, which makes this a slightly conservative estimate. If any other data sizes of variables are implemented, i.e. 32/32 bit data/virtual address, or 8/8 bit etc. a similar argument can be made calculating the number of pages needed. Also the number of times all pages have been erased can be obtained runtime, and can be used to validate whether the calculations made have acceptable accuracy.

3 Software Examples

The software example supplied, demonstrates a realization of an EEPROM emulator software API.

3.1 Data Granularity

In this example, the data and virtual address are each assigned 16 bits of space. This may not be sufficient for some applications. The data granularity should therefore be adjusted according to the requirements of the application. Different sets of common variable sizes could be implemented to allow byte rewritable 8/8 bits data/virtual address variables, or full word 32/32 bits variables. It is important to notice that this would also affect the wear and lifetime of the flash memory accordingly. It can for example be noticed that even though 32+32 bits variables would be very useful in many settings, it will more than halve the lifetime of the flash.

3.2 Initialization and Recovery

An important part of an EEPROM emulation software, is to ensure correct page state, and data recovery on system startup. For this reason, the initialization function should be run near the start of the software, before any data is written or read to/from any variables. The initializing function will decide how many pages that are allocated to the emulator, it will then check the head of each of these pages to ensure that the set of pages are valid. The valid sets are:

- One receiving and one active page.
  - Data will be transferred from the active to the receiving page. The active page will be erased, and the receiving page will be marked active.
- Exactly one receiving page.
  - The receiving page will be marked as the new active page.
- Exactly one active page.
  - No action is required.

In addition, pages marked as erased are checked to verify that they really are erased.

If an invalid set of pages is found, either if all are erased or there are more than one of the active or receiving, the set of pages will be formatted. This should usually only happen the first time the software is executed on a MCU with old data present. On a format, all pages are erased and page 0 is set to be the active one. More advanced functionality could be added, recovering some data in case a fatal error corrupts the page headers during normal operation of the system.

3.3 Additional Considerations

The EEPROM emulation example can run on both the STK and DVK with both plug-in cards. Some consideration should however be given to ensure that the required number of pages can be allocated without any conflict with the application binary file located at the beginning of the flash. This could be a problem with parts low on flash memory, or if too many pages are allocated during initialization.

4 Revision History

4.1 Revision 1.04

2012-04-20

Adapted software projects to new peripheral library naming and CMSIS_V3.

4.2 Revision 1.03

2011-10-21

Updated IDE project paths with new kits directory.

4.3 Revision 1.02

2011-05-18.

Updated projects to align with new bsp version.

4.4 Revision 1.01

2010-11-16

Changed example folder structure, removed build and src folders.

Updated chip init function to newest efm32lib version.

Updated register defines in code to match newest efm32lib release.

4.5 Revision 1.00

2010-09-21

Initial revision.

A Disclaimer and Trademarks

A.1 Disclaimer

Energy Micro AS intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Energy Micro products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Energy Micro reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Energy Micro shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products must not be used within any Life Support System without the specific written consent of Energy Micro. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Energy Micro products are generally not intended for military applications. Energy Micro products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

A.2 Trademark Information

Energy Micro, EFM32, EFR, logo and combinations thereof, and others are the registered trademarks or trademarks of Energy Micro AS. ARM, CORTEX, THUMB are the registered trademarks of ARM Limited. Other terms and product names may be trademarks of others.

B Contact Information

B.1 Energy Micro Corporate Headquarters

Postal Address	Visitor Address	Technical Support
Energy Micro AS P.O. Box 4633 Nydalen N-0405 Oslo NORWAY	Energy Micro AS Sandakerveien 118 N-0484 Oslo NORWAY	support.energymicro.com Phone: +47 40 10 03 01

www.energymicro.com
Phone: +47 23 00 98 00
Fax: + 47 23 00 98 01

B.2 Global Contacts

Visit **www.energymicro.com** for information on global distributors and representatives or contact **sales@energymicro.com** for additional information.

Americas	Europe, Middle East and Africa	Asia and Pacific
www.energymicro.com/americas	www.energymicro.com/emea	www.energymicro.com/asia

Table of Contents

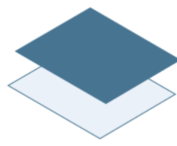
1. General Theory 2
    1.1. EEPROM and Flash Based Memory 2
    1.2. Flash Limitations 2
2. EEPROM Emulator Realization 4
    2.1. Implementation 4
    2.2. Pages and Their States 4
    2.3. Read and Write 5
    2.4. Flash Wear and Page Allocation 6
3. Software Examples 8
    3.1. Data Granularity 8
    3.2. Initialization and Recovery 8
    3.3. Additional Considerations 8
4. Revision History 9
    4.1. Revision 1.04 9
    4.2. Revision 1.03 9
    4.3. Revision 1.02 9
    4.4. Revision 1.01 9
    4.5. Revision 1.00 9
A. Disclaimer and Trademarks 10
    A.1. Disclaimer 10
    A.2. Trademark Information 10
B. Contact Information 11
    B.1. Energy Micro Corporate Headquarters 11
    B.2. Global Contacts 11

List of Figures

1.1. Flash Pages and Words. 3
2.1. EEPROM Emulation Page Status Flow. 4
2.2. EEPROM Emulation Variable Flow. 5
2.3. Flash Writes Without Erase 6

List of Equations

2.1. Page number calculations 7



ENERGY®
micro

Energy Micro AS
Sandakerveien 118
P.O. Box 4633 Nydalen
N-0405 Oslo
Norway

www.energymicro.com