使用Data Flash模拟EEPROM

Application Note for 32-bit NuMicro® Family

Document Information

Abstract	这份文件介绍如何使用Data Flash模拟EEPROM，并且利用SRAM加速资料读写的速度。内容包含原理介绍、性能数据以及使用建议。附录提供范例程序源码以及函数库。
Apply to	NuMicro® Cortex®-M0/M4全系列，本文以NuMicro® M051 DE系列为例。

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

目录

1 简介

EEPROM具有可byte write/byte read以及高达百万次可靠的擦写次数，通常被使用者用来存放程序中会时常变更的非挥发性资料。对于单晶片产品，通常不具有内建的EEPROM能够提供给使用者，而是基于Flash来存放使用者的资料。但是Flash的擦写次数无法与EEPROM比拟。

现在我们提出一个机制，能够组合两个page以上的Data Flash来模拟EEPROM使用，使用SRAM加速读写资料的速度、能够达到百万次可靠的擦写次数、记录擦写循环次数，并且可以将资料量分成数个较小的资料群以减少Data Flash page数量。

- 使用 SRAM 加速读写资料速度

当进行资料写入的时候，会同步写入Data Flash和SRAM；一旦当前使用的Data Flash page已经写满，就会使用下一个Data Flash page，并且能够直接将SRAM存放的资料存入，节省一般将资料移入新的Data Flash page，需要搜寻全部Data Flash中已写入的资料的时间。

当需要读取资料的时候，可以直接由SRAM中读出资料，无须从Data Flash中寻找需要的资料，能够节省搜寻过程的时间，Data Flash中的资料只用来初始化SRAM。

- 达到百万次可靠的擦写次数

这样使用Data Flash模拟EEPROM的方法，能够使用byte write/byte read以及超过百万次可靠的擦写次数。

- 记录擦写循环次数

使用者可以由写入Counter的值了解Data Flash page的擦写循环次数。

- 将资料量分成数个较小的资料群以减少 Data Flash page 数量

对于NuMicro® Cortex®-M0系列，如图 1-1所示，如果使用者需要存放的资料量增加，为了满足要求的可靠擦写次数，需要的Data Flash page数量会大量增加。因此，我们建议使用者可以将要存放的资料量分成数个较小的资料群，利用较少的Data Flash page数量就可以达到需要的可靠擦写次数。

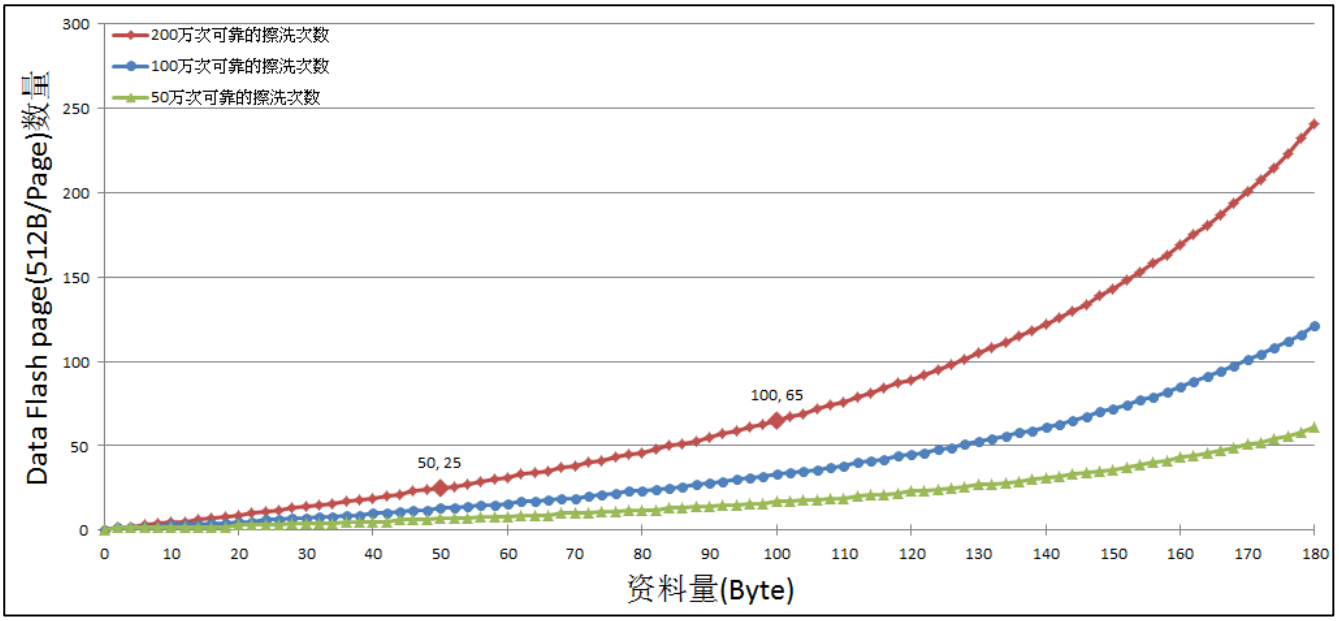


图 1-1 资料量与需要的 Data Flash page 数量 – NuMicro® Cortex®-M0 系列 (512B/page)

资料量(Byte)	Data Flash page(512B/page)数量	资料量(Byte)	Data Flash page(512B/page)数量
32	15	50	25
34	16	52	26
36	17	54	27
38	18	56	29
40	19	58	30
42	20	60	31
44	21	62	33
46	23	64	34
48	24		

表 1-1 常用资料量与需要的 Data Flash page 数量 – NuMicro® Cortex®-M0 系列 (512B/page)

对于NuMicro® Cortex®-M4系列，如图 1-2所示，如果使用者需要存放的资料量增加，为了满足要求的可靠擦写次数，需要增加的Data Flash page数量并不会大量增加。因此，使用者可以直接操作，不需将资料量分成数个较小的资料群。

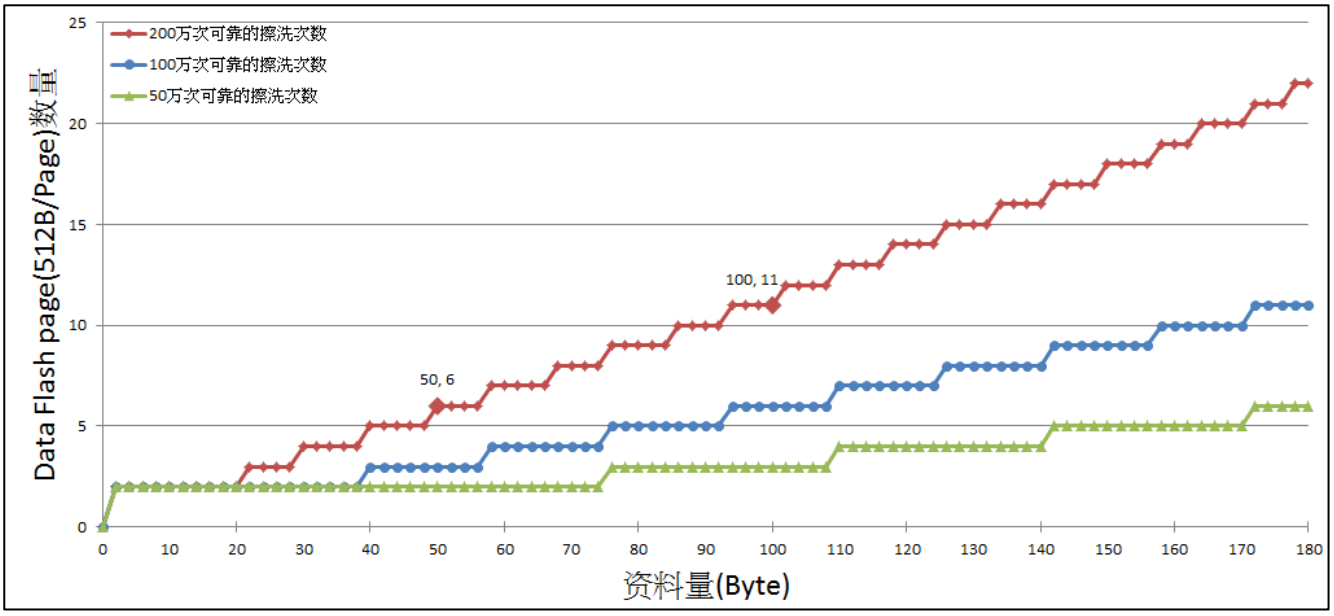


图 1-2 资料量与需要的Data Flash page数量 - NuMicro® Cortex®-M4系列 (2KB/page)

资料量(Byte)	Data Flash page(2KB/page)数量	资料量(Byte)	Data Flash page(2KB/page)数量
32	4	50	6
34	4	52	6
36	4	54	6
38	4	56	6
40	5	58	7
42	5	60	7
44	5	62	7
46	5	64	7
48	5		

表 1-2 常用资料量与需要的 Data Flash page 数量 - NuMicro® Cortex®-M4 系列 (2KB/page)

2 模拟EEPROM

本节将介绍如何使用Data Flash模拟EEPROM，并且利用SRAM加速资料读写的速度。

2.1 机制原理

在使用Data Flash模拟EEPROM时，使用者必须使用至少2个page以上的Data Flash，并且将每一page的Data Flash，以每两个字节为单位划分成若干个区块，如图 2-1所示，以NuMicro® Cortex®-M0系列为例，每一个page的Data Flash大小为512字节。第一个区块记录目前擦写循环次数的Counter值，其余存放资料的地址和值。同时SRAM也划分一个区块，用来记录存放的资料。

当使用者要存放资料的时候，会将资料的地址和值依序写到第一个Data Flash page，并且将值写入对应地址的SRAM。如果第一个Data Flash page的空间被写满时，会将目前已写入的有效资料(非0xFF) 从SRAM整理到第二个Data Flash page，并清除第一个Data Flash page。

使用SRAM来存放资料，可以减少从Data Flash读取资料的时间；当Data Flash page写满要将资料整理到下一个Data Flash page的时候，直接从SRAM读出资料，无须从Data Flash中寻找需要的资料，能够节省搜寻Data Flash中有效资料的时间。

Counter值能夠帮助使用者了解，目前已经使用过的可擦写次数。使用者可以有效掌握剩余的可靠擦写次数，推算产品的可用年限。

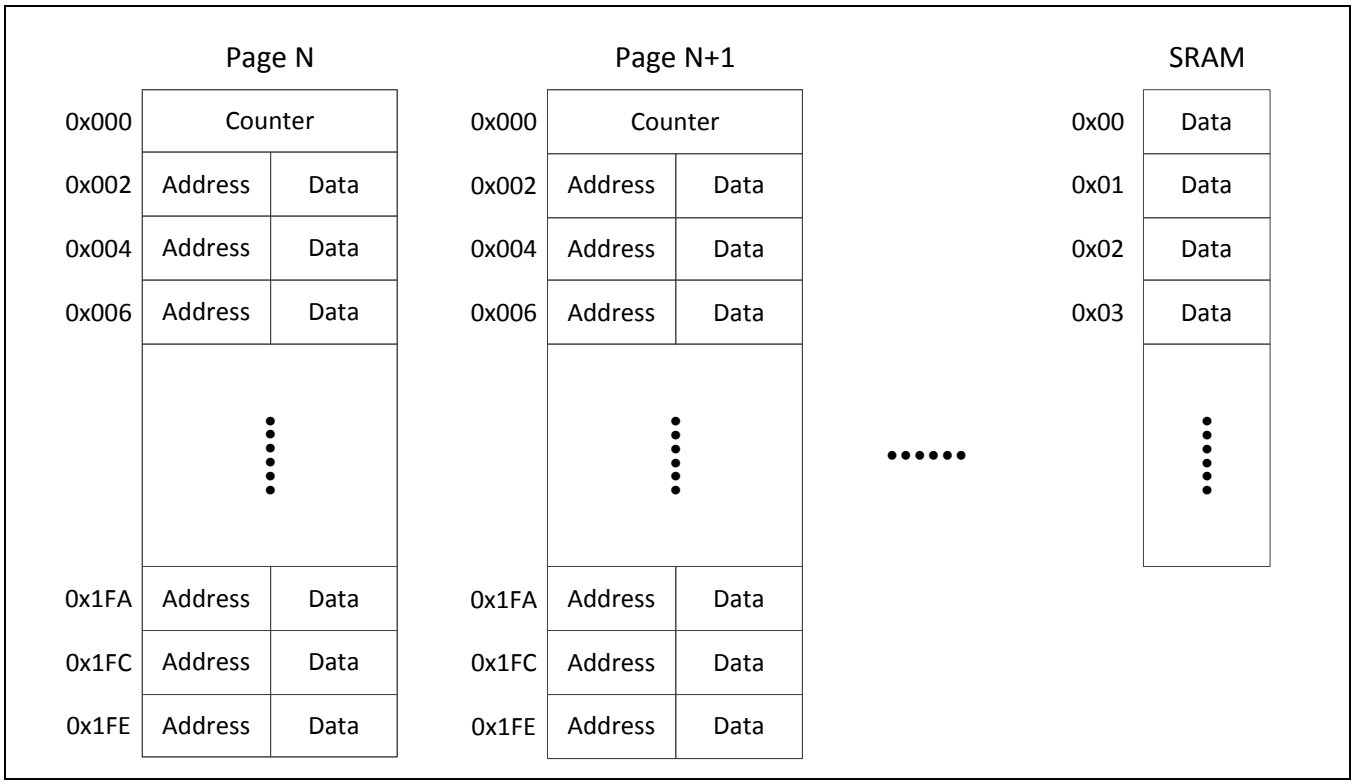


图 2-1 Data Flash page 和 SRAM

2.2 初始化 Data Flash

在使用者开始执行主程序前，首先需要初始化要调用的Data Flash page。由于只有保存资料的Data Flash page具有有效的Counter值(非0xFFFF)，我们可以通过寻找有效的Counter值来找出保存资料的Data Flash page。接着除了指向可以写入资料的地址，还需要将已保存的资料写到SRAM。详细流程请参考图 2-2，并请考函数Init_EEPROM()和Search_Valid_Page()。

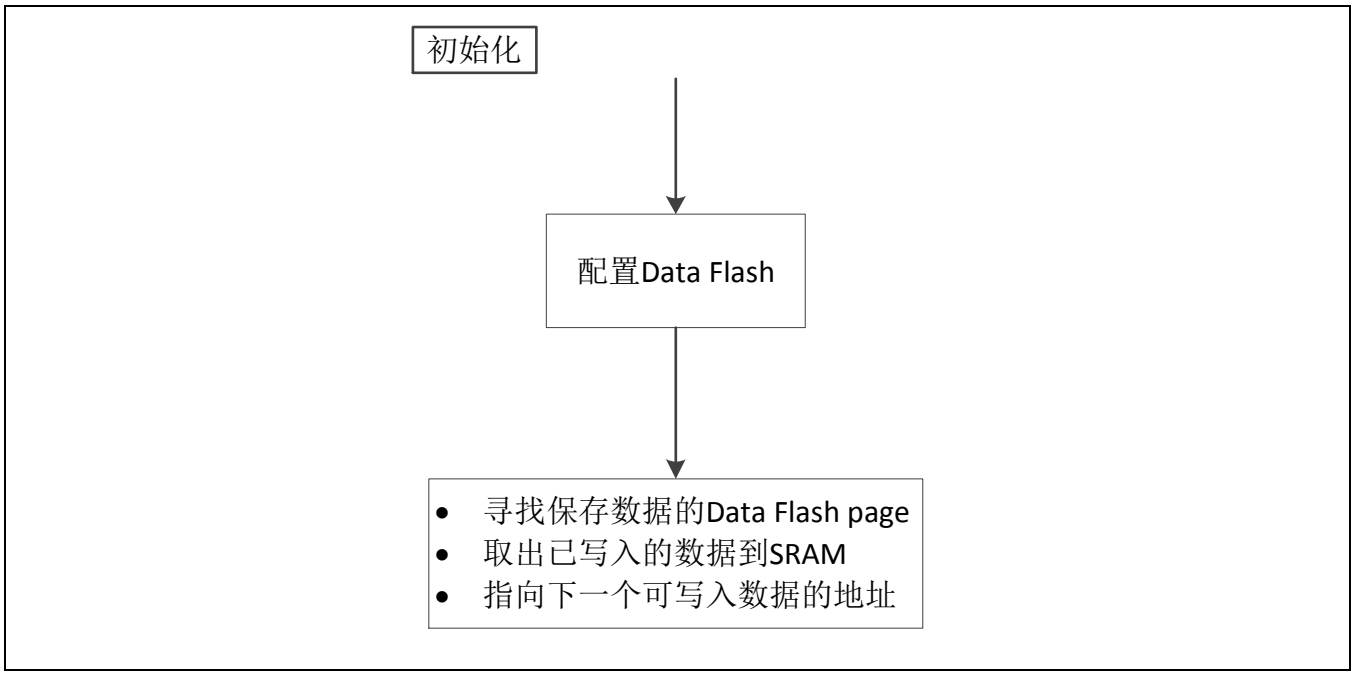


图 2-2 Data Flash初始化流程

2.3 写入资料

当使用者有资料写入的时候，首先比对是否已与SRAM中的资料相同。如果资料相同，则不需要执行写入，略过写入资料步骤；如果资料不同，则需要更新资料。

资料写入时，会同时写入到Data Flash和SRAM，接着判断目前使用的Data Flash page是否已经写满。如果还有未写入的空间，就指向下一个地址；如果Data Flash page已经写满，则将SRAM中有效的资料(非0xFF)写入下一个调用的Data Flash page储存。

相对于一般需要搜寻整个Data Flash page的方式，直接从SRAM中写入有效资料，可以大幅缩短，当目前Data Flash page已经写满，要调用下一个Data Flash page的时间。

如果目前使用的Data Flash page已经是可调用的最后一个Data Flash page，就将Counter增加计数1，再将SRAM中有效的资料(非0xFF)写入第一个可调用的Data Flash page。

使用者可以读取Counter的值，了解目前已经使用过的可擦写次数。

完成资料写入后，清除已写满的Data Flash page。当有新的资料需要写入的时候，将写入新的Data Flash page。

详细流程请参考图 2-3，并请参考函数Write_Data()和Manage_Next_Page()。

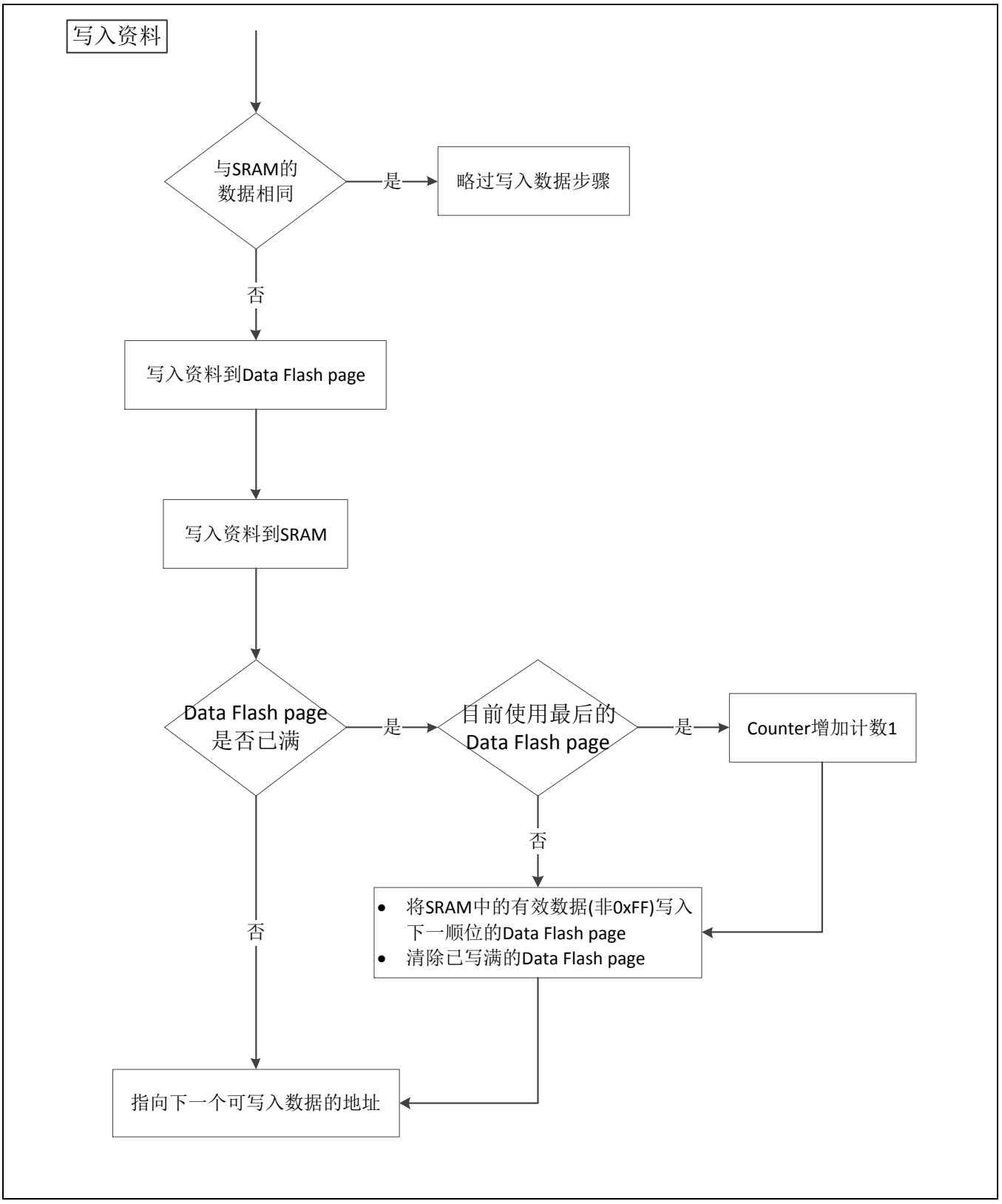


图 2-3 写入资料流程

2.4 读取资料

读取资料的时候，只需要将SRAM中的值读出即可，不需要从Data Flash page中寻找最后更新的资料，可以大幅节省读取资料需要的时间。详细流程请参考图 2-4，并请参考函数Read_Data()。

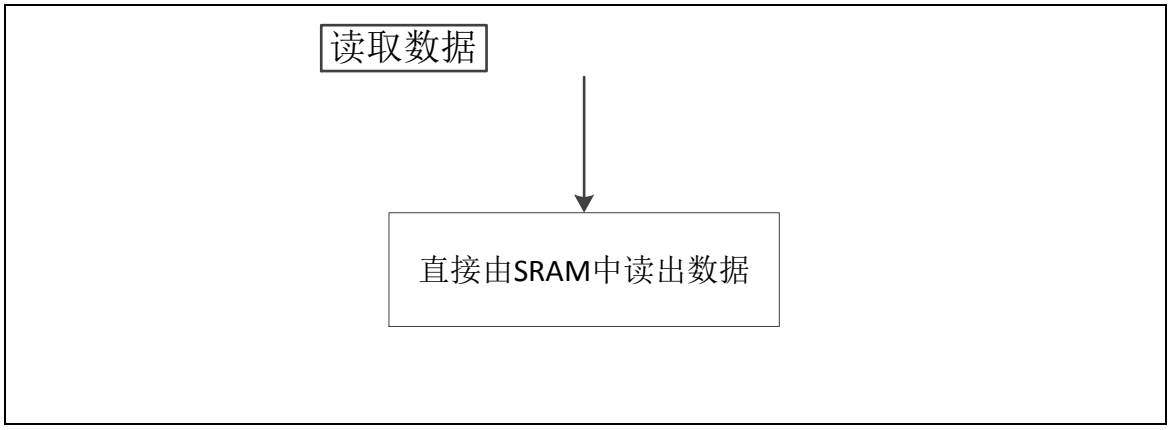


图 2-4 读取资料流程

2.5 读取 Counter 值

使用者可以读取Counter的值了解目前Data Flash page已经使用多少次可擦写循环，并推算目前剩余的可靠擦写次数。请参考函数Get_Cycle_Counter()。

2.6 资料读写范例

在主程序运行之前，首先需要初始化Data Flash。如图 2-5所示，我们选择组合两个page的Data Flash来模拟EEPROM使用。初始化过程会先找出目前储存有效资料的Data Flash page，Page 1，并将有效资料(非0xFF)放入SRAM。此范例中储存的资料量为8个字节，在SRAM中以队列方式存放这些资料。最后指向下一个可以写入资料的地址。

当使用者要写入0x07[0x68] (地址[资料]) 的时候，如图 2-6所示。首先会判断与SRAM中的资料是否相同。由于目前SRAM中的值与0x68不相等，因此会将资料的地址和值写入Data Flash的区块，并且将值写入对应地址0x07的SRAM阵列中。因为目前的Data Flash page并未写满，仅需要指向下一个可以写入资料的地址即可。

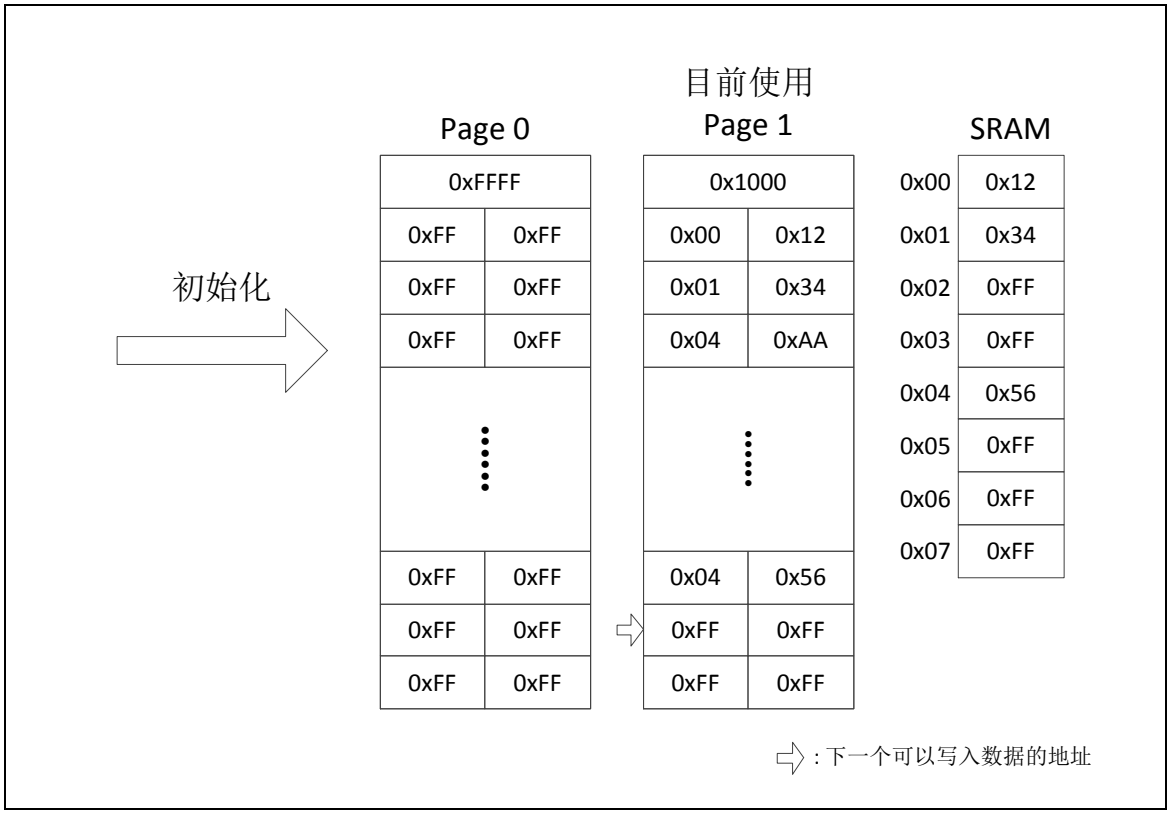


图 2-5 资料读写流程范例 - 初始化

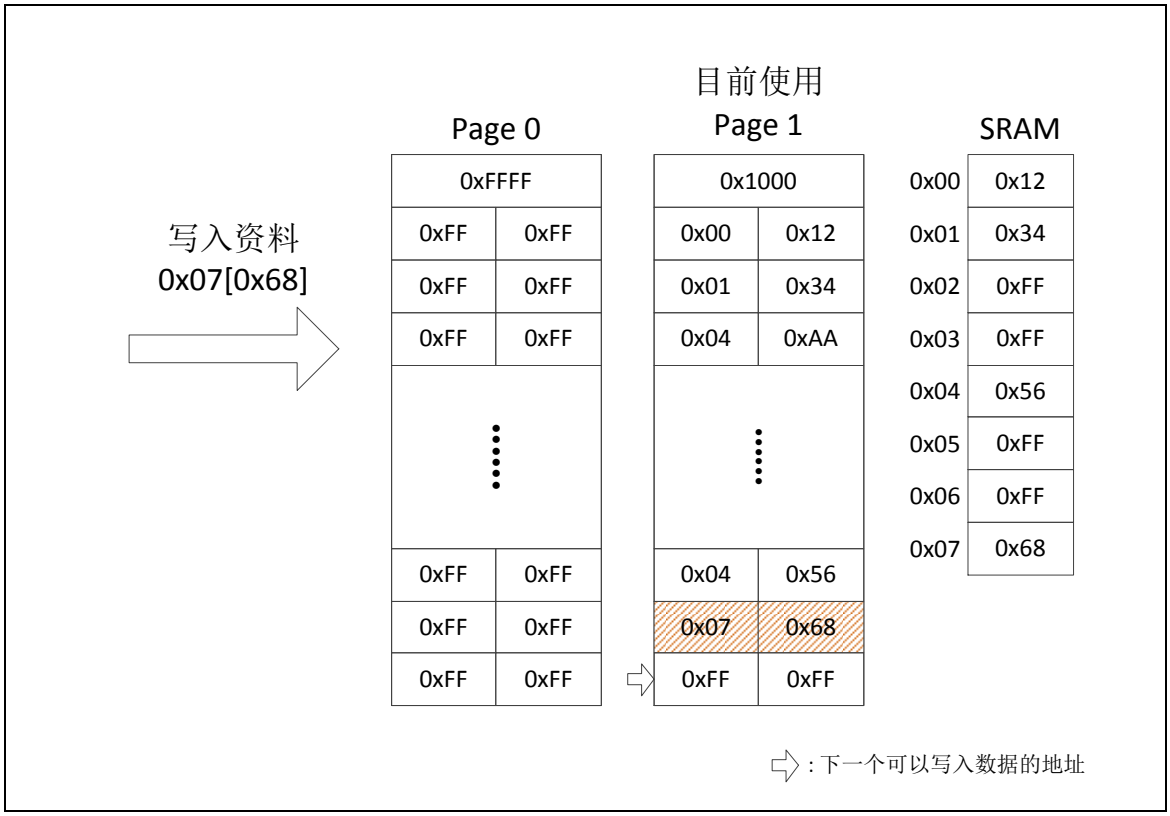


图 2-6 资料读写流程范例 - 写入新资料

如果使用者要写入资料0x04[0x56]，如图 2-7所示。由于目前SRAM中的值与0x56相等，因此略过写入资料的步骤。

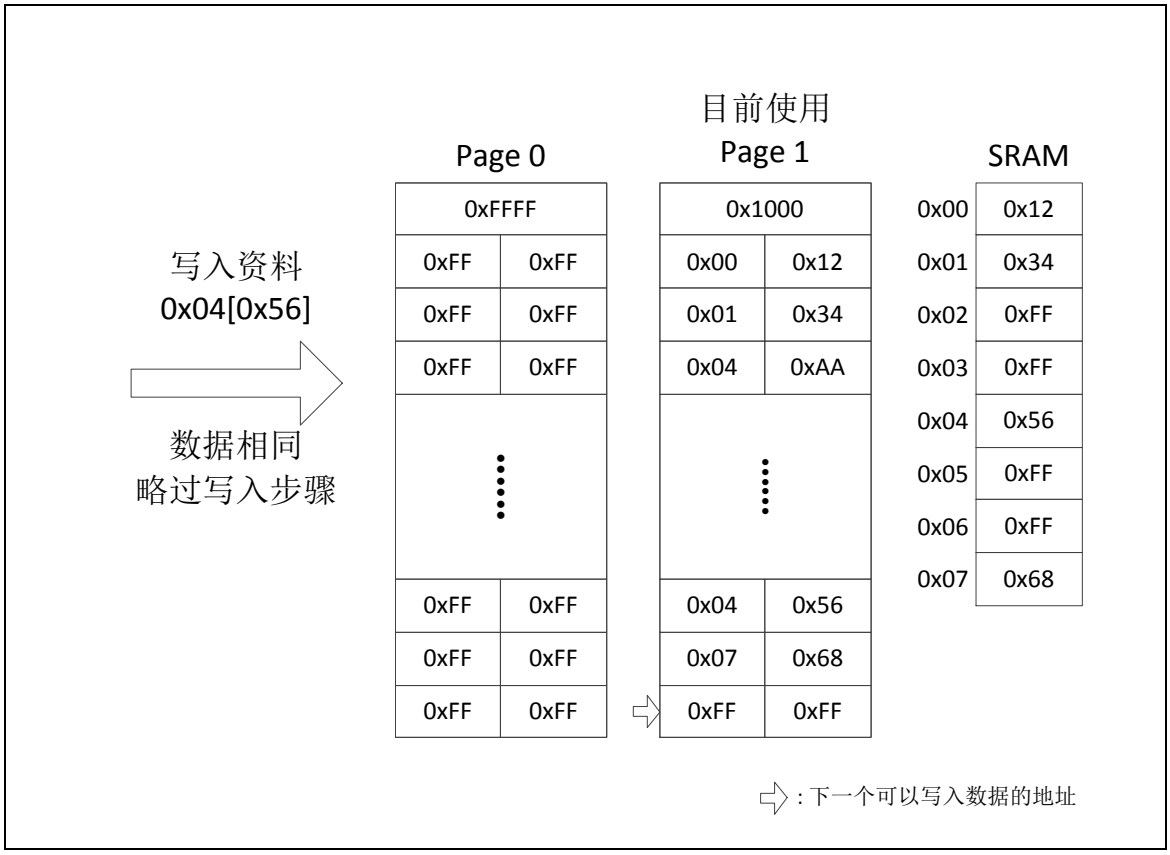


图 2-7 资料读写流程 - 写入相同资料

接着，当使用者要写入资料0x00[0xFF]的时候，如图 2-8所示。首先会判断与SRAM中的资料是否相同。由于目前SRAM中的值与0xFF不相等，因此会将资料的地址和值写入Data Flash的区块，并且将值写入对应地址0x00的SRAM队列中。

这时目前的Data Flash page已经写满，需要将SRAM的资料写到新的Data Flash page。而且目前使用的Data Flash page已经是可调用的最后一个Data Flash page，因此将Counter增加计数1。再将SRAM中有效的资料(非0xFF)写入第一个调用的Data Flash page，所以仅将地址0x01、0x04、0x06、0x07的值写入到新调用的Data Flash page。

完成资料写入后，清除已写满的Data Flash page，最后指向下一个可以写入资料的地址。

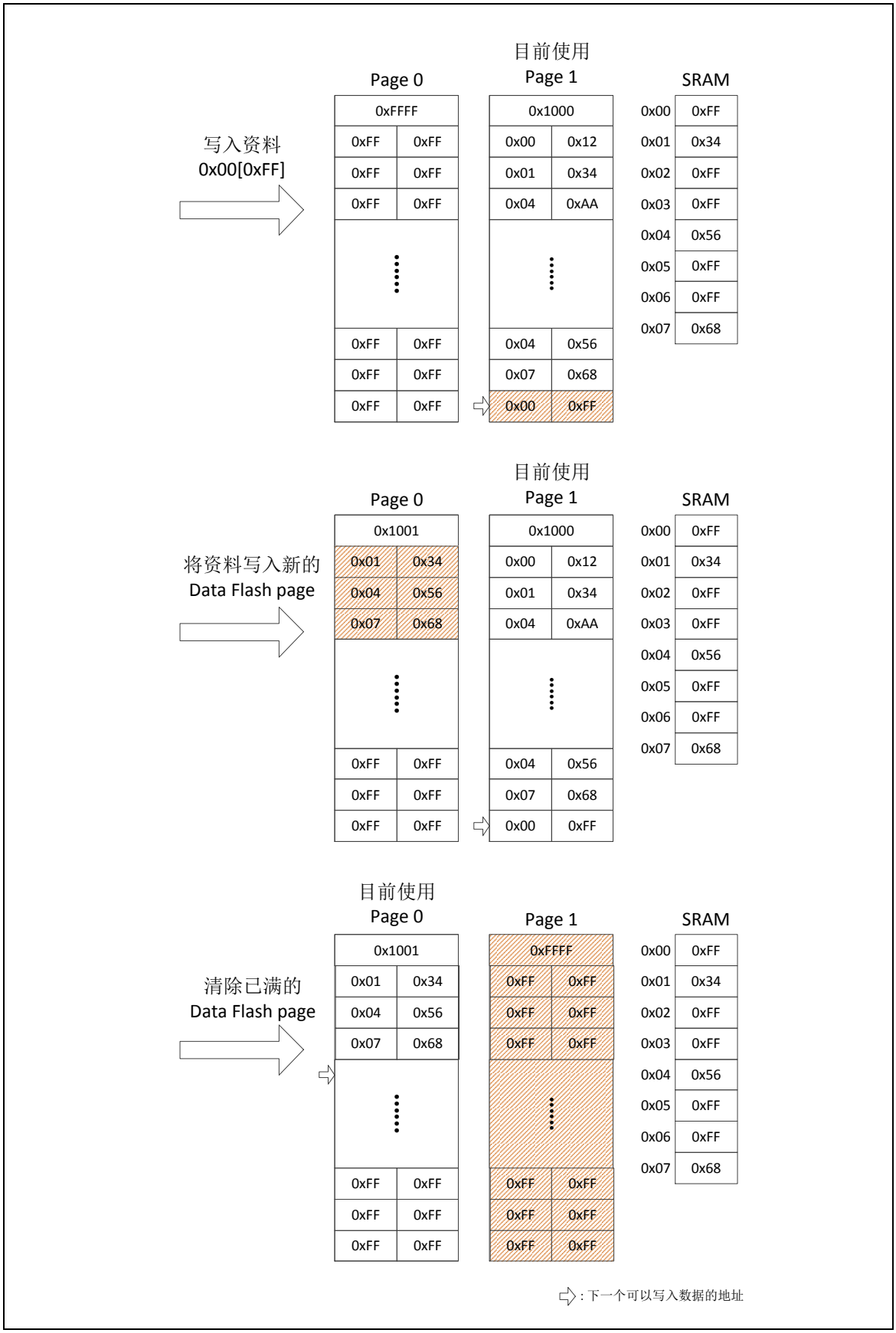
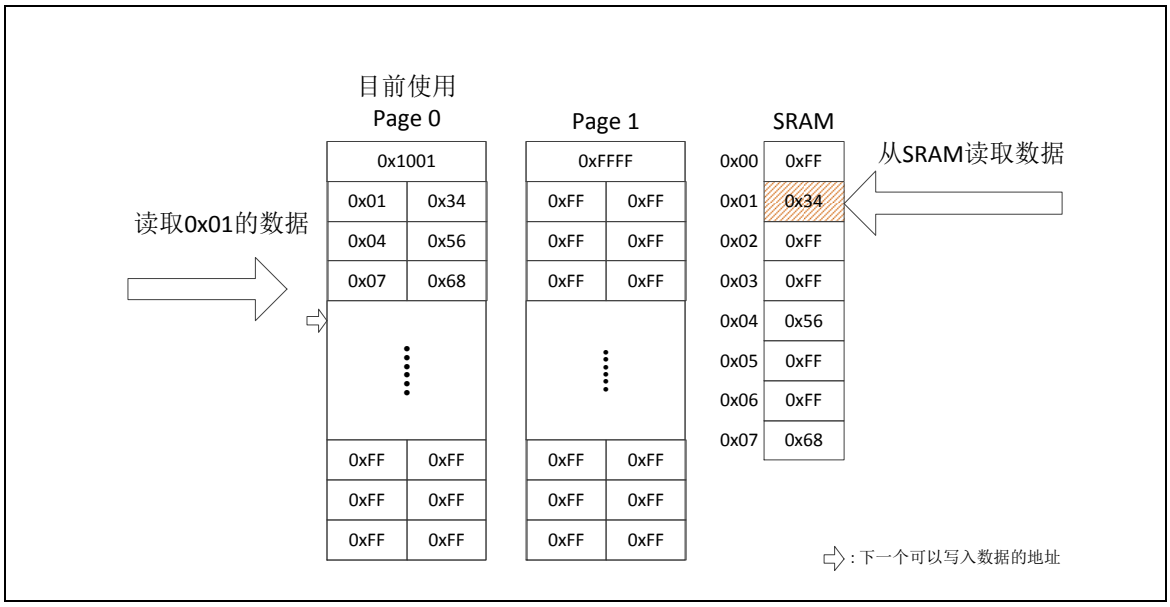

目前使用
Page 0
Page 1
SRAM
写入资料
0x00[0xFF]
0xFFFF
0x1000
0x00 0x12
0x01 0x34
0x04 0xAA
0x04 0x56
0x07 0x68
0x00 0xFF
0x00 0xFF
0x01 0x34
0x02 0xFF
0x03 0xFF
0x04 0x56
0x05 0xFF
0x06 0xFF
0x07 0x68
目前使用
Page 0
Page 1
SRAM
将资料写入新的
Data Flash page
0x1001
0x01 0x34
0x04 0x56
0x07 0x68
0x1000
0x00 0x12
0x01 0x34
0x04 0xAA
0x04 0x56
0x07 0x68
0x00 0xFF
0x00 0xFF
0x01 0x34
0x02 0xFF
0x03 0xFF
0x04 0x56
0x05 0xFF
0x06 0xFF
0x07 0x68
目前使用
Page 0
Page 1
SRAM
清除已满的
Data Flash page
0x1001
0x01 0x34
0x04 0x56
0x07 0x68
0xFFFF
0xFF 0xFF
0xFF 0xFF
0xFF 0xFF
0xFF 0xFF
0xFF 0xFF
0xFF 0xFF
0x00 0xFF
0x01 0x34
0x02 0xFF
0x03 0xFF
0x04 0x56
0x05 0xFF
0x06 0xFF
0x07 0x68
: 下一个可以写入数据的地址

图 2-8 资料读写流程范例 - 写入新资料并调用下一个 Data Flash page

当使用者读取资料的时候，只需要将SRAM中的值读出即可。



2.7 可靠度计算

在计算可靠度的时候，我们以每笔资料的平均可靠擦写次数做为评估基准。为了满足EEPROM的使用性，每笔资料的平均可靠擦写次数都需要在一百万次以上。而每笔资料的平均可靠擦写次数公式如下：

$$\text{每笔资料的平均可靠擦写次数} = \frac{(S - C - D)}{D} \times P \times E$$

S: Data Flash page大小，在NuMicro® Cortex®-M0系列中为512字节，在NuMicro® Cortex®-M4系列中为2048字节。

C: Counter使用大小， 固定为2字节。

D: 每笔资料使用大小 × 资料量，其中每笔资料使用大小固定为2字节。

P: 使用的Data Flash page数量，由使用者选择调用多少Data Flash page模拟EEPROM。

E: Data Flash page可靠的可擦写次数。

以NuMicro® M051 DE系列为例，如果使用者调用4个Data Flash page，共2K字节模拟成EEPROM存放8笔资料，则每笔资料的平均可靠擦写次数为：

每笔资料的平均可靠擦写次数= $\frac{(512 - 2 - 2 \times 8)}{2 \times 8} \times 4 \times 20{,}000 = 2{,}470{,}000$次

2.8 读写速度

不同的资料量以及所调用的Data Flash page数量，如表 2-1所示。由于使用SRAM存放资料，在读取资料时仅需要低于1微秒的时间，就可以取出资料。而在调用新的Data Flash page的时候，直接将SRAM存放的资料存入可以将所需要的时间大幅缩小，不到100毫秒就可以完成。

HCLK = 50MHz	资料量8笔 调用 4 page Data Flash	资料量20笔 调用 4 page Data Flash	资料量20笔 调用 6 page Data Flash
Init_EEPROM()	8	13.28	13.28
Search_Valid_Page()	61.2	61.2	65.6
Write_Data()	52	62.8	62.8
Write_Data() Manage_Next_Page()	20400	20753	20753
Read_Data()	0.88	0.88	0.88

单位：微秒

表 2-1 读写速度

2.9 降低需要的 Data Flash page 数量

2.9.1 NuMicro® Cortex®-M0 系列

对于NuMicro® Cortex®-M0系列，如图 1-1所示，如果使用者需要存放的资料量增加，为了满足要求的可靠擦写次数，需要的Data Flash page数量会大量增加。

举例来说，如果使用者有100笔资料需要储存，为了满足200万次的可靠擦写次数，使用者必须使用65个Data Flash page，共32.5K字节模拟成EEPROM。但是储存50笔资料，仅需要25个Data Flash page，共12.5K字节模拟成EEPROM。因此，如果使用者将100笔资料分成2个50笔的资料群，仅需要50个Data Flash page，共25K字节模拟成EEPROM即可。如表 2-2所示。

资料量	Data Flash page数量
50	25
100	65
50 + 50	50

表 2-2 资料量大小与 Data Flash page 数量比较 – NuMicro® Cortex®-M0 系列 (512B/page)

因此，我们建议使用者可以将要存放的资料量分成数个较小的资料群，利用较少的Data Flash page数量就可以达到需要的可靠擦写次数。

2.9.2 NuMicro® Cortex®-M4 系列

对于NuMicro® Cortex®-M4系列，如图 1-2所示，如果使用者需要存放的资料量增加，为了满足要求的可靠擦写次数，需要增加的Data Flash page数量并不会大量增加。

举例来说，如果使用者有100笔资料需要储存，为了满足200万次的可靠擦写次数，使用者必须使用11个Data Flash page，共22K字节模拟成EEPROM。但是储存50笔资料，仍然需要6个Data Flash page，共12K字节模拟成EEPROM。如表 2-3所示。

资料量	Data Flash page数量
50	6
100	11
50 + 50	12

表 2-3 资料量大小与 Data Flash page 数量比较 – NuMicro® Cortex®-M4 系列 (2KB/page)

因此，使用者可以直接操作，不需将要资料量分成数个较小的资料群。

3 结论

本编提出一个新的机制，能够组合两个page以上的Data Flash来模拟EEPROM使用，并且利用SRAM加速资料读写的速度，同时满足超过百万次可靠的擦写次数。

经由特别写入的Counter值，使用者可以了解Data Flash page目前已经使用过的可擦写次数，有效掌握剩余的可靠擦写次数，进一步推算产品的可用年限。

对于NuMicro® Cortex®-M0系列，我们建议使用者可以将要存放的大量资料量分成数个较小的资料群，利用较少的Data Flash page数量达到需要的可靠擦写次数。而对于NuMicro® Cortex®-M4系列，使用者可以直接操作，不需将要资料量分成数个较小的资料群。

4 附录：范例程序码

4.1提供函数库的程序源码，使用者可以应用在自己的程序中；4.2提供范例程序源码，以NuMicro® M051 DE系列为例，使用4个Data Flash page模拟EEPROM储存8笔资料。在主程式中将不断写入数值，直到Counter值等于2万。

4.1 函数库

4.1.1 Init_EEPROM()

```
/**
* @brief     Initial Data Flash as EEPROM.
* @param[in]  data_size: The amount of user's data, unit in byte. The maximun amount is
                         128.
* @param[in] use_pages: The amount of page which user want to use.
* @retval    Err_OverPageSize: The amount of user's data is over than the maximun amount.
* @retval    Err_OverPageAmount: The amount of page which user want to use is over than
                                 the maximun amount.
* @retval    0: Success
*/
uint32_t Init_EEPROM(uint32_t data_amount, uint32_t use_pages)
{
        uint32_t i;

        /* The amount of data includes 1 byte address and 1 byte data */
        Amount_of_Data = data_amount;
        /* The amount of page which user want to use */
        Amount_Pages = use_pages;

        /* Check setting is valid or not */
        /* The amount of user's data is more than the maximun amount or not */
        if(Amount_of_Data > Max_Amount_of_Data)
                return   Err_OverAmountData;
        /* For M051 Series, the max. amount of Data Flash pages is 8 */
        if(Amount_Pages > 8)
                return   Err_OverPageAmount;

        /* Init SRAM for data */
        Written_Data = (uint8_t *)malloc(sizeof(uint8_t) * Amount_of_Data);
        /* Fill initial data 0xFF*/
```

```
        for(i = 0; i < Amount_of_Data; i++)
        {
                Written_Data[i] = 0xFF;
        }

        return 0;
}
```

```
/**
* @brief     Search which page has valid data and where is current cursor for the next
            data to write.
*/
void Search_Valid_Page(void)
{
        uint32_t i, temp;
        uint8_t  addr, data;
        uint16_t *Page_Status_ptr;

        /* Enable FMC ISP function */
        FMC_Enable();

        /* Set information of each pages to Page_Status */
        Page_Status_ptr = (uint16_t *)malloc(sizeof(uint16_t) * Amount_Pages);
        for(i = 0; i < Amount_Pages; i++)
        {
                Page_Status_ptr[i] = (uint16_t)FMC_Read(DataFlash_BaseAddr +
(FMC_FLASH_PAGE_SIZE * i));
        }

        /* Search which page has valid data */
        for(i = 0; i < Amount_Pages; i++)
        {
                if(Page_Status_ptr[i] != Status_Unwritten)
                        Current_Valid_Page = i;
        }
        /* If Data Flash is used for first time, set counter = 0 */
        if(Page_Status_ptr[Current_Valid_Page] == Status_Unwritten)
        {
                /* Set counter = 0 */
                FMC_Write(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE *
Current_Valid_Page), 0xFFFF0000);
```

```
                /* Set cursor to current Data Flash address */
                Current_Cursor = 2;
        }
        else
        {
                /* Search where is current cursor for the next data to write and get the
data has been written */
                /* Check even value */
                temp = FMC_Read(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE *
Current_Valid_Page));
                addr = (temp & Even_Addr_Mask) >> Even_Addr_Pos;
                data = (temp & Even_Data_Mask) >> Even_Data_Pos;
                /* Check Address is 0xFF (un-written) of not */
                if(addr == 0xFF)
                {
                        /* If Address is 0xFF, then set cursor to current Data Flash
address */
                        Current_Cursor = 2;
                }
                else
                {
                        /* Copy the address and data to SRAM */
                        Written_Data[addr] = data;

                        /* Check the whole Data Flash */
                        for(i = 4; i < FMC_FLASH_PAGE_SIZE; i += 4)
                        {
                                /* Check odd value */
                                temp = FMC_Read(DataFlash_BaseAddr +
(FMC_FLASH_PAGE_SIZE * Current_Valid_Page) + i);
                                addr = (temp & Odd_Addr_Mask) >> Odd_Addr_Pos;
                                data = (temp & Odd_Data_Mask) >> Odd_Data_Pos;
                                /* Check Address is 0xFF (un-written) of not */
                                if(addr == 0xFF)
                                {
                                        /* If Address is 0xFF, then set cursor to
current Data Flash address */
                                        Current_Cursor = i;
                                        break;
                                }
                                else
                                {
```

```
                                        /* Copy the address and data to SRAM */
                                        Written_Data[addr] = data;
                                }

                                /* Check even value */
                                addr = (temp & Even_Addr_Mask) >> Even_Addr_Pos;
                                data = (temp & Even_Data_Mask) >> Even_Data_Pos;
                                /* Check Address is 0xFF (un-written) of not */
                                if(addr == 0xFF)
                                {
                                        /* If Address is 0xFF, then set cursor to
current Data Flash address */
                                        Current_Cursor = i + 2;
                                        break;
                                }
                                else
                                {
                                        /* Copy the address and data to SRAM */
                                        Written_Data[addr] = data;
                                }
                        }
                }
        }
}
```

4.1.2 Write_Data()

```
/**
* @brief     Write one byte data to SRAM and current valid page.
             If this index has the same data, it will not changed in SRAM and Data Flash.
             If current valid page is full, execute Manage_Next_Page() to copy valid data
             to next page.
* @param[in] index: The index of data address.
* @param[in] data: The data that will be writeen.
* @retval    Err_ErrorIndex: The input index is not valid.
* @retval    0: Success
*/
uint32_t Write_Data(uint8_t index, uint8_t data)
{
        uint32_t temp = 0;
```

```
        /* Check the index is valid or not */
        if(index > Amount_of_Data)
        {
                return Err_ErrorIndex;
        }
        /* If the writing data equals to current data, the skip the write process */
        if(Written_Data[index] == data)
        {
                return 0;
        }

        /* Enable FMC ISP function */
        FMC_Enable();

        /* Current cursor points to odd position*/
        if((Current_Cursor & 0x3) == 0)
        {
                /* Write data to Data Flash */
                temp = 0xFFFF0000 | (index << Odd_Addr_Pos) | (data << Odd_Data_Pos);
                FMC_Write(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE * Current_Valid_Page)
+ Current_Cursor, temp);
                /* Write data to SRAM */
                Written_Data[index] = data;
        }
        /* Current cursor points to even position*/
        else
        {
                /* Read the odd position data */
                temp = FMC_Read(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE *
Current_Valid_Page) + (Current_Cursor - 2));
                /* Combine odd position data and even position data */
                temp &= ~(Even_Addr_Mask | Even_Data_Mask);
                temp |= (index << Even_Addr_Pos) | (data << Even_Data_Pos);
                /* Write data to Data Flash */
                FMC_Write(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE * Current_Valid_Page)
+ (Current_Cursor - 2), temp);
                /* Write data to SRAM */
                Written_Data[index] = data;
        }

        /* If current cursor points to the last position, then execute Manage_Next_Page()
*/
```

```
        if(Current_Cursor == (FMC_FLASH_PAGE_SIZE - 2))
        {
                /* Copy valid data to next page */
                Manage_Next_Page();
        }
        /* Add current cursor */
        else
        {
                /* Set current cursor to next position */
                Current_Cursor += 2;
        }

        return 0;
}
```

```
/**
* @brief     Manage the valid data from SRAM to new page.
*/
void Manage_Next_Page(void)
{
        uint32_t i = 0, j, counter, temp = 0, data_flag = 0, new_page;

        /* Copy the valid data (not 0xFF) from SRAM to new valid page */
        /* Get counter from the first two bytes */
        counter = FMC_Read(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE *
Current_Valid_Page));

        /* If current valid page is the last page, choose the first page as valid page */
        if((Current_Valid_Page + 1) == Amount_Pages)
        {
                new_page = 0;
                /* Add counter to record 1 E/W cycle finished for all pages */
                counter++;
        }
        else
        {
                new_page = Current_Valid_Page + 1;
        }

        /* Enable FMC ISP function */
```

```
        FMC_Enable();

        /* Copy first valid data */
        while(1)
        {
                /* Not a valid data, skip */
                if(Written_Data[i] == 0xFF)
                {
                        i++;
                }
                /* Combine counter and first valid data, and write to new page */
                else
                {
                        counter &= ~(Even_Addr_Mask | Even_Data_Mask);
                        counter |= (i << Even_Addr_Pos) | (Written_Data[i] <<
Even_Data_Pos);
                        FMC_Write(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE * new_page),
counter);
                        i++;
                        break;
                }
        }
        /* Copy the rest of data */
        for(j = 4; i < Amount_of_Data; i++)
        {
                /* Not a valid data, skip */
                if(Written_Data[i] == 0xFF)
                {
                        continue;
                }
                /* Write to new page */
                else
                {
                        /* Collect two valid data and write to Data Flash */
                        /* First data, won't write to Data Flash immediately */
                        if(data_flag == 0)
                        {
                                temp |= (i << Odd_Addr_Pos) | (Written_Data[i] <<
Odd_Data_Pos);
                                data_flag = 1;
                        }
```

```
                        /* Second data, write to Data Flash after combine with first
data */
                        else
                        {
                                temp |= (i << Even_Addr_Pos) | (Written_Data[i] <<
Even_Data_Pos);
                                FMC_Write(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE *
new_page) + j, temp);
                                temp = 0;
                                data_flag = 0;
                                j += 4;
                        }
                }
        }

        /* Set cursor to new page */
        Current_Cursor = j;

        /* If there is one valid data left, write to Data Flash */
        if(data_flag == 1)
        {
                temp |= 0xFFFF0000;
                FMC_Write(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE * new_page) + j,
temp);
                Current_Cursor += 2;
        }

        /* Erase the old page */
        FMC_Erase(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE * Current_Valid_Page));
        /* Point to new valid page */
        Current_Valid_Page = new_page;
}
```

4.1.3 Read_Data()

```
/**
* @brief      Read one byte data from SRAM.
* @param[in]  index: The index of data address.
* @param[in] data: The data in the index of data address from SRAM.
* @retval    Err_ErrorIndex: The input index is now valid.
* @retval    0: Success
*/
```

```
uint32_t Read_Data(uint8_t index, uint8_t *data)
{

        /* Check the index is valid or not */
        if(index >= Max_Amount_of_Data)
        {
                return Err_ErrorIndex;
        }

        /* Get the data from SRAM */
        *data = Written_Data[index];

        return 0;
}
```

4.1.4 Get_Cycle_Counter()

```
/**
* @brief     Get the cycle counter for how many cycles has page been erased/programmed.
* @retval    Cycle_Conter: The cycles that page has been erased/programmed.
*/
uint16_t Get_Cycle_Counter(void)
{
        uint16_t Cycle_Counter;

        /* Get the cycle counter from first two bytes in current Data Flash page */
        Cycle_Counter = (uint16_t)FMC_Read(DataFlash_BaseAddr + (FMC_FLASH_PAGE_SIZE *
Current_Valid_Page));

        return Cycle_Counter;
}
```

4.2 范例程序源码

```
#include <stdio.h>
#include "M051Series.h"
#include "EEPROM_Emulate.h"

#define PLLCON_SETTING      CLK_PLLCON_50MHz_HXT
#define PLL_CLOCK           50000000
```

```
#define Test_data_size         8
#define Test_page_amount       4

void SYS_Init(void)
{
    /*---------------------------------------------------------------------------------------------*/
    /* Init System Clock                                                                           */
    /*---------------------------------------------------------------------------------------------*/

    /* Enable External XTAL (4~24 MHz) */
    CLK->PWRCON |= CLK_PWRCON_XTL12M_EN_Msk;

    CLK->PLLCON = PLLCON_SETTING;

    /* Waiting for clock ready */
    CLK_WaitClockReady(CLK_CLKSTATUS_PLL_STB_Msk | CLK_CLKSTATUS_XTL12M_STB_Msk);

    /* Switch HCLK clock source to PLL */
    CLK->CLKSEL0 = CLK_CLKSEL0_HCLK_S_PLL;

    /* Update System Core Clock */
    /* User can use SystemCoreClockUpdate() to calculate PllClock, SystemCoreClock and
CycylesPerUs automatically. */
    SystemCoreClockUpdate();
    PllClock        = PLL_CLOCK;            // PLL
    SystemCoreClock = PLL_CLOCK / 1;        // HCLK
    CyclesPerUs     = PLL_CLOCK / 1000000;  // For SYS_SysTickDelay()
}

int main()
{
    uint32_t i;
    uint8_t u8Data;

    /* Unlock protected registers */
    SYS_UnlockReg();

    SYS_Init();

    /* Test Init_EEPROM() */
    Init_EEPROM(Test_data_size, Test_page_amount);
```

```
    /* Test Search_Valid_Page() */
    Search_Valid_Page();

    /* Test Write_Data() */
    for(i = 0; i < 254; i++)
    {
        Write_Data(i%Test_data_size, i%256);
    }

    /* Test Write_Data() contain Manage_Next_Page() */
    Write_Data(i%Test_data_size, 0xFF);

    /* Test Read_Data() */
    Read_Data(0x7, &u8Data);

    /* Test Write over 20000 times */
    while(Get_Cycle_Counter() < 20000)
    {
        for(i = 0; i < 247; i++)
        {
            Write_Data(i%Test_data_size, i%256);
        }
    }

    while(1);
}
```

5 版本历史

日期	版本	描述
2016.09.20	1.00	1. 初次发布
2019.09.18	1.01	1. 修改编号数字

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.