AN0002
Application Note

EEPROM Emulation

Introduction

EEPROMs (Electrically Erasable Programmable Read-Only Memory) are often used in many applications to store data. For low-cost purposes, some of AT32 MCUs are not equipped with EEPROM. This application note describes a software solution for substituting an EEPROM by emulating the EEPROM mechanism using the on-chip Flash of AT32 devices.

Note: The corresponding code in this application note is developed on the basis of V2.x.x BSP provided by Artery. For other versions of BSP, please pay attention to the differences in usage.

Applicable products:

Part number	All series

Contents

List of Tables

List of Figures

1 Embedded FLASH and EEPROM

FLASH and EEPROM are non-volatile memories that can store data for a long time after power-off, which provides conditions for EEPROM emulation by using on-chip FLASH. Features and differences of FLASH and EEPROM are summarized in the table below.

Table 1. Features of FLASH and EEPROM

Feature	FLASH	EEPROM
Similarities	Non-volatile memory that can still store data after power-off	Non-volatile memory that can still store data after power-off
Differences	Flash needs to be erased before writing data (write bit from 1 to 0 only).	EEPROM is not erased before writing data (write bit from 1 to 0 or from 0 to 1)
	Flash is erased by sector.	There is no sector in EEPROM.
	Large capacity, low-cost	Small capacity, high-cost
	Erase life cycle is about 100,000 times.	Erase life cycle is about 1,000,000 times.
	Data retention time is more than 20 years.	Data retention time is more than 100 years.

Advantages of using FLASH for EEPROM emulation:

- Low-cost: no need to equip an additional EEPROM chip;
- Rapid storage and read access: communication speed is faster than that of EEPROM components using I^2C or SPI;
- Strong anti-interference ability: Flash is embedded in the MCU, and therefore the communication bus will not be interfered;
- Adjustable capacity: the storage size can be adjusted according to the actual application.

2 Principle of emulating EEPROM

2.1 EEPROM data structure

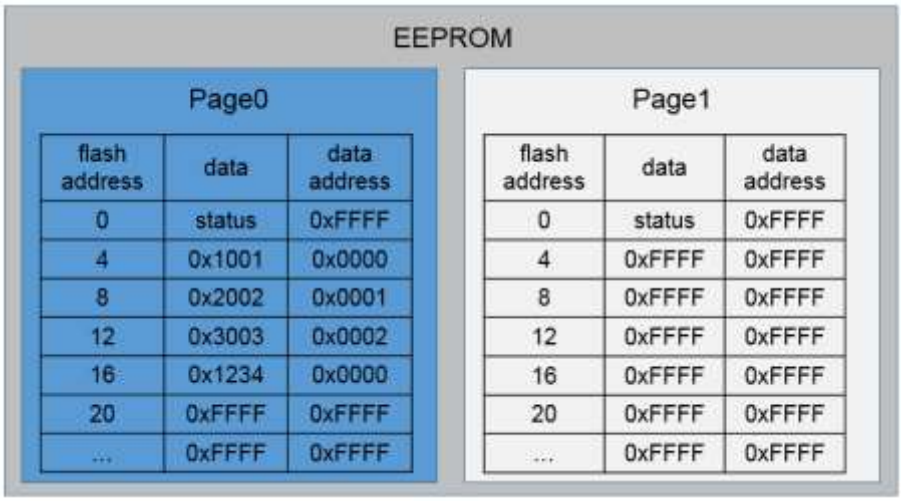
It is necessary to erase FLASH and set to 0xFF before writing data, and the FLASH is generally erased by sector (for example, the sector of AT32F403A series is 2 Kbytes). This feature determines that users cannot simply erase data and then write new data; otherwise, other data in this sector will be erased, and the FLASH will be erased frequently and its service life will be reduced.

The principles of emulating EEPROM are as follows:

- New data storage does not affect the old data;
- Minimize the FLASH erase operations to prolong its service life.

Based on the above principles, we designed a memory structure as shown below:

Figure 1. EEPROM data storage structure



EEPROM structure

EEPROM consists of page0 and page1. When EEPROM is being used, one page is in valid state, and the other page is in erased state. Read access or data writing is performed on the valid page.

Data format

The stored data is in “data + data address” format. Both address and data are stored in 16-bit mode, and each storage takes up 32 bits or 4 bytes. As shown in the figure above, the data column lists related data, the data address column lists corresponding data address, and the flash address column lists the actual flash address offset. For example, when flash address=12 in page0, the data is 0x3003, and the data address is 0x0002.

Page status flag

The page status flag is stored in the first data storage area. There are three page status flags:

- Valid: EE_PAGE_VALID, status = 0x0000; read access and data writing are performed on this page;
- Data Transfer: EE_PAGE_TRANSFER, status = 0xCCCC; this page is receiving data from the other full page;
- Erased: EE_PAGE_ERASED, status = 0xFFFF.

Data writing

Before writing data each time, search from the page start address to find the first area (value: 0xFFFFFFFF) where no data is stored, and then write the data and data address to be written to this area. For example, at flash address = 20 in page 0, the value is 0xFFFFFFFF, and thus it is the first area where no data is stored.

When the page size is obtained, user can calculate the maximum number of variables to be stored through the formula of “page size/4-1”. For example, when the page size is 1 K, the maximum number of variables is 1024/4-1=255. It should be noted that, in actual use, more free capacity should be reserved as much as possible to reduce FLASH erase operations and prolong its service life.

In addition, the data address should not exceed the maximum number of variables that can be stored. For example, when the page size is 1 K, the maximum number of variables is 1024/4-1=255, and the data address cannot exceed 255.

Data reading

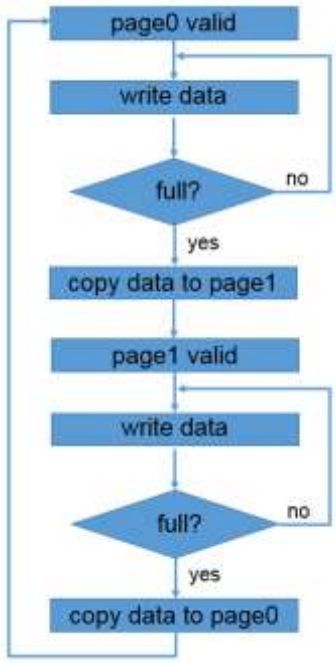
When reading data each time, search forward to the last stored valid data from the page end address. For example, when reading the data where the address is 0x0000 (as shown in figure 1, the address is 0x0000 at flash address = 4 and flash address = 16), the last stored data is the data at flash address = 16; therefore, the data being read is 0x1234.

Data transfer

When a page is full, the data is transferred to other free pages as follows (page0: full, page1: free):

- Set page1 to Data Transfer status (EE_PAGE_TRANSFER);
- Copy all valid data to page1;
- Erase page0;
- Set page1 to Valid status (EE_PAGE_VALID).

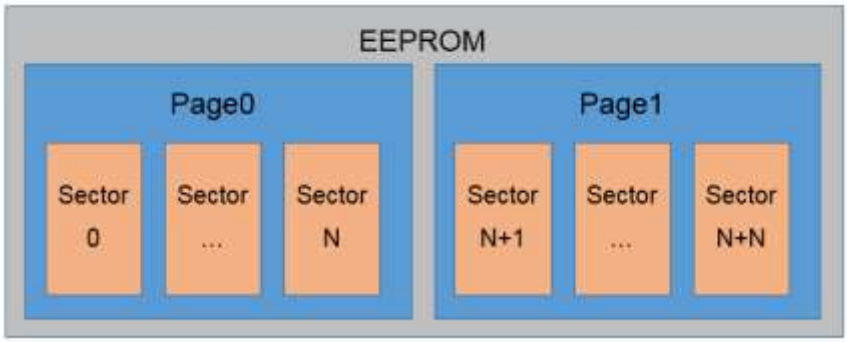
Figure 2. EEPROM data writing process



2.2 EEPROM structure

The structure of emulated EEPROM for AT32 MCUs is shown below. The EEPORM consists of one or more sectors according to the actual use. The more sectors the EEPORM have, the more data can be stored. Generally, the EEPROM storage area is defined at the end of the entire FLASH, so that programming and execution of program do not affect the EEPROM area.

Figure 3. EEPROM structure



Sectors can be configured through macros in project\at_start_f403a\eeprom\inc\eeprom.h:

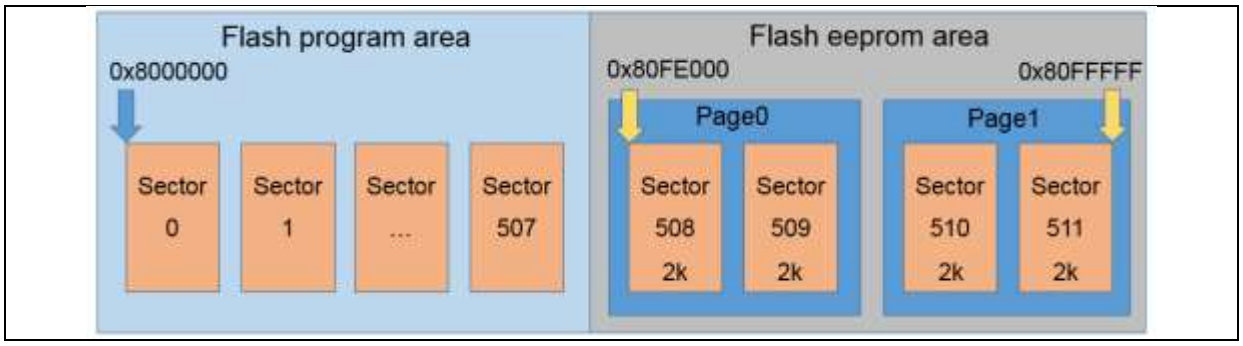
- EE_SECTOR_NUM: define the number of sectors of a single page.
- EE_SECTOR_SIZE: define the sector size, in terms of byte; sectors of different AT32 series are different in size, as shown in the table below.
- EE_BASE_ADDRESS: define EEPROM sector start address, and the EEPROM is usually placed at the end (calculated automatically and placed at the end, no user configuration required).

Table 2. Sector sizes of different AT32 series

Part number	Sector size
AT32F403	2K
AT32F403A	2K
AT32F413	2K
AT32F415	2K
AT32F435	4K
AT32F421	1K
AT32F425	1K
AT32L021	1K

For AT32F403A 1024K Flash, EEPROM is defined as follows:

Figure 4. EEPROM definition



As shown in the figure above, the FLASH consists of 512 sectors with the total capacity of 1024 K. Sectors 0~507 (1016 K) are used for program storage, and sectors 508~511 (8 K) are used for EEPROM. The EEPROM is stored in the last four sectors, and therefore the EEPROM sector start address EE_BASE_ADDRESS is defined as 0x80FE000 (the EE_BASE_ADDRESS is automatically calculated and placed at the end of FLASH, and users do not need to configure it).

Each sector is 2 K; therefore, the total sector size EE_SECTOR_SIZE is defined as 2048.

Each page contains two sectors; therefore, the sector number EE_SECTOR_NUM is defined as 2.

In this case, the size of a single page is 4 K; therefore, the maximum number of variables to be stored is 4096/4-1=1023. Note that 1023 is the theoretical maximum number, and more free capacity should be reserved as much as possible to reduce FLASH erase operations and prolong its service life.

3 How to use EEPROM

3.1 Initialization state machine

EEPROM is managed through the status flags of page0 and page1. When one page is full, valid data will be copied to a new page, and power-off or MCU reset may occur during this process. When the MCU restarts, it can continue to work only after the previous operations are completed. Therefore, related initialization operations must be executed according to the status flag value before using the EEPROM. The initialization state machine is encapsulated into the flash_ee_init() function that users can call directly.

Table 3. EEPROM initialization state machine

		Page1		
		VALID	TRANSFER	ERASE
Page0	VALID	Erase page0 Set page0 as VALID Erase page1	Erase page1 Copy data from page0 to page1 Erase page0 Set page1 as VALID	Erase page1
	TRANSFER	Erase page0 Copy data from page1 to page0 Erase page1 Set page0 as VALID	Erase page0 Set page0 as VALID Erase page1	Erase page1 Set page0 as VALID
	ERASE	Erase page0	Erase page0 Set page1 as VALID	Erase page0 Set page0 as VALID Erase page1

3.2 Function interface

The emulated EEPROM by using FLASH provides three functions, i.e., initialization, write data and read data.

1) EEPROM initialization: this function should be called for initialization every time the MCU is reset.

```
flash_status_type flash_ee_init(void);
```

- Return value: Flash operation status

2) Read data from EEPROM

```
uint16_t flash_ee_data_read(uint16_t address, uint16_t* pdata);
```

- address: the address of variables
- pdata: the data being read
- Return value: data read status 0: the data is read successfully; 1: data is not found.

3) Write data to EEPROM

```
flash_status_type flash_ee_data_write(uint16_t address, uint16_t data);
```

- address: the address of variables

- data: the data being written
- Return value: Flash operation status

Note that the address value should not be larger than the number of variables. For example, when the page size is 2 K, and the maximum number of variables to be stored is 2048/4-1=511. In this case, the address value is within the range of 0~511.

4 Direct data storage mode

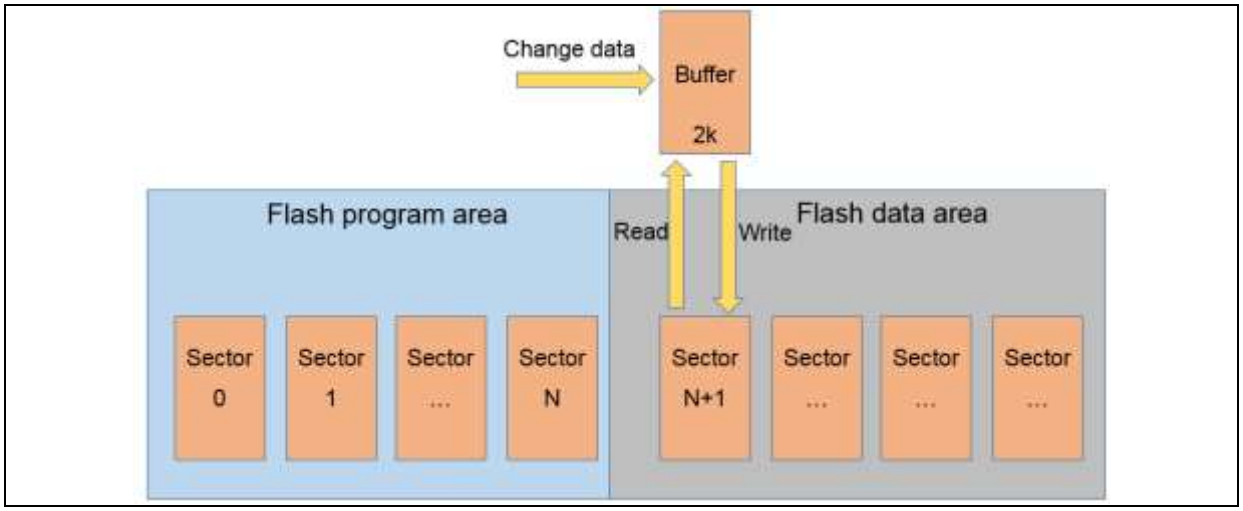
4.1 Principle of storage

The emulated EEPROM using FLASH as described in the previous chapters has the advantages of convenient use and reliable storage when storing a small amount of data. However, it is not suitable when storing a large amount of data or performing storage access to any FLASH address.

Therefore, we provided a solution to allow users to access FLASH directly to store data. As shown in the figure below, allocate an area behind FLASH for data storage, and users can store or read data at any address in this area. The FLASH is erased by sector (1 K, 2 K or 4 K, as shown in Table 2); therefore, users cannot write data to sectors directly. Before writing data, set a buffer with the same size as the sector; then read data from the sector and modify it in the buffer; finally, erase the sector and write the data to this sector.

However, this solution is only suitable for storing non-critical data (such as operating logs) for the reason that the data stored in sector may be lost when power-off or MCU reset occurs during the process of reading data to the buffer and erasing the sector. By contrast, the emulated EEPROM using FLASH is better for storing critical data.

Figure 5. FLASH storage structure

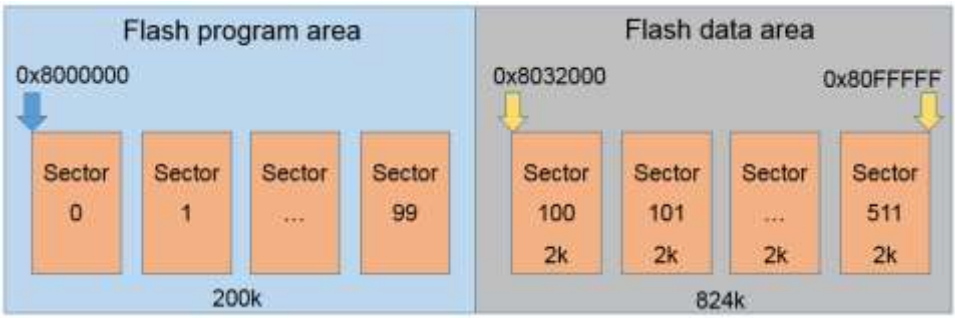


Sectors can be configured through macros in project\at_start_f403a\flash_write_read\inc\flash.h.

- FLASH_SECTOR_SIZE: define the sector size, in terms of byte; sectors of different AT32 series are different in size, as shown in the Table 2.
- FLASH_CODE_SIZE: define the size of program storage area, in terms of byte.

Define a data area using the AT32F403A 1024K Flash, as shown below:

Figure 6. FLASH storage definition



As shown in the figure above, FLASH contains 512 sectors with the total capacity of 1024 K. Sectors 0~99 are 200 K for program storage, and sectors 100~511 are 824 K for data storage.

Therefore, the FLASH_SECTOR_SIZE is 2048, and the FLASH_SECTOR_SIZE is 1024*200.

4.2 Function interface

FLASH direct data access mode provides two functions, i.e., write data and read data.

1) Read data from FLASH

```
void flash_read(uint32_t address, uint16_t *pdata, uint32_t number);
```

- address: the address of FLASH, which is located in the data storage area and aligned with two bytes (for example, it can be configured as 0x8032002 and 0x8032004, but not 0x8032003)
- pdata: the data being read
- number: the number of data being read

2) Write data to FLASH

```
void flash_write(uint32_t address, uint16_t *pdata, uint32_t number);
```

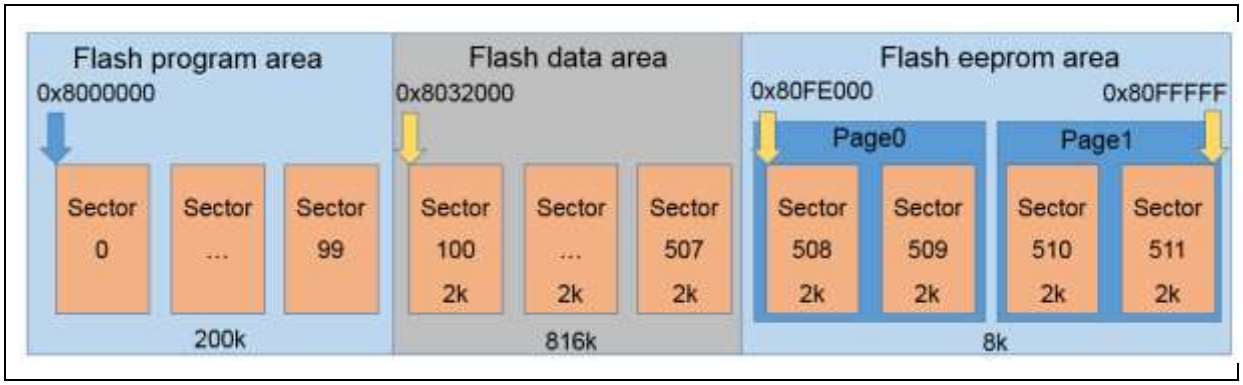
- address: the address of FLASH, which is located in the data storage area and aligned with two bytes (for example, it can be configured as 0x8032002 and 0x8032004, but not 0x8032003)
- data: the data being written
- number: the number of data being written

5 Combination of emulated EEPROM and direct data storage mode

The emulated EEPROM using Flash and direct data storage mode can be used in combination to realize reliable storage of critical data as well as large amount of data.

Define the storage area using the AT32F403A 1024K Flash, as shown below:

Figure 7. Combined storage modes



As shown in the figure above, FLASH contains 512 sectors with the total capacity of 1024 K. Sectors 0~99 are 200 K for program storage; sectors 100~507 are 816 K for data storage, and sectors 508~511 are 8 K for EEPROM.

6 Case 1: EEPROM data read/write

6.1 Function overview

Demonstrate how to write data to and read data from EEPROM.

This sample program is commonly used in all AT32 MCUs. When transplanting between different models, users only need to modify the EE_SECTOR_NUM and EE_SECTOR_SIZE. For details, please refer to Section 2.2.

6.2 Resources preparation

1) Hardware environment

   AT32F403A AT-START BOARD

2) Software environment

   EEPROM_Emulation_V2.0.0\project\at_start_f403a\eeprom

Note: All projects are built around keil 5. If users want to use them in other compiling environments, please refer to AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates (such as IAR6/7, keil 4/5) for a simple change.

6.3 Software design

1) Configuration process
   - Initialize EEPROM
   - Write data to EEPROM
   - Read data from EEPROM
   - Compare and check whether the data read and the data written are equal
2) Code
   - main function code

```
int main(void)
{
  uint16_t i, address;

  /* Initialize system clock */
  system_clock_config();

  /* Initialize START board */
  at32_board_init();

  /* Initialize flash eeprom */
  flash_ee_init();

  /* Write data to eeprom */
  for(i = 0; i < BUF_SIZE; i++)
  {
    address = i;

    flash_ee_data_write(address, buf_write[i]);
```

```
    }

    /* Read data from eeprom */
    for(i = 0; i < BUF_SIZE; i++)
    {
        address = i;

        flash_ee_data_read(address, &buf_read[i]);
    }

    /* Compare and check whether the data read and the data written are equal */
    if(buffer_compare(buf_write, buf_read, BUF_SIZE) == 0)
    {
        at32_led_on(LED3);
    }
    else
    {
        at32_led_off(LED3);
    }

    while(1)
    {
    }
}
```

6.4 Test result

- If the data read and the data written are equal, LED3 will be ON; otherwise, LED3 will be OFF.

7 Case 2: Direct data storage

7.1 Function overview

Demonstrate how to write data to and read data from FLASH directly.

This sample program is commonly used in all AT32 MCUs. When transplanting between different models, users only need to modify the FLASH_CODE_SIZE and FLASH_SECTOR_SIZE. For details, please refer to Section 4.1.

7.2 Resources preparation

1) Hardware environment

   AT32F403A AT-START BOARD

2) Software environment

   EEPROM_Emulation_V2.0.0\project\at_start_f403a\flash_write_read

Note: All projects are built around keil 5. If users want to use them in other compiling environments, please refer to AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates (such as IAR6/7, keil 4/5) for a simple change.

7.3 Software design

1) Configuration process
   - Write data to FLASH
   - Read data from FLASH
   - Compare and check whether the data read and the data written are equal
2) Code
   - main function code

```
int main(void)
{
  /* Initialize system clock */
  system_clock_config();

  /* Initialize START board */
  at32_board_init();

  /* Write data to FLASH */
  flash_write(0x8000000 + 1024 * 256, buf_write, BUF_SIZE);

  /* Read data from FLASH */
  flash_read(0x8000000 + 1024 * 256, buf_read, BUF_SIZE);

  /* Compare and check whether the data read and the data written are equal */
  if(buffer_compare(buf_write, buf_read, BUF_SIZE) == 0)
  {
    at32_led_on(LED3);
  }
  else
  {
```

```
        at32_led_off(LED3);
    }

    while(1)
    {
    }
}
```

7.4 Test result

- If the data read and the data written are equal, LED3 will be ON; otherwise, LED3 will be OFF.

8 Revision history

Table 4. Document revision history

Date	Version	Revision note
2021.12.15	2.0.0	Initial release
2022.09.07	2.0.1	Modified Table 1.