Application Note

Application Note of APM32F4xx Flash Simulating EEPROM

Document No.: AN1094

Version: V1.0

1 Introduction

This application note is used to introduce the implementation of applying internal Flash simulation EEPROM function on the APM32F4xx series, including hardware design and software design methods. For information about chips and registers, please refer to the *User Manual* and *Datasheet* on the official website.

This application note takes APM32F407ZE model of APM32F4xx series as an example.

Contents

2 Introduction to APM32F4xx FLASH Memory

FLASH memory, also called flash memory, is usually used to store program codes, data and other information. In the APM32F4xx series, the system can be booted from Flash memory by configuring the BOOT0 to connect to GND pin. When the main flash memory is mapped to the boot space, it can still be accessed at its original address, that is, the contents of the flash memory can be accessed in two address areas.

2.1 FLASH Memory Structure Block Diagram

The Flash memory structure of APM32F4xx mainly consists of main storage block and information block. In the information block, there are system storage blocks, OTP areas, and option bytes, as shown in Figure 2-1:

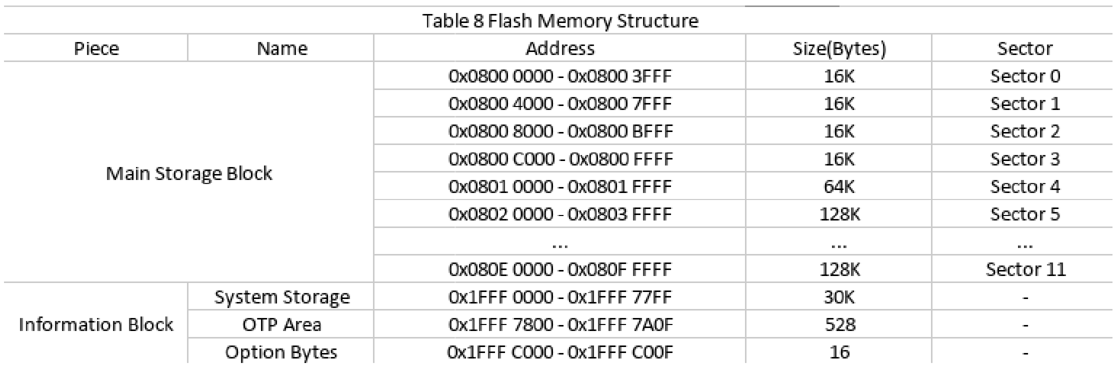


Table 8 Flash Memory Structure

Piece	Name	Address	Size(Bytes)	Sector
Main Storage Block		0x0800 0000 - 0x0800 3FFF	16K	Sector 0
		0x0800 4000 - 0x0800 7FFF	16K	Sector 1
		0x0800 8000 - 0x0800 BFFF	16K	Sector 2
		0x0800 C000 - 0x0800 FFFF	16K	Sector 3
		0x0801 0000 - 0x0801 FFFF	64K	Sector 4
		0x0802 0000 - 0x0803 FFFF	128K	Sector 5
		...	...	...
		0x080E 0000 - 0x080F FFFF	128K	Sector 11
Information Block	System Storage	0x1FFF 0000 - 0x1FFF 77FF	30K	-
	OTP Area	0x1FFF 7800 - 0x1FFF 7A0F	528	-
	Option Bytes	0x1FFF C000 - 0x1FFF C00F	16	-

Figure 2-1 FLASH Memory Structure Block Diagram

2.2 Introduction to FLASH Memory Structure Block Diagram

Main storage block: This section is mainly used to store application code, boot program code, and data constants. It is mainly divided into 12 sectors. The first four sectors are 16k, the sector 4 is 64k, and the sectors 5-11 are 128k. The start address of the main storage block is 0x08000000, from which the code starts running.

System storage block: This section is mainly used to store BootLoader code, which is used to initialize the boot initialization phase of the system, initiate the hardware before loading the operating system, and load the operating system.

OTP area: One-time programmable (storage area), which means that once the data is written, it cannot be modified. This area has a total of 528 bytes and is divided into two areas. One is divided into 512 bytes, which can be used to store user data, and the other is divided into 16 bytes, which can be used to lock corresponding blocks.

Option byte: It is used to configure read/write protection for FLASH, BOR level in power management, software/hardware watchdog, and other functions.

3 Principle of APM32F4xx FLASH simulating EEPROM

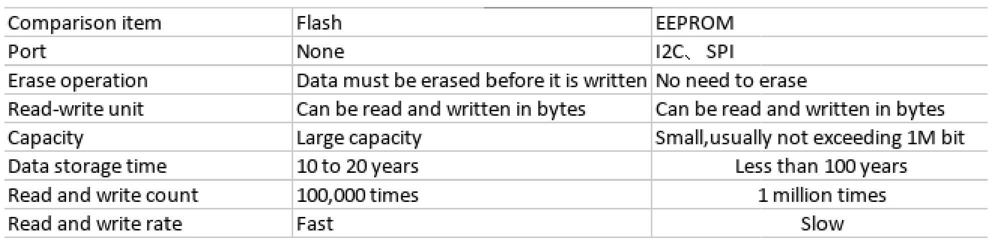
Since there is no EEPROM memory in the APM32F4xx series, and the Flash memory and RAM memory are built in the APM32F4xx series, Flash memory can be combined with RAM memory to simulate EEPROM, and produce the same effect as read-write EEPROM memory in terms of user experience.

3.1 Introduction to EEPROM

EEPROM, Electrically Erasable Programmable Read Only Memory, is a non-volatile storage area. EEPROM can read and write data by bytes, but data does not need to be erased before writing.

3.2 Comparison between APM32F4xx Flash and EEPROM

Flash and EEPROM are both non-volatile memory and can both save data in case of power-down. But they also have many differences, and Figure 3-1 lists the main differences between internal Flash of APM32F4xx chip and EEPROM:



Comparison item	Flash	EEPROM
Port	None	I2C、 SPI
Erase operation	Data must be erased before it is written	No need to erase
Read-write unit	Can be read and written in bytes	Can be read and written in bytes
Capacity	Large capacity	Small,usually not exceeding 1M bit
Data storage time	10 to 20 years	Less than 100 years
Read and write count	100,000 times	1 million times
Read and write rate	Fast	Slow

Figure 3-1 Comparison between Flash and EEPROM

3.2.1 Advantages of APM32F4xx Flash simulating EEPROM

1. The cost is low. Using internal Flash to simulate EEPROM can reduce the use of EEPROM storage chips and save costs.

2. The read and write speed is fast. The read and write speed of the internal Flash of the MCU is higher than that of EEPROM, and it can read a large

amount of data in a short period of time.

3. The capacity is large. The Flash memory has a larger capacity than EEPROM memory, and can store more data and programs.

4. The anti-jamming capability is strong. The internal Flash of MCU does not use external communication bus interfaces such as I2C and SPI, and there is no problem of interference of external communication bus.

3.3 Implementation of APM32F4xx FLASH simulating EEPROM

3.3.1 Implementation steps

Write operation:

1. Create a buffer with the same size as the sector in RAM.

2. Read the sector data to be written into the cache.

3. Overwrite the data in the cache.

4. Erase the sector (before erasing the sector, a judgment will be made. If it is already in the erased state, there is no need to erase it again).

5. Write the data to the sector.

Read operation:

1. Similar to the operation of reading the memory. For example, to read 0x8000000 data: *(uint32_t*)0x80000000.

3.3.2 Implementation difficulties and solutions

1. In the APM32F4xx series, the minimum sector of Flash is 16KB, the maximum sector is 128KB, and the maximum RAM size available to the user is 128KB. In the process of simulating EEPROM in Flash, in order not to change the existing data when erasing Flash, it is necessary to create a RAM space with the same size as the Flash sector as the cache when writing or erasing the Flash. If a sector size of 128KB is selected as the space for Flash to simulate EEPROM, there is no enough RAM size to create a cache for Flash to simulate the EEPROM.

2. In the APM32F4xx series, the first four sectors are 16KB, so it is suitable for

Flash to simulate the EEPROM. However, among these four sectors, for the first sector, when the MCU is powered on, it will fetch instructions from this sector to run the program. If there are no valid CPU instructions stored in this area, the program will run abnormally.

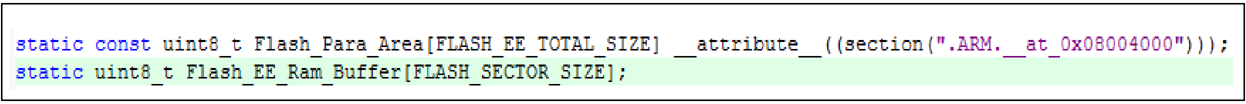
3. Based on the analysis of points 1 and 2, only sectors 1-3 are suitable for Flash to simulate the EEPROM.

4 Initialization Parameter Description for APM32F4xx FLASH Simulating EEPROM Routine

When simulating EEPROM through APM32F4 Flash, the sector size of the Flash, the total size of the Flash, the start address of the Flash, and the size of the test array need to be configured. Below are explanations and configuration description of initialization parameter macro definition or variable definition of this section.

4.1 Specify the Flash start address and RAM space

The code is as follows:



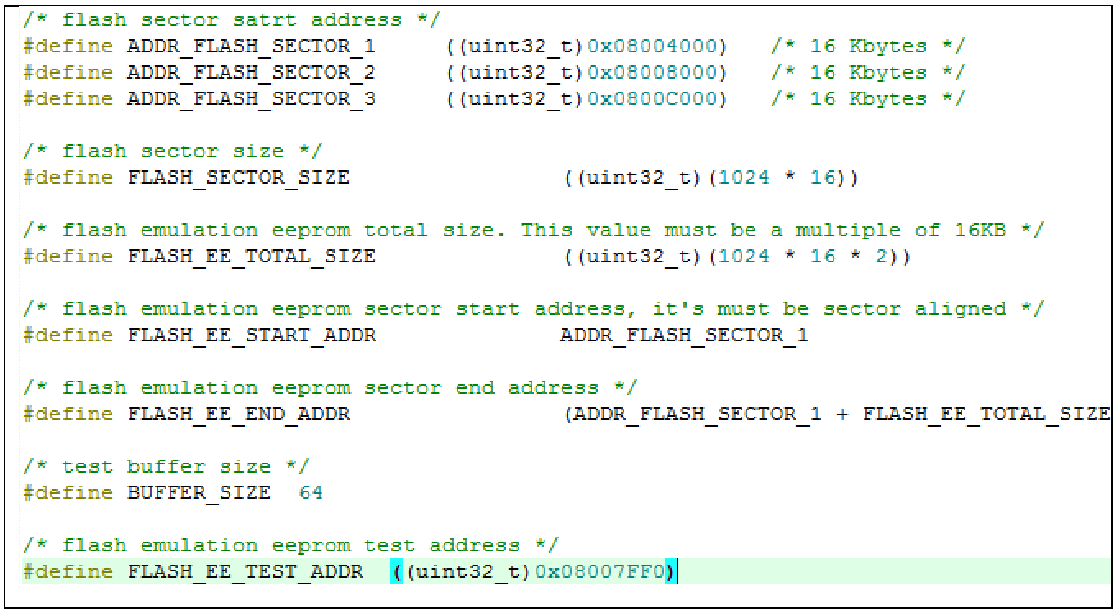
```
static const uint8_t Flash_Para_Area[FLASH_EE_TOTAL_SIZE] __attribute__((section(".ARM.__at_0x08004000")));
static uint8_t Flash_EE_Ram_Buffer[FLASH_SECTOR_SIZE];
```

1. In the first sentence of code, a const array is defined and the compiler properties are used to specify the start address of the array in Flash. So when we use the sectors 1-3, this Flash space will not be overwritten by the code. The size of this array is the size of the EEPROM simulated by Flash, and users can define the size of the space themselves, but it must be an integer multiple of the sector size.

2. Define a buffer with the same size as the Flash sector in RAM to solve the problem that the original data cannot be changed when erasing Flash.

4.2 Flash sector configuration initialization

The code is as follows:



```
/* flash sector satrt address */
#define ADDR_FLASH_SECTOR_1     ((uint32_t)0x08004000)   /* 16 Kbytes */
#define ADDR_FLASH_SECTOR_2     ((uint32_t)0x08008000)   /* 16 Kbytes */
#define ADDR_FLASH_SECTOR_3     ((uint32_t)0x0800C000)   /* 16 Kbytes */

/* flash sector size */
#define FLASH_SECTOR_SIZE                ((uint32_t)(1024 * 16))

/* flash emulation eeprom total size. This value must be a multiple of 16KB */
#define FLASH_EE_TOTAL_SIZE              ((uint32_t)(1024 * 16 * 2))

/* flash emulation eeprom sector start address, it's must be sector aligned */
#define FLASH_EE_START_ADDR              ADDR_FLASH_SECTOR_1

/* flash emulation eeprom sector end address */
#define FLASH_EE_END_ADDR                (ADDR_FLASH_SECTOR_1 + FLASH_EE_TOTAL_SIZE

/* test buffer size */
#define BUFFER_SIZE   64

/* flash emulation eeprom test address */
#define FLASH_EE_TEST_ADDR   ((uint32_t)0x08007FF0)
```

1. In the first three sentences of code, the start address of each sector is configured for experimental operation.

2. In the fourth sentence of the code, the size of the experimental sector is configured.

3. In the fifth sentence of the code, the total size of the experimental sector is configured, and is twice the size of the experimental sector. The reason is that in the Flash simulating EEPROM experiment, every time a data is written, an erase operation will be performed to leave space for new data. Moreover, Flash can only erase within the entire sector, so if the new data only covers partial data of a certain sector, it will result in the mixture of old and new data, and data loss or error will be caused.

4. In the sixth sentence of the code, the start address in the Flash simulating EEPROM experiment is configured and aligned with the sector.

5. In the seventh sentence of the code, the end address of the Flash simulating EEPROM is configured.

6. In the eighth sentence of the code, the test cache size is configured.

7. In the ninth sentence of the code, the boundary address of the first sector is selected as the test address to test whether cross-sector test can be conducted and observe the read and write stability in the entire Flash chip.

5 Introduction to routine design of APM32F4xx FLASH simulating EEPROM

Routine function introduction: After downloading the program, initialize the serial port, fill in the test array data by loop, and then call the Flash write operation interface function to perform write operation. After the write operation is over, call the Flash read operation interface function to perform read operation. Finally, by loop, judge whether write and read are consistent. If they are consistent, print "Test Successful!"; otherwise, print "Test Error!".

5.1 Hardware design

Hardware design required for the experiment of this routine:

1. Internal Flash of APM32F4
2. USB-to-TTL line

The wiring of the serial port used in this experiment is shown in PA9 and PA10 of Figure 5-1, and a USB-to-TTL line is used to receive and transmit data through the serial port.

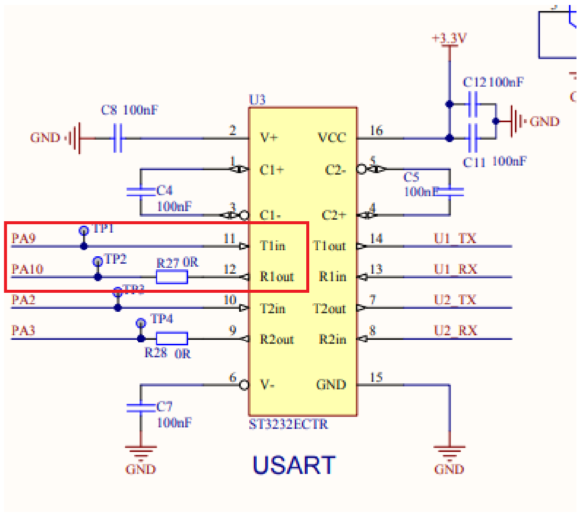


Figure 5-1 Serial Port Wiring Schematic Diagram

5.2 Software design

To implement the experiment of APM32F4xx Flash simulating EEPROM, the size of the sector shall be planned and defined, and the read and write operation functions of Flash shall be implemented. The main program code is mainly used to write the defined cache size into the test array, then perform Flash read and write operation, and finally compare and verify the written array and read array. The flowchart and code content of the main program code and read and write operation flow code will be introduced below:

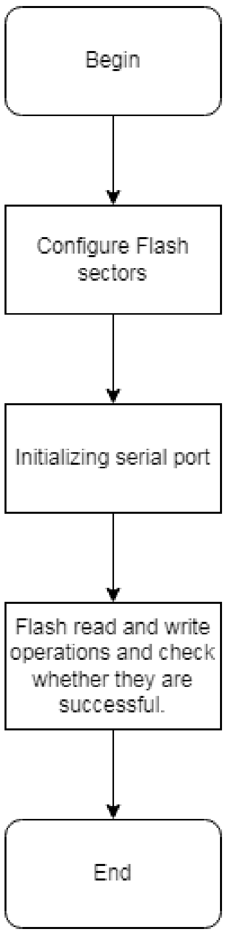
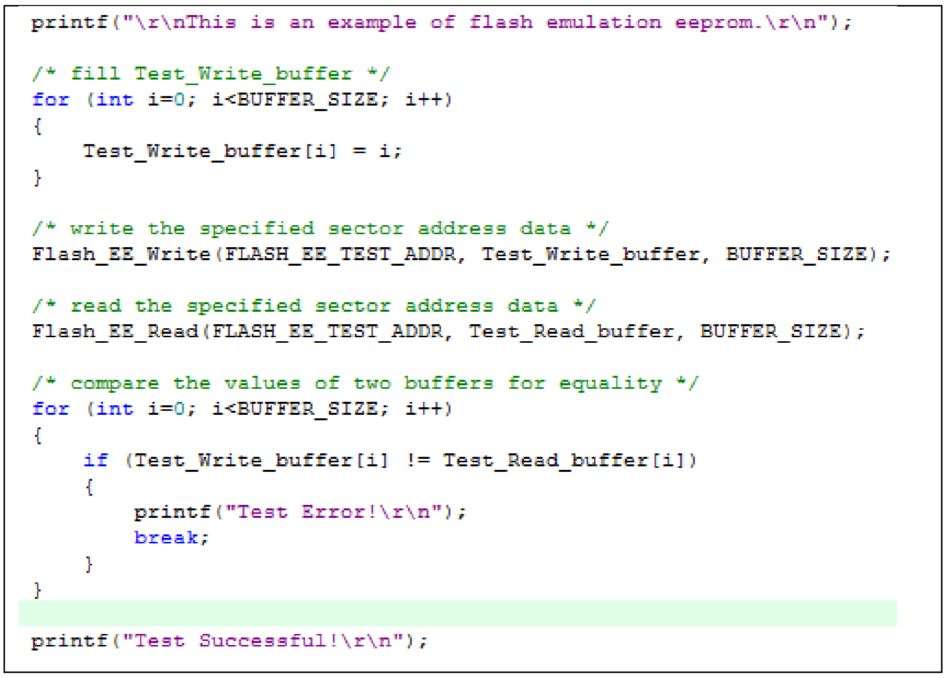

Begin
Configure Flash sectors
Initializing serial port
Flash read and write operations and check whether they are successful.
End

Figure 5-2 Flow Chart of Main Program Code for Flash Simulating EEPROM Experiment

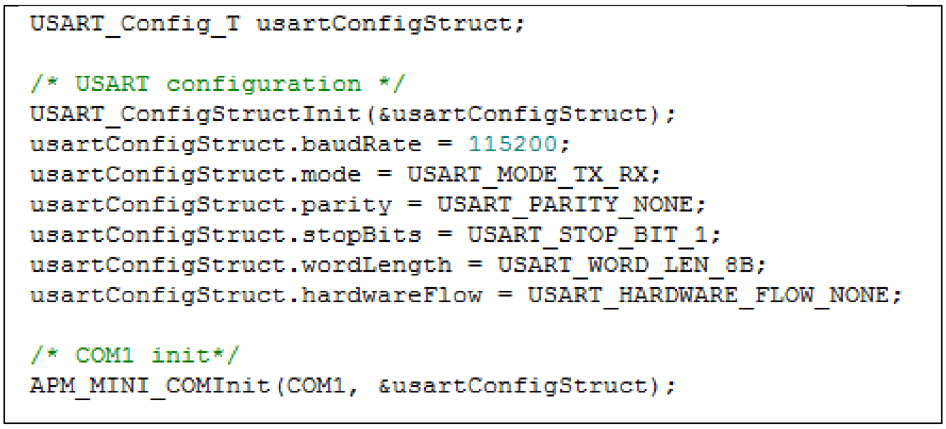


```
printf("\r\nThis is an example of flash emulation eeprom.\r\n");

/* fill Test_Write_buffer */
for (int i=0; i<BUFFER_SIZE; i++)
{
    Test_Write_buffer[i] = i;
}

/* write the specified sector address data */
Flash_EE_Write(FLASH_EE_TEST_ADDR, Test_Write_buffer, BUFFER_SIZE);

/* read the specified sector address data */
Flash_EE_Read(FLASH_EE_TEST_ADDR, Test_Read_buffer, BUFFER_SIZE);

/* compare the values of two buffers for equality */
for (int i=0; i<BUFFER_SIZE; i++)
{
    if (Test_Write_buffer[i] != Test_Read_buffer[i])
    {
        printf("Test Error!\r\n");
        break;
    }
}

printf("Test Successful!\r\n");
```

Figure 5-3 Main Program Code Diagram of Flash Simulating EEPROM Experiment

5.2.1 USART GPIO pin configuration

Conduct basic structure configuration for the GPIO pins used by USART, and configure the baud rate and output mode. The reference code is as follows:



```
USART_Config_T usartConfigStruct;

/* USART configuration */
USART_ConfigStructInit(&usartConfigStruct);
usartConfigStruct.baudRate = 115200;
usartConfigStruct.mode = USART_MODE_TX_RX;
usartConfigStruct.parity = USART_PARITY_NONE;
usartConfigStruct.stopBits = USART_STOP_BIT_1;
usartConfigStruct.wordLength = USART_WORD_LEN_8B;
usartConfigStruct.hardwareFlow = USART_HARDWARE_FLOW_NONE;

/* COM1 init*/
APM_MINI_COMInit(COM1, &usartConfigStruct);
```

5.2.2 Flash write operation configuration

In the APM32F4xx Flash simulating EEPROM experiment, the implementation of the write operation is crucial. The code block diagram and code content of the write operation implementation are introduced below. First make initialization definition of variables in Flash_EE_Write function. The specific code is as follows:

```
uint32_t NumOfSector = 0, NumOfByte = 0, OfsetAddr = 0;

uint32_t count = 0, temp = 0;

/* offerset address in the sector */
OfsetAddr = WriteAddr % FLASH_SECTOR_SIZE;

/* The size of the remaining space inthe sector from WriteAddr */
count = FLASH_SECTOR_SIZE - OfsetAddr;

/* Calculate how many sectors to write */
NumOfSector =  len / FLASH_SECTOR_SIZE;

/* Calculate how many bytes are left less than one sector */
NumOfByte = len % FLASH_SECTOR_SIZE;
```

In the initialization definition of Flash_EE_Write function:

OfsetAddr: Calculate the offset address of the incoming parameter in the sector. This variable is used to judge whether the incoming offset address is aligned with the start address of the sector.

Count: It is used to count how much remaining space in the operating sector can be used for writing.

NumOfSector: Calculate how many sectors are used for write operation.

NumOfByte: Calculate the number of remaining bytes in the sector that is less than one sector. When there are bytes written in the sector, this variable is used to calculate the number of remaining bytes in the sector and determine the number of places where write operation can be performed.

After the definition initialization is introduced, the specific operation in Flash_EE_Write function is introduced. The specific code is as follows:

Flash_EE_Write:

The if statement judges whether the offset address is aligned with the start address of the sector.

```
/* OfsetAddr = 0, WriteAddr is sector aligned */
if (OfsetAddr == 0)
{
    /* len < FLASH_SECTOR_SIZE */
    if (NumOfSector == 0)
    {
        Flash_EE_WriteOneSector(WriteAddr, pData, len);
    }
    /* len > FLASH_SECTOR_SIZE */
    else
    {
        /* write NumOfSector sector */
        while (NumOfSector--)
        {
            Flash_EE_WriteOneSector(WriteAddr, pData, FLASH_SECTOR_SIZE);
            WriteAddr +=  FLASH_SECTOR_SIZE;
            pData += FLASH_SECTOR_SIZE;
        }

        /* write remaining data */
        Flash_EE_WriteOneSector(WriteAddr, pData, NumOfByte);
    }
}
```

The else statement is interpreted as misaligned.

```
    else
    {
        /* len < FLASH_SECTOR_SIZE, the data length is less than one sector */
        if (NumOfSector == 0)
        {
            /* NumOfByte > count,  need to write across the sector */
            if (NumOfByte > count)
            {
                temp = NumOfByte - count;
                /* fill the current sector */
                Flash_EE_WriteOneSector(WriteAddr, pData, count);

                WriteAddr +=  count;
                pData += count;
                /* write remaining data */
                Flash_EE_WriteOneSector(WriteAddr, pData, temp);
            }
            else
            {
                Flash_EE_WriteOneSector(WriteAddr, pData, len);
            }
        }
        /* len > FLASH_SECTOR_SIZE */
        else
        {
            len -= count;
            NumOfSector =  len / FLASH_SECTOR_SIZE;
            NumOfByte = len % FLASH_SECTOR_SIZE;

            /* write count data */
            Flash_EE_WriteOneSector(WriteAddr, pData, count);

            WriteAddr +=  count;
            pData += count;
            /* write NumOfSector sector */
            while (NumOfSector--)
            {
                Flash_EE_WriteOneSector(WriteAddr, pData, FLASH_SECTOR_SIZE);
                WriteAddr +=  FLASH_SECTOR_SIZE;
                pData += FLASH_SECTOR_SIZE;
            }

            if (NumOfByte != 0)
            {
                Flash_EE_WriteOneSector(WriteAddr, pData, NumOfByte);
            }
        }
    }
```

The specific operation code includes the following steps:

1. When the incoming address is aligned with the start address of the sector, first determine whether the written space is consistent with the size of the sector, and then determine whether the remaining bytes of the sector and the size of the remaining space can be written at once. After the judgment is made, the variable initialization definition is calculated and written.

2. When the incoming address is not aligned with the start address of the sector, first determine whether the written space is consistent with the space size of the sector, and then determine whether the remaining bytes of the sector and the size of the remaining space can be written at once. After the judgment is made, the variable initialization definition is calculated and written.

Next, the initialization definition of variables in Flash_EE_WriteOneSector will be introduced specifically as follows:

```
startAddr = WriteAddr / FLASH_SECTOR_SIZE * FLASH_SECTOR_SIZE;
offesetAddr = WriteAddr % FLASH_SECTOR_SIZE;
```

In the initialization definition of Flash_EE_Write function:

StartAddr: It is used to count the start address when writing the sector data. This variable is used to read or write data to RAM before the erase operation.

OffsetAddr: It is used to write data to this offset address.

Flash_EE_WriteOneSector:

```
/* unlock flash for erase or write*/
FMC_Unlock();

/* check whether the sector need to be erased */
for (i=0; i<len; i++)
{
    if (Flash_EE_ReadByte(WriteAddr + i) != 0xFF)
    {
        isErase = 1;
        break;
    }
}
/* the sector needs to be erase */
if (isErase == 1)
{
    /* read the entire sector data to the buffer before write or erase */
    Flash_EE_Read(startAddr, Flash_EE_Ram_Buffer, FLASH_SECTOR_SIZE);

    /* copy the data to the buffer */
    for (i=0; i<len; i++)
    {
        Flash_EE_Ram_Buffer[offesetAddr + i] = pData[i];
    }

    /* erase the sector where the address is located */
    FMC_EraseSector(Get_Flash_Sector_Num(WriteAddr), FMC_VOLTAGE_3);

    /* write the entire sector data */
    for (i=0; i<FLASH_SECTOR_SIZE; i++)
    {
        FMC_ProgramByte(startAddr, Flash_EE_Ram_Buffer[i]);
        startAddr += 1;
    }
}
```

```
/* the sector don't need to be erase */
else
{
    /* write n bytes of data to the sector */
    for (i=0; i<len; i++)
    {
        FMC_ProgramByte(WriteAddr, pData[i]);
        WriteAddr += 1;
    }
}

/* lock flash */
FMC_Lock();
```

The specific operation code includes the following steps:

1. Perform unlock operation
2. Check if the sector needs to erase.
3. If erase operation is required, first read the data and copy it to RAM, perform the erase operation, and then write the data of the entire sector to RAM.

4. If erase operation is not required, the data can be directly written to the incoming address.

5.2.3 Flash read operation configuration

In the experiment of APM32F4xx Flash simulating EEPROM, an eight-bit data from the Flash only needs to be read for the read operation. The operation step is to use the dereference to perform operation. The code content of the function Flash_EE_Read and Flash_EE_ReadByte involving read operation is described below:

Flash_EE_Read：

```
/* read data */
for (int i = 0; i < len; i++)
{
    pData[i] = Flash_EE_ReadByte(ReadAddr);
    ReadAddr += 1;
}
```

The specific operation code includes the following steps:

1. Read the values in the incoming addresses successively in the loop.

Flash_EE_ReadByte：

```
static uint8_t Flash_EE_ReadByte(uint32_t Addr)
{
    return (*(__IO uint8_t*)Addr);
}
```

The specific operation code includes the following steps:

1. Read an eight-digit data by dereference.

6 Revision history

Table 1 Document Version History

Date	Version	Revision History
June 16, 2023	1.0	First draft

Statement

This manual is formulated and published by Zhuhai Geehy Semiconductor Co., Ltd. (hereinafter referred to as "Geehy"). The contents in this manual are protected by laws and regulations of trademark, copyright and software copyright. Geehy reserves the right to correct and modify this manual at any time. Please read this manual carefully before using the product. Once you use the product, it means that you (hereinafter referred to as the "users") have known and accepted all the contents of this manual. Users shall use the product in accordance with relevant laws and regulations and the requirements of this manual.

1. Ownership of rights

This manual can only be used in combination with chip products and software products of corresponding models provided by Geehy. Without the prior permission of Geehy, no unit or individual may copy, transcribe, modify, edit or disseminate all or part of the contents of this manual for any reason or in any form.

The "Geehy" or "Geehy" words or graphics with "®" or "TM" in this manual are trademarks of Geehy. Other product or service names displayed on Geehy products are the property of their respective owners.

2. No intellectual property license

Geehy owns all rights, ownership and intellectual property rights involved in this manual. Geehy shall not be deemed to grant the license or right of any intellectual property to users explicitly or implicitly due to the sale and distribution of Geehy products and this manual.

If any third party’s products, services or intellectual property are involved in this manual, it shall not be deemed that Geehy authorizes users to use the aforesaid third party’s products, services or intellectual property, unless otherwise agreed in sales order or sales contract of Geehy.

3. Version update

Users can obtain the latest manual of the corresponding products when ordering Geehy products.

If the contents in this manual are inconsistent with Geehy products, the agreement in Geehy sales order or sales contract shall prevail.

4. Information reliability

The relevant data in this manual are obtained from batch test by Geehy Laboratory or cooperative third-party testing organization. However, clerical errors in correction or errors caused by differences in testing environment may occur inevitably. Therefore, users should understand that Geehy does not bear any responsibility for such errors that may occur in this manual. The relevant data in this manual are only used to guide users as performance parameter reference and do not constitute Geehy's guarantee for any product performance.

Users shall select appropriate Geehy products according to their own needs, and effectively verify and test the applicability of Geehy products to confirm that Geehy products meet their own needs, corresponding standards, safety or other reliability requirements. If loses are caused to users due to the user's failure to fully verify and test Geehy products, Geehy will not bear any responsibility.

5. Compliance requirements

Users shall abide by all applicable local laws and regulations when using this manual and the matching Geehy products. Users shall understand that the products may be restricted by the export, re-export or other laws of the countries of the product suppliers, Geehy, Geehy distributors and users. Users (on behalf of itself, subsidiaries and affiliated enterprises) shall agree and promise to abide by all applicable laws and regulations on the export and re-export of Geehy products and/or technologies and direct products.

6. Disclaimer

This manual is provided by Geehy "as is". To the extent permitted by applicable laws, Geehy does not provide any form of express or implied warranty, including without limitation the warranty of product merchantability and applicability of specific purposes. Geehy will bear no responsibility for any disputes arising from the subsequent design and use of Geehy products by users.

7. Limitation of liability

In any case, unless required by applicable laws or agreed in writing, Geehy and/or any third party providing this manual "as is" shall not be liable for damages, including any general damages, special direct, indirect or collateral damages arising from the use or no use of the information in this manual (including without limitation data loss or inaccuracy, or losses suffered by users or third parties).

8. Scope of application

The information in this manual replaces the information provided in all previous versions of the manual.