

Application Note

EEPROM Emulation Type A Design



ABSTRACT

This application note introduces the library of type A EEPROM emulation. The document describes the structure and behavior of the type A EEPROM emulation. Then the document introduces the use of the associated APIs. In addition, the relevant code is provided to the users. Users can call functions in applications to write, read, and modify data.

Table of Contents

1 Introduction	2
2 Implementation	2
3 Software Description	5
4 Application Aspects	13
5 References	14

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

Many applications require store data in non-volatile memory, so that the application can be reused or modified even after the system is powered up again. EEPROM is design for such applications. Although the MSPM0 MCUs does not have an internal EEPROM, the MSPM0 internal Flash supports EEPROM emulation. Compared to using an external serial EEPROM, EEPROM emulation using the internal Flash saves pin usage and cost.

Different applications require different storage structures. The type A solution described in this article is suitable for storing large 'block' of data. If the application needs to store small 'variable' data, the type B solution can be referenced.

Difference between EEPROM and on-chip Flash

The EEPROM can erase and write to a single byte of memory multiple times, allowing programmed locations to retain data for long periods of time even if the system is powered off.

Flash memory has a higher density than EEPROM, allowing larger memory arrays (sectors) to be implemented on chip. The flash erase and write cycle is performed by applying a time-controlled voltage to each unit. Each cell (bit) reads the logical value 1 under erase conditions. Therefore, each Flash location reads 0xFFFF After erase. By programming, the cell can be changed to a logic 0. Any word can be overwritten to change a bit from logical 1 to 0; however, the reverse is not possible. In addition, Flash memory has the restriction that the memory is erased by a region. For MSPM0 MCUs, the erase resolution is 'sector' with the size of 1k bytes.

One major difference between EEPROM and Flash is cycling capability. Flash endurance is typically 10000 cycles, far less than real EEPROM. EEPROM emulation is a software based on Flash to provide the equivalent endurance that can be satisfied for the application. It also simulates the behavior of the EEPROM, simplifying the operation of reading and writing data.

A typical emulation scheme involves using a portion of the Flash memory and dividing it into multiple areas. These areas are used alternately to store data. Due to the block erase requirement of Flash, entire Flash sector has to be reserved for the EEPROM emulation. To achieve wear-leveling, at least two sectors are used.

2 Implementation

Principle

In the implementation described in this app note, sectors are divided into areas called 'records' according to the size of the virtual EEPROM. Each record contains header and data. Header shows the status of the record. The remainder of the record (total record size minus the 8-byte size of the header) is used for storing user data. The number of records in one sector is $(\text{SectorSize} / \text{RecordSize})$. For 128-bytes record size, there are 8 records in one sector. [Figure 2-1](#) shows the structure of EEPROM emulation.

All of these records are used to store data of the same virtual EEPROM. When trying to modify the data of the virtual EEPROM, a new record is actually created instead of being modified on the original record. The difference between records in Flash is whether the data is new or old. In other word, they are different versions of the same data. It is more like a backup for memory area in RAM.

There are three user-configurable parameters, which can be configured in `eeeprom_emulation_type_a.h` due to the application requirements. These parameters affect space usage and cycling capability, which will be analyzed later.

- Record size: 64 bytes or 128bytes or 256bytes
- Number of sectors used: at least 2
- Sector address

The basic behaviors of EEPROM emulation can be seen in [Figure 2-2](#). When write operation is performed, user's data is stored into Flash to be a new record. When read operation is performed, the latest record is read. The erase operation is performed only when the sector is full.

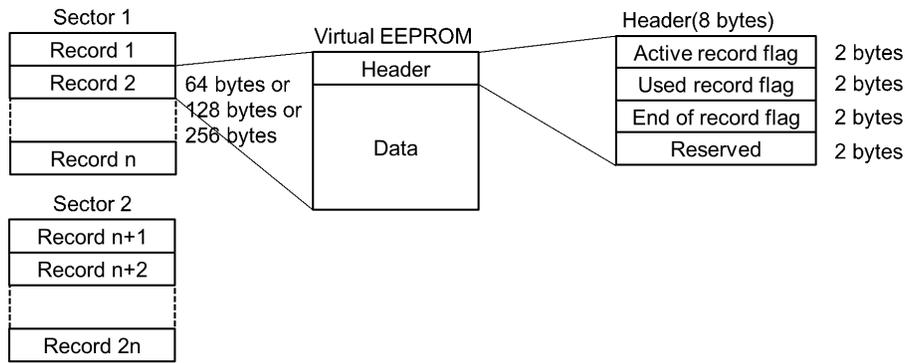


Figure 2-1. Structure of EEPROM Emulation

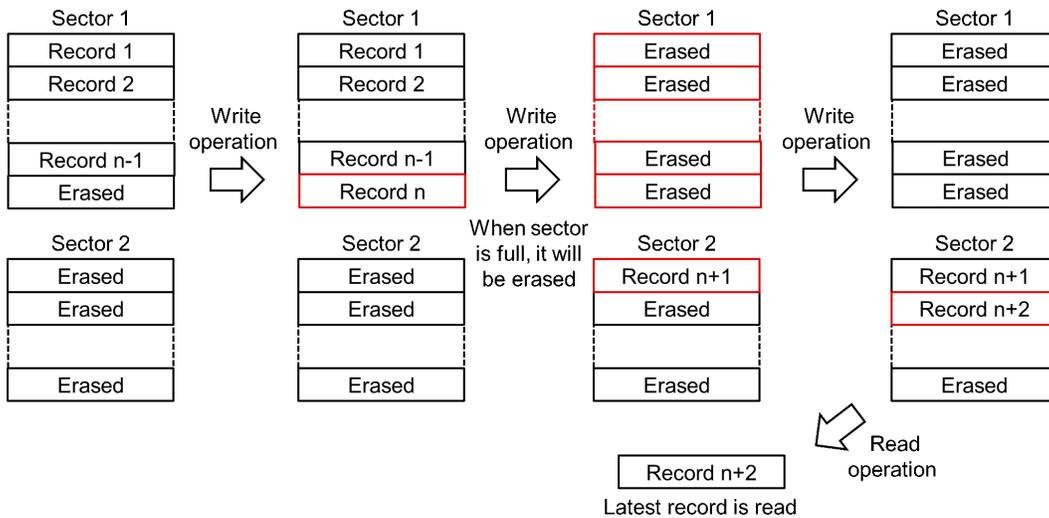


Figure 2-2. Basic Behaviors of EEPROM Emulation

Header

The header is designed to manage the record. By checking the header of a single record, the status of the record can be determined. By checking the headers of all records, the latest record can be found and the format of EEPROM emulation can be checked.

Each record has a header to show its status. The header is set to 8 bytes with 3 flags. Depending on the flags, there are four record status in total. The relationship between flags and record status is showed below.

Table 2-1. Relationship Between Flags and Record Status

Record Status	Active Record Flag	Used Record Flag	End of Record Flag
Erased	0xFFFF	0xFFFF	0xFFFF
Recording	0x0000	0xFFFF	0xFFFF
Active (latest)	0x0000	0xFFFF	0x0000
Used (Not latest)	0x0000	0x0000	0x0000

All flags are erased at first. When a new record is written, firstly active record flag is set and the status change to 'Recording'. Then the data is written to the record. The end of record flag is set only when the data is completely written. It monitors the completion of the write operation. If the system loses power while writing data, it will be detected on recovery that record status is 'Recording'.

After a new active record appearing, the old active record then changes to 'Used' by setting used record flag. Figure 2-3 shows how record status change when write operation is performed. It can be seen that there is always an active record, which helps to recover from power off.

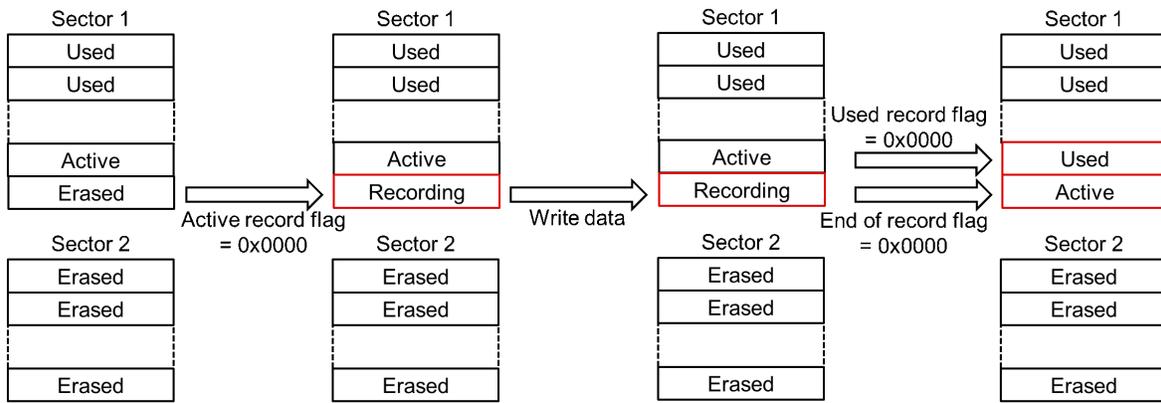


Figure 2-3. How Record Status Changes When Write Operation is Performed

3 Software Description

This software provides basic EEPROM functionality. At least 2 sectors are used to emulate the EEPROM. These sectors are broken into several records each containing header to determine the validity of the data as described above. To reduce the number of flash operations, a buffer in RAM of the same size as virtual EEPROM is used to copy data from active record. Besides, 4 global variables are used to trace the active record and 3 global variables are used for flags.

Software functionality and flow

In total, only three functions are called directly by the user.

- EEPROM_TypeA_init
- EEPROM_TypeA_writeData
- EEPROM_TypeA_eraseLastSector

The high-level software flow is shown in [Figure 3-1](#). The device should first go through the initialization code. By calling EEPROM_TypeA_init, it will search active record and check the format of Flash. If active record exists, the data of active will be copied to the buffer in RAM. If format is not correct, the format will be repaired. After initialization, there will be a properly formatted flash area for EEPROM emulation, several global variables tracing the active record, and a buffer in RAM which has copied the active record's data.

In the application, user can directly read or edit buffer in RAM. Only when EEPROM_TypeA_writeData is called, the buffer is stored into flash to be a new active record. EEPROM_TypeA_writeData will also set the erase flag when the sector is full. In the flow, EEPROM_TypeA_eraseLastSector is called immediately when the erase flag is set. According to the application, users can select the appropriate point in time to erase.

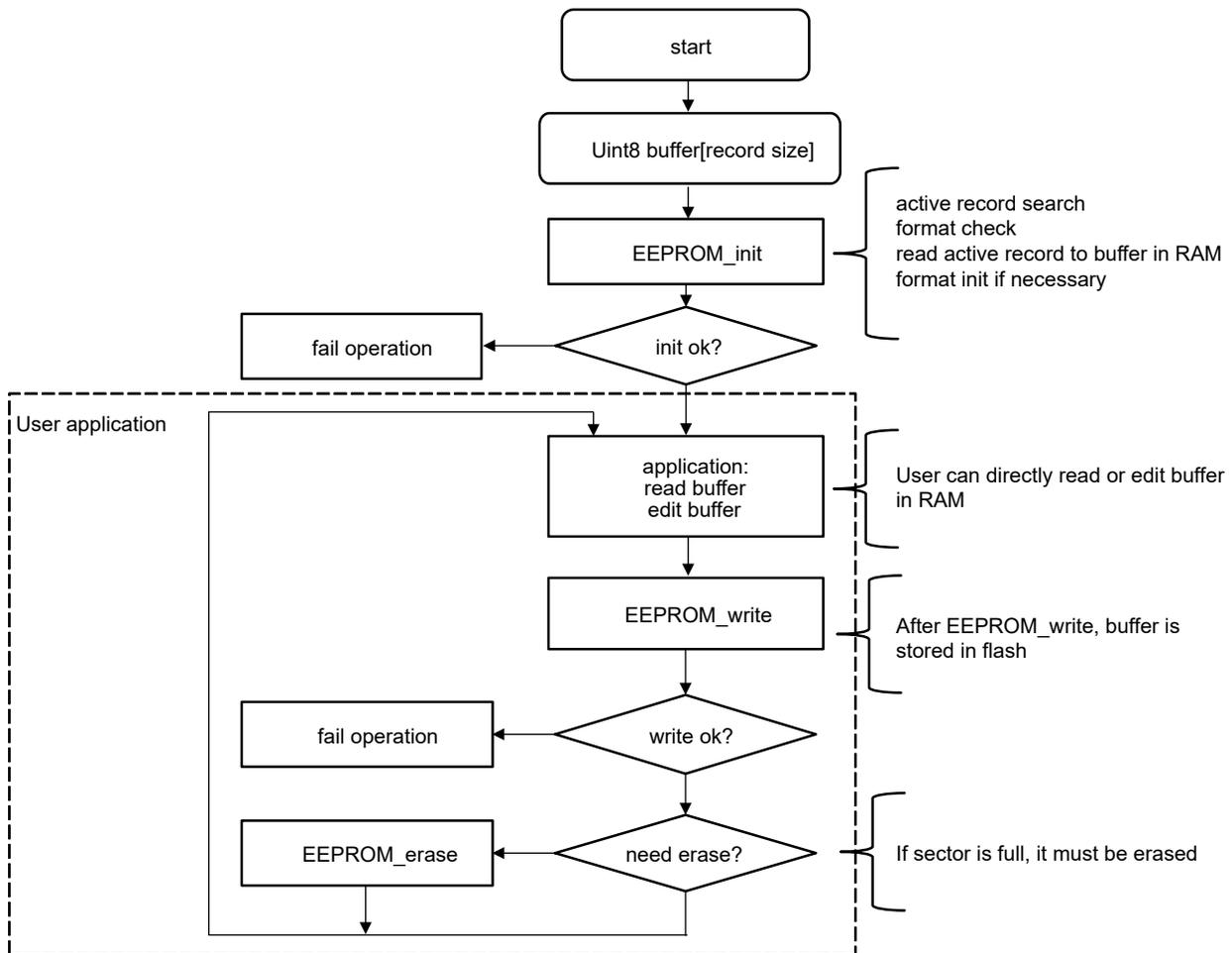


Figure 3-1. High-Level Software Flow

EEPROM Functions

To implement this functionality, six functions are required. In addition to the three functions mentioned above, the remaining three functions are called primarily by EEPROM_TypeA_init.

- EEPROM_TypeA_init
- EEPROM_TypeA_writeData
- EEPROM_TypeA_eraseLastSector
- EEPROM_TypeA_readData
- EEPROM_TypeA_searchCheck
- EEPROM_TypeA_repairFormat

Besides, 7 global variables are used to record the status of the EEPROM emulation. 4 global variables are used to trace the active record.

- uint32_t gActiveRecordAddress;
- uint32_t gNextRecordAddress;
- uint16_t gActiveRecordNum;
- uint16_t gActiveSectorNum;

gActiveRecordAddress and gNextRecordAddress are used to store the address about active record.

gActiveRecordNum and gActiveSectorNum are used to trace the position of active record.

3 global variables are used for flags.

- bool gEEPROMTypeASearchFlag;
- bool gEEPROMTypeAEraseFlag;
- bool gEEPROMTypeAFormatErrorFlag;

gEEPROMTypeASearchFlag is set when the active record exists.

gEEPROMTypeAEraseFlag is set when the sector is full and needs to be erased.

gEEPROMTypeAFormatErrorFlag is set When format error is found.

EEPROM_TypeA_init

The function is used to initialize the EEPROM emulation. Through this function, user's data is restored and active record in Flash is tracked. The function includes the following features:

- active record search and format check
- read active record to buffer in RAM
- format repair if necessary, by calling EEPROM_TypeA_repairFormat

The software flow is shown in [Figure 3-2](#). Firstly, it searches the active record and checks the format by calling EEPROM_TypeA_searchCheck. According to the flag set by EEPROM_TypeA_searchCheck, it can be determined whether active record exists or is format correct. If active record exists, the data of active record will be copied to the buffer in RAM by calling EEPROM_TypeA_readData, and the pointer is set to the active record. If active record does not exist, all sectors are erased and the pointer is set to the beginning of the first sector.

If format is not correct, format will be repaired by calling EEPROM_TypeA_repairFormat. After format repair, all sectors are erased and the active record is restored. Flash area comparison before and after EEPROM_TypeA_init is shown in [Figure 3-3](#) and [Figure 3-4](#).

The input of the function is the address of the buffer in RAM. The output of the function is the operation states. Besides, all 7 global variables are updated in the function.

- Input: &buffer[0]
- Output: operation state

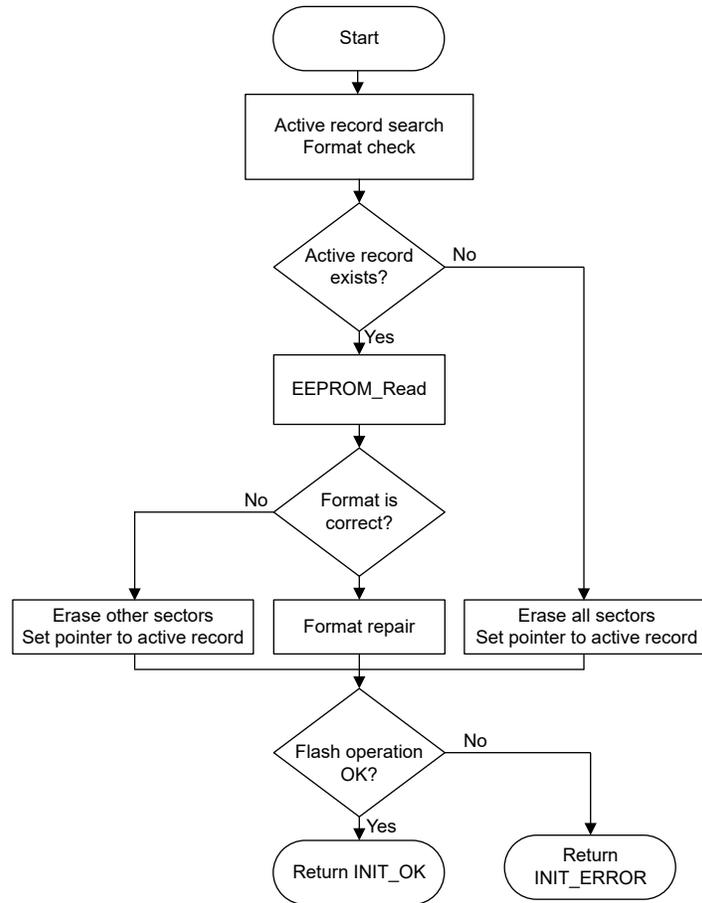


Figure 3-2. Software Flow of EEPROM_TypeA_init

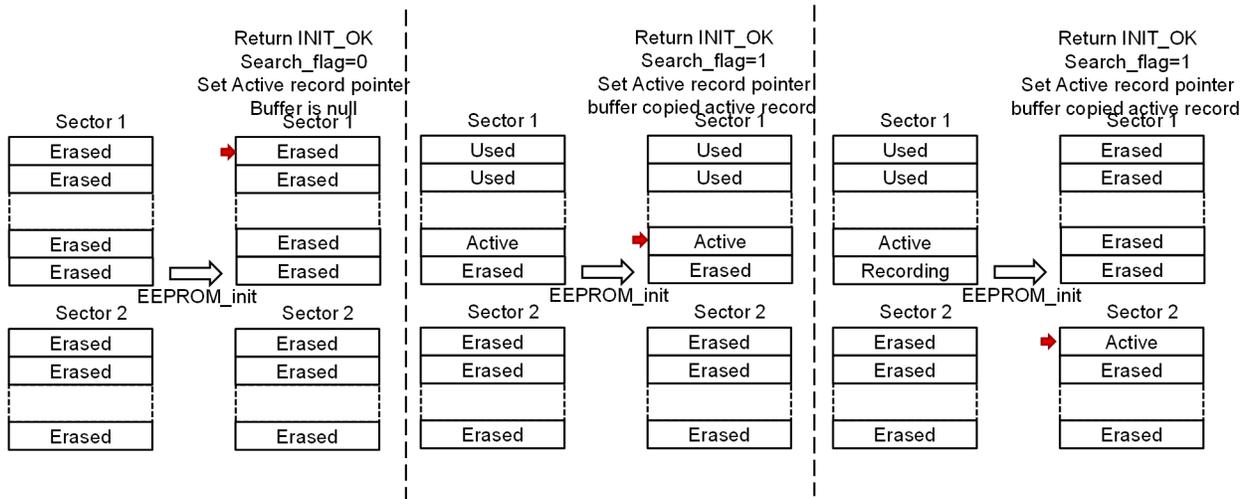


Figure 3-3. EEPROM_TypeA_init for the Normal Scene

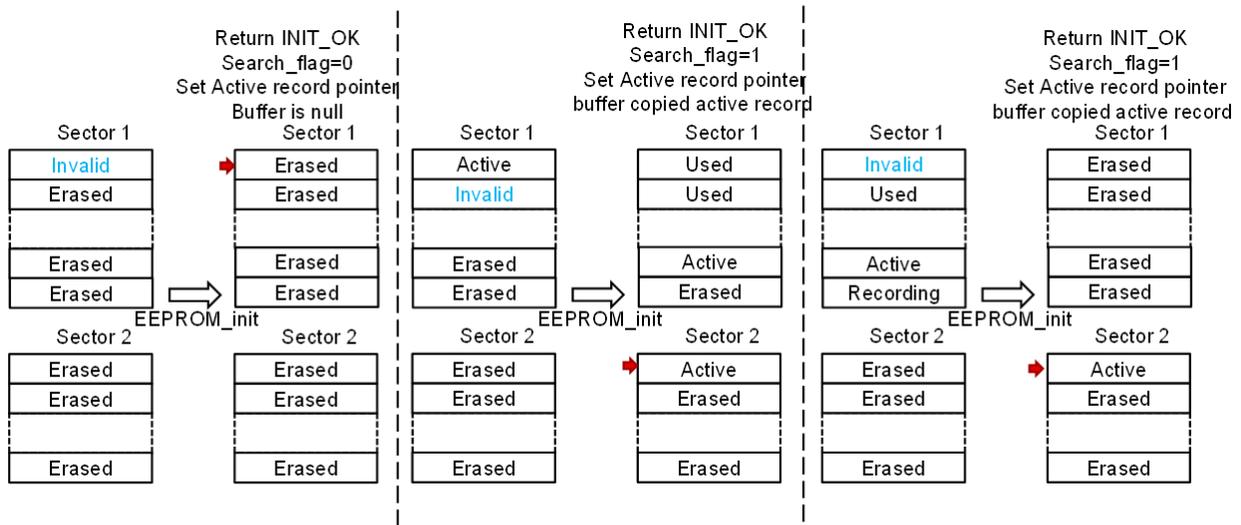


Figure 3-4. EEPROM_TypeA_init for the Abnormal Scene

EEPROM_TypeA_writeData

EEPROM_TypeA_writeData is used to store the data from buffer in RAM to Flash. Through the function, a new active record is added to the Flash.

Users can directly read or edit buffer in RAM. However, only after calling this function, buffer's data is copied to the new active record. In other words, Flash area is to record/backup the buffer. When power off, all data in RAM is lost, and the active record is used to recover the data.

The software flow is shown in [Figure 3-5](#). Firstly, it checks whether next record is erased. Then it begins storing data to next record. The process follows [Figure 2-3](#):

1. Set the record's header to 'Recording'
2. Copy the data from buffer in RAM to the record
3. Set the record's header to 'Active'
4. If last active record exists, set it header to 'Used'

Finally, it checks whether the sector is full. If so, gEEPROMTypeAEraseFlag will be set. Besides, the global variables about active record are updated. Flash area comparison before and after EEPROM_TypeA_writeData is shown in [Figure 3-6](#).

The input of the function is the address of the buffer in RAM. The output of the function is the operation states.

- Input: &buffer[0]
- Output: operation state

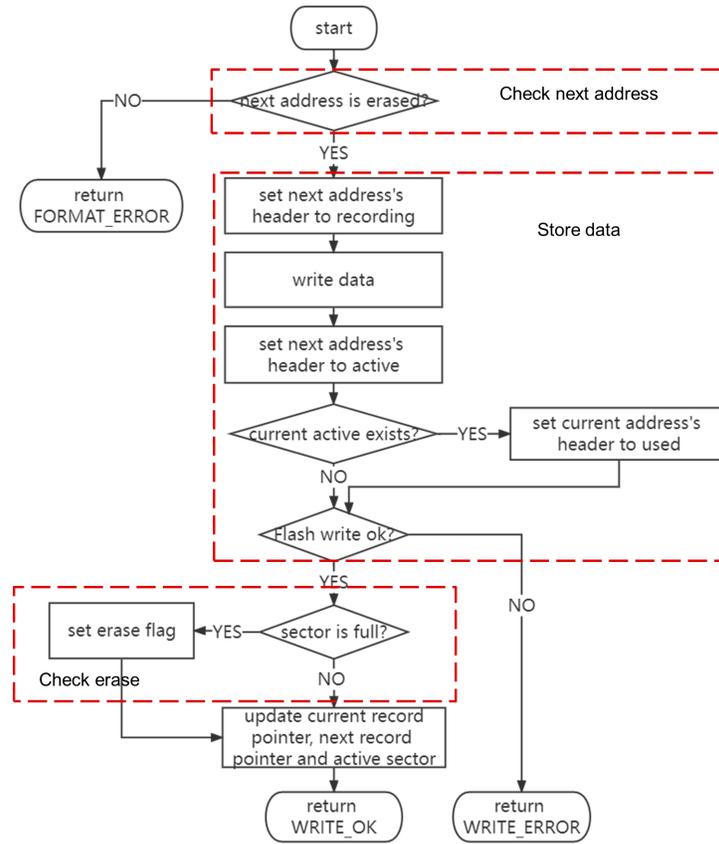


Figure 3-5. Software Flow of EEPROM_TypeA_writeData

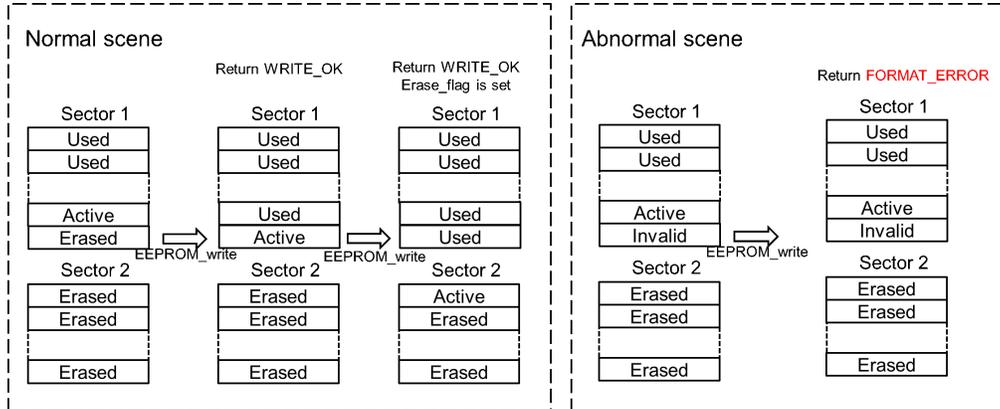


Figure 3-6. Software Flow of EEPROM_TypeA_writeData

EEPROM_TypeA_eraseLastSector

EEPROM_TypeA_eraseLastSector is used to erase the Flash sector when it is full. While calling EEPROM_TypeA_writeData, gEEPROMTypeAEraseFlag will be set if the sector is full. It is recommended to call the EEPROM_TypeA_eraseLastSector immediately when gEEPROMTypeAEraseFlag is set, as in Figure 3-1. However, users can change the time point to erase the sector by modifying the high-level software flow.

The output of the function is the operation states.

- Input: void
- Output: operation state

EEPROM_TypeA_readData

EEPROM_TypeA_readData is used to copy the data from active record in Flash to buffer in RAM. This function is called in EEPROM_TypeA_init. While users can directly read or edit buffer in RAM, EEPROM_TypeA_readData is not usually used directly by users.

The input of the function is the address of the buffer in RAM.

- Input: &buffer[0]
- Output: void

EEPROM_TypeA_searchCheck

EEPROM_TypeA_searchCheck is used to search the active record and check the format. The function traverses the headers of all records. If active record is found, search flag will be set, and global variables about active record will be updated. If there is a header of 'Recording' or an invalid header or other format-error situation, error flag will be set. The function is called in EEPROM_TypeA_repairFormat. The software flow is shown in [Figure 3-7](#).

The function outputs the result by setting global variables, so both the input and output of the function are void.

- Input: void
- Output: void

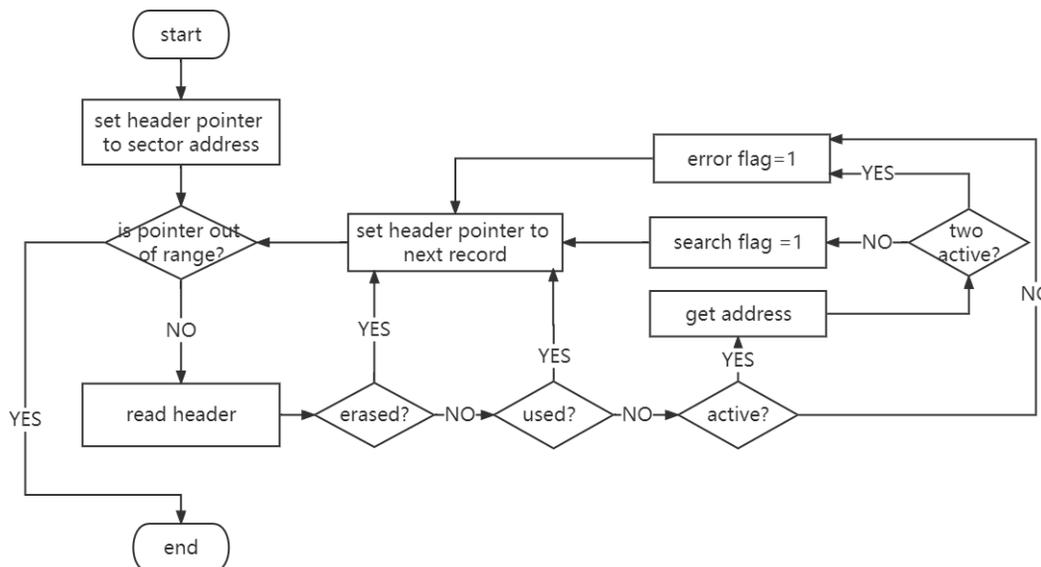


Figure 3-7. Software Flow of EEPROM_TypeA_searchCheck

EEPROM_TypeA_repairFormat

EEPROM_TypeA_repairFormat is used to repair the format. Before calling the function, active record should have been read to the buffer in RAM. Through the function, all sectors are erased and the data is copied from buffer in RAM to the new record in Flash. The function is called in EEPROM_TypeA_repairFormat. The software flow is shown in [Figure 3-8](#).

The input of the function is the address of the buffer in RAM. The output of the function is the operation states.

- Input: &buffer[0]
- Output: operation state



Figure 3-8. Software Flow of EEPROM_TypeA_repairFormat

Application Integration

Applications requiring this functionality need to include the `eeeprom_emulation_type_a.c` and `eeeprom_emulation_type_a.h` files provided for MSPM0 MCUs. The Flash API also needs to be included for the specific device. For example, for the MSPM0G3507/MSPM0L1306 the following files are needed:

- `eeeprom_emulation_type_a.c`
- `eeeprom_emulation_type_a.h`
- `ti_msp_dl_config.c`
- `ti_msp_dl_config.h`

EEPROM emulation library has been included in the SDK supporting MSPM0 products.

All Flash API files are also included in the SDK.

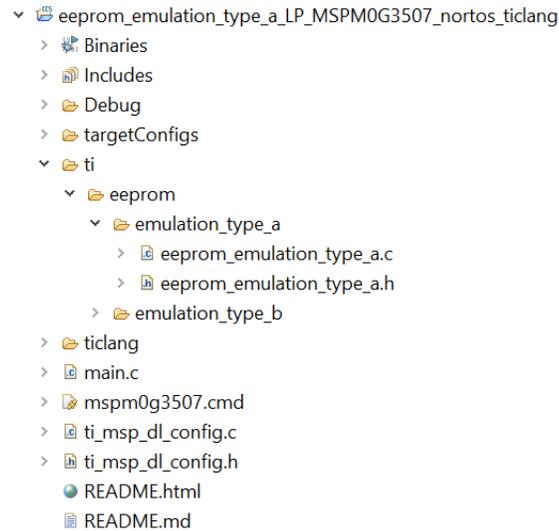


Figure 3-9. Files Required by the Software

EEPROM emulation memory footprint

Table 3-2 details the footprint of the EEPROM emulation driver in terms of flash size and RAM size. The table and figure below have been determined using the Code Composer Studio (Version: 11.2.0.00007) with optimization level 2.

Table 3-1. Structure of EEPROM Emulation

Mechanism	Minimum Required Code Size (bytes)	
	Flash	SRAM
EEPROM emulation type A with 64 bytes record size	2616	71
EEPROM emulation type A with 128 bytes record size	2616	135
EEPROM emulation type A with 256 bytes record size	2616	273

EEPROM emulation timing

This section describes the timing parameters associated with the EEPROM emulation driver based on two 1KB flash sectors.

All timing measurements are performed with these conditions:

- MSPM0G3507
- System clock at 32 MHz
- With execution from flash
- At room temperature

The functions are tested with these parameters:

- Record size: 128 bytes
- Number of sectors used: 2
- Sector address: 0x00001000

Table 3-2. Timing of EEPROM Emulation Operations

Operation	Typical (μ s)
EEPROM_TypeA_init with correct format	80
EEPROM_TypeA_init with format repair	4467
EEPROM_TypeA_writeData	848
EEPROM_TypeA_eraseLastSector	3612

4 Application Aspects

This section describes the application-level features of EEPROM emulation solution and how to configure it to meet the application needs.

Selection of configurable parameters

There are three user-configurable parameters in `EEPROM_Emulation_TypeA.h`. These parameters can be configured accordingly, depending on the requirements of the application.

- Record size: 64, 128, or 256 bytes
- Number of sectors used: at least 2
- Sector address

Memory Space Utilization

Although the header takes up 8 bytes of space in the structure, memory space utilization depends on the user's application.

$$\text{Memory space utilization} = \frac{\text{User's data size}}{\text{Record size}}$$

$$\text{The largest memory space utilization} = \frac{\text{Record size} - \text{Header size}}{\text{Record size}}$$

It is recommended to evaluate the virtual EEPROM size required for the application before selecting the appropriate record size. For example, if there are 40 bytes of data to be stored, it is recommended to select the record size of 64 bytes.

Cycling Capability

Flash endurance is typically 10000 cycles, far less than real EEPROM. One feature of EEPROM emulation is that its endurance is extended compared to flash. By dividing the flash sector into multiple records and writing data one by one, the flash sector does not need to erase each time it writes, but only after it is full. And the equivalent endurance is further enhanced by the use of multiple flash sectors.

$$\text{Number of records} = \frac{\text{Sector size}}{\text{Record size}} \times \text{Number of sectors}$$

$$\text{Equivalent endurance} = \text{Number of records} \times \text{flash endurance}$$

For example, if there are 2 1KB sectors used and record size is 128 bytes, equivalent endurance is 160k cycles. If there are 3 1KB sectors used and record size is 128 bytes, equivalent endurance is 240k cycles. If there are 2 1KB sectors used and record size is 64 bytes, equivalent endurance is 320k cycles.

It is recommended that the user evaluate the cycling capability required by the application before selecting the appropriate number of sectors to use.

For three user-configurable parameters, a recommended design process is as follows:

1. Evaluate the data size required by application and select the appropriate record size.
2. Evaluate the cycling capability required by application, and select the appropriate number of sectors to use.
3. Select the appropriate flash address

For example, if there is an application which needs to update data in EEPROM every 10 minutes. The data size is 40 bytes and 10 years of continuous service is guaranteed. Firstly, record size could be 64 bytes. Secondly, cycling capability required by application is 525600 cycles (10 years × 365 days × 24 hours × 6 cycles per hour). 4 sectors are needed and the equivalent endurance is 640000 cycles.

Recovery in case of power loss

Data or header corruption is possible in case of a power loss during a `EEPROM_TypeA_writeData` or `EEPROM_TypeA_eraseLastSector`.

To detect the corruption and recover from it, `EEPROM_TypeA_init` is implemented. It should be called immediately after power-up. `EEPROM_TypeA_init` checks all records' header to confirm whether data storage of EEPROM emulation is correct, and performs format-repair if necessary.

In the structure of EEPROM emulation, headers show the status of corresponding records. There are 4 states in total. The changes between the four states are described in detail in the previous section.

5 References

1. [EEPROM Emulation Using Low-Memory MSP430™ FRAM MCUs](#)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#) or other applicable terms available either on [ti.com](https://www.ti.com) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2023, Texas Instruments Incorporated