使用Data Flash模擬EEPROM並且利用SRAM加速資料讀寫的速度

Application Note for 32-bit NuMicro® Family

Document Information

Abstract	這份文件介紹如何使用Data Flash模擬EEPROM，並且利用SRAM加速資料讀寫的速度。內容包含原理介紹、性能數據以及使用建議。附錄提供範例程式以及函式庫。
Apply to	NuMicro® Cortex®-M0/M4全系列，本文以NuMicro® M051 DE系列為例。

目錄

1 簡介

EEPROM具有可byte write/byte read以及高達百萬次可靠的擦寫次數，通常被使用者用來存放程式中會時常變更的非揮發性資料。對於單晶片產品，通常不具有內建的EEPROM能夠提供給使用者，而是基於Flash來存放使用者的資料。但是Flash的擦寫次數無法與EEPROM比擬。

現在我們提出一個機制，能夠組合兩個page以上的Data Flash來模擬EEPROM使用，使用SRAM加速讀寫資料速度、能夠達到百萬次可靠的擦寫次數、記錄擦寫循環次數，並且可以將資料量分成數個較小的資料群以減少Data Flash page數量。

- 使用 SRAM 加速讀寫資料速度

當進行資料寫入的時候，會同步寫入Data Flash和SRAM；一旦當下使用的Data Flash page已經寫滿，就會使用下一個Data Flash page，並且能夠直接將SRAM存放的資料存入，節省一般將資料移入新的Data Flash page，需要搜尋全部Data Flash中已寫入的資料的時間。

當需要讀取資料的時候，可以直接由SRAM中讀出資料，無須從Data Flash中尋找需要的資料，能夠節省搜尋過程的時間，Data Flash中的資料只用來初始化SRAM。

- 達到百萬次可靠的擦寫次數

這樣使用Data Flash模擬EEPROM的方法，能夠讓使用進行byte write/byte read以及超過百萬次可靠的擦寫次數。

- 記錄擦寫循環次數

使用者可以經由寫入的Counter值了解Data Flash page的擦寫循環次數。

- 將資料量分成數個較小的資料群以減少 Data Flash page 數量

對於NuMicro® Cortex®-M0系列，如圖 1-1所示，如果使用者需要存放的資料量增加，為了滿足要求的可靠擦寫次數，需要的Data Flash page數量會大量增加。因此，我們建議使用者可以將要存放的資料量分成數個較小的資料群，利用較少的Data Flash page數量就可以達到需要的可靠擦寫次數。

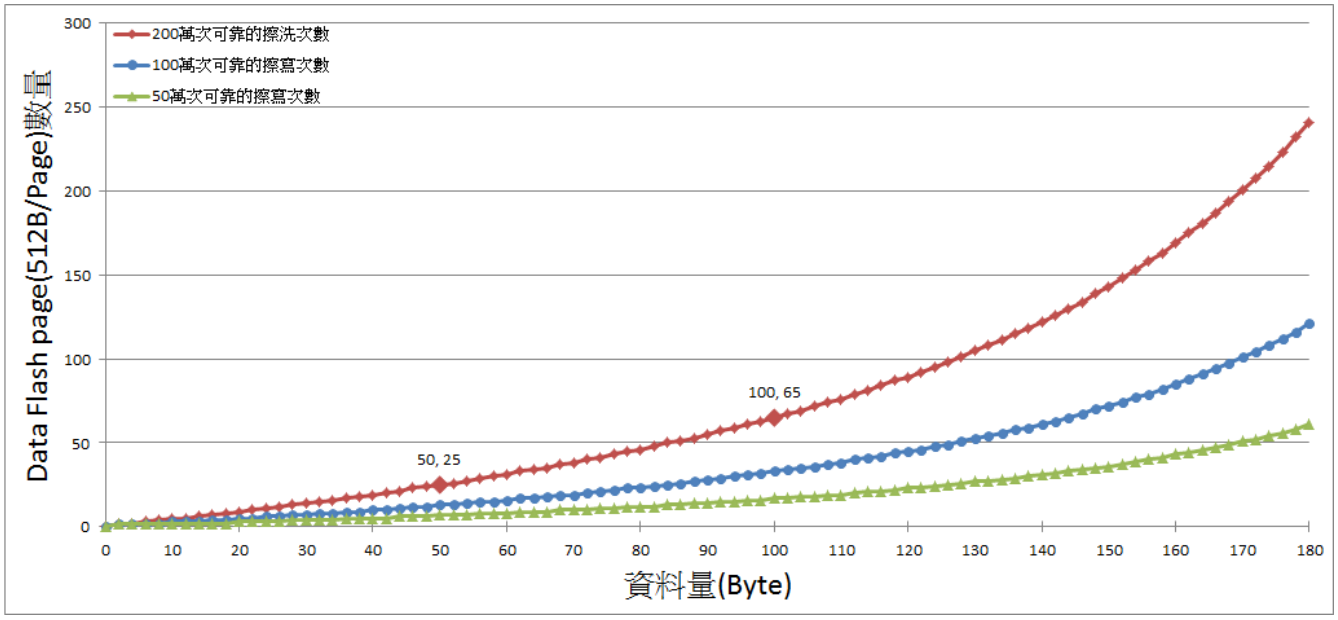


圖 1-1 資料量與需要的 Data Flash page 數量 - NuMicro® Cortex®-M0 系列

資料量(Byte)	Data Flash page(512B/page)數量	資料量(Byte)	Data Flash page(512B/page)數量
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 常用資料量與需要的 Data Flash page 數量- NuMicro® Cortex®-M0 系列

對於NuMicro® Cortex®-M4系列，如圖 1-2所示，如果使用者需要存放的資料量增加，為了滿足要求的可靠擦寫次數，需要增加的Data Flash page數量並不會大量增加。因此，使用者可以直接操作，不需將要資料量分成數個較小的資料群。

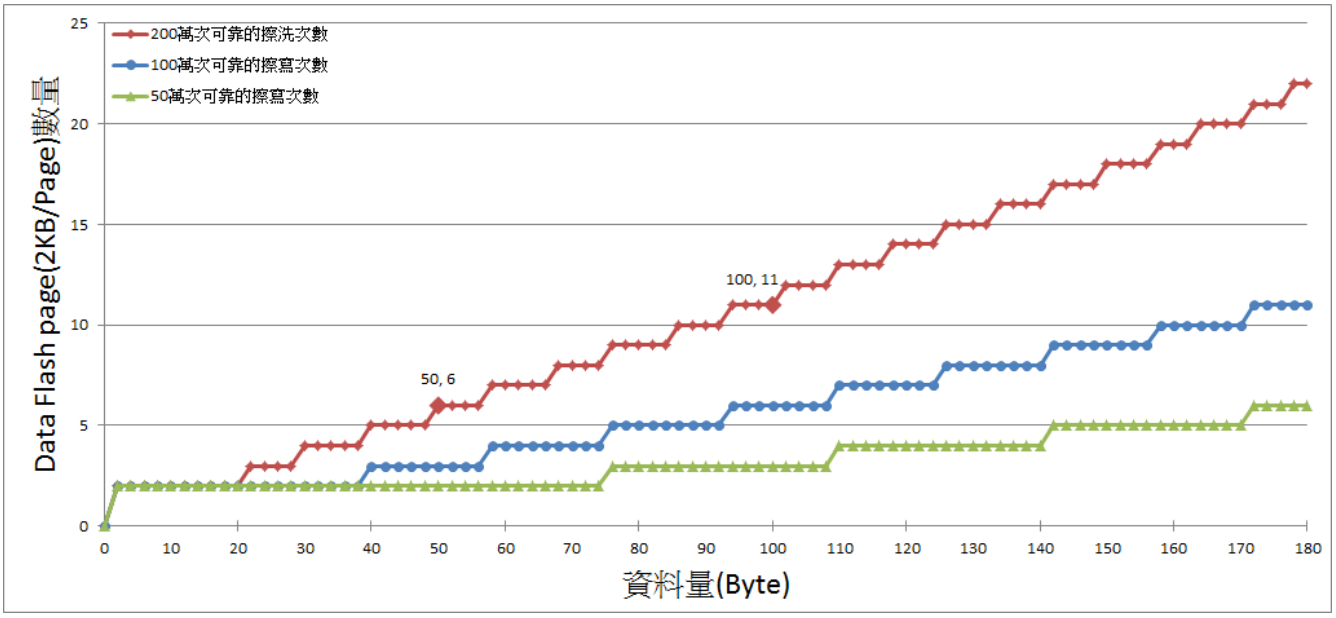


圖 1-2 資料量與需要的Data Flash page數量 - NuMicro® Cortex®-M4系列

資料量(Byte)	Data Flash page(2KB/page)數量	資料量(Byte)	Data Flash page(2KB/page)數量
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 常用資料量與需要的 Data Flash page 數量- NuMicro® Cortex®-M4 系列

2 模擬EEPROM

本節將介紹如何使用Data Flash模擬EEPROM，並且利用SRAM加速資料讀寫的速度。

2.1 機制原理

在使用Data Flash模擬EEPROM時，使用者必須使用至少2個page以上的Data Flash，並且將每一page的Data Flash，以每兩個位元組為單位劃分成若干個區塊，如圖 2-1所示，以NuMicro® Cortex®-M0系列為例，每一個page的Data Flash大小為512位元組。第一個區塊紀錄目前擦寫循環次數的Counter值，其餘存放資料的位址和值。同時SRAM也劃分一個區塊，用來記錄存放的資料。

當使用者要存放資料的時候，會將資料的位址和值依序寫到第一個Data Flash page，並且將值寫入對應位址的SRAM。如果第一個Data Flash page的空間被寫滿時，會將目前已寫入的有效資料(非0xFF) 從SRAM整理到第二個Data Flash page，並清除第一個Data Flash page。

使用SRAM來存放資料，可以減少從Data Flash讀取資料的時間；當Data Flash page寫滿要將資料整理到下一個Data Flash page的時候，直接從SRAM讀出資料，無須從Data Flash中尋找需要的資料，能夠節省搜尋Data Flash中有效資料的時間。

Counter值能夠幫助使用者了解，目前已經使用過的可擦寫次數。使用者可以有效掌握剩餘的可靠擦寫次數，推算產品的可用年限。

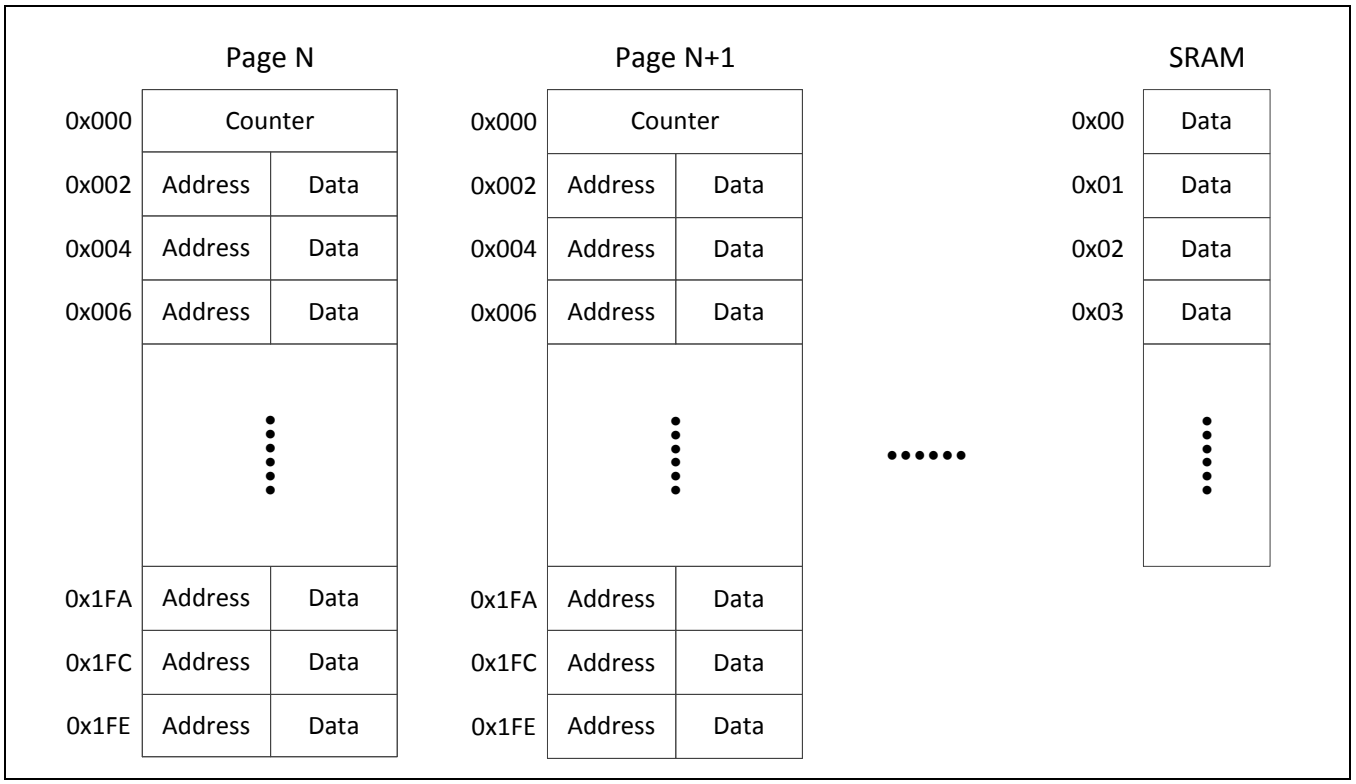


圖 2-1 Data Flash page 和 SRAM

2.2 初始化 Data Flash

在使用者開始執行主程式前，首先需要初始化要調用的Data Flash page。由於只有保存資料的Data Flash page具有有效的Counter值(非0xFFFF)，我們可以透過尋找有效的Counter值來找出保存資料的Data Flash page。接著除了指向可以寫入資料的位址，還需要將已保存的資料寫到SRAM。詳細流程請參考圖 2-3，並請參考函式`Init_EEPROM()`和`Search_Valid_Page()`。

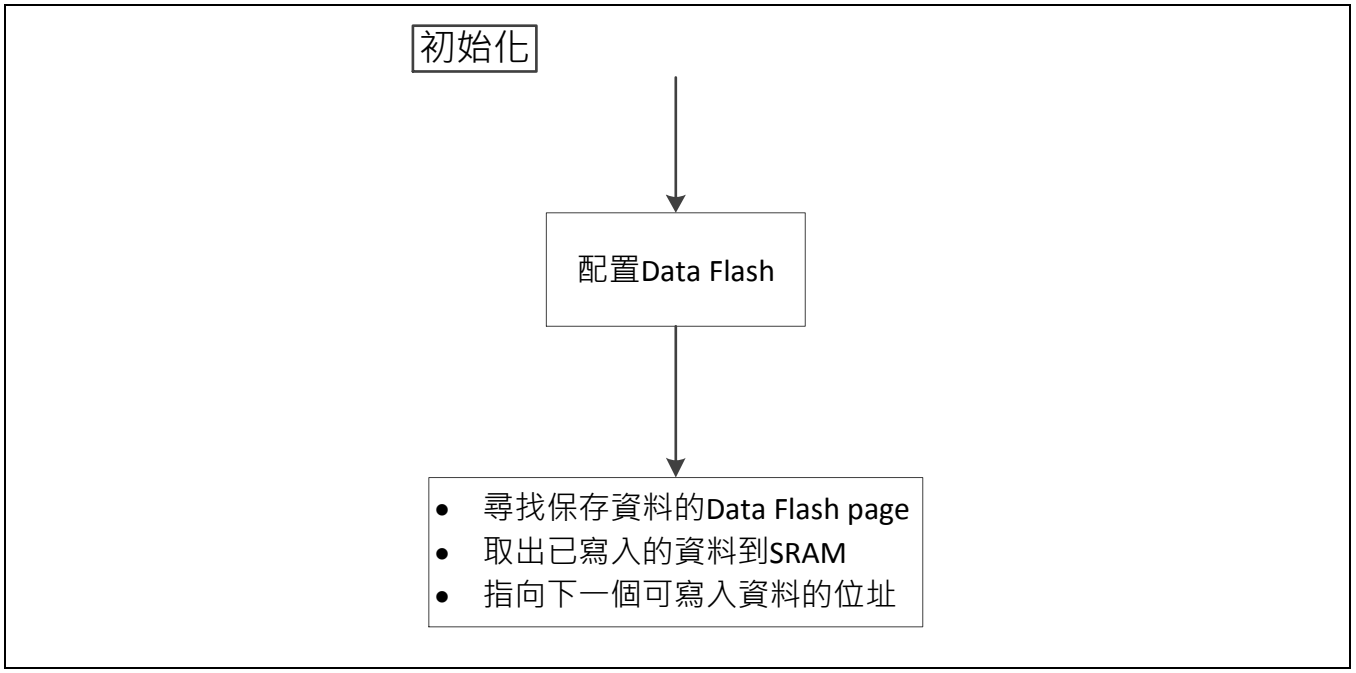


圖 2-2 Data Flash初始化流程

2.3 寫入資料

當使用者有資料寫入的時候，首先比對是否已與SRAM中的資料相同。如果資料相同，則不需要執行寫入，略過寫入資料步驟；如果資料不同，則需要更新資料。

資料寫入時，會同時寫入到Data Flash和SRAM，接著判斷目前使用的Data Flash page是否已經寫滿。如果還有未寫入的空間，就指向下一個位址；如果Data Flash page已經寫滿，則將SRAM中有效的資料(非0xFF)寫入下一個調用的Data Flash page儲存。

相較於一般需要搜尋整個Data Flash page的方式，直接從SRAM中寫入有效資料，可以大幅縮短當目前Data Flash page已經寫滿，要調用下一個Data Flash page的時間。

如果目前使用的Data Flash page已經是可調用的最後一個Data Flash page，就將Counter增加計數1，再將SRAM中有效的資料(非0xFF)寫入第一個可調用的Data Flash page。

使用者可以讀取Counter的值，了解目前已經使用過的可擦寫次數。

完成資料寫入後，清除已寫滿的Data Flash page。當有新的資料需要寫入的時候，將寫入新的Data Flash page。

詳細流程請參考圖 2-3，並請參考函式Write_Data()和Manage_Next_Page()。

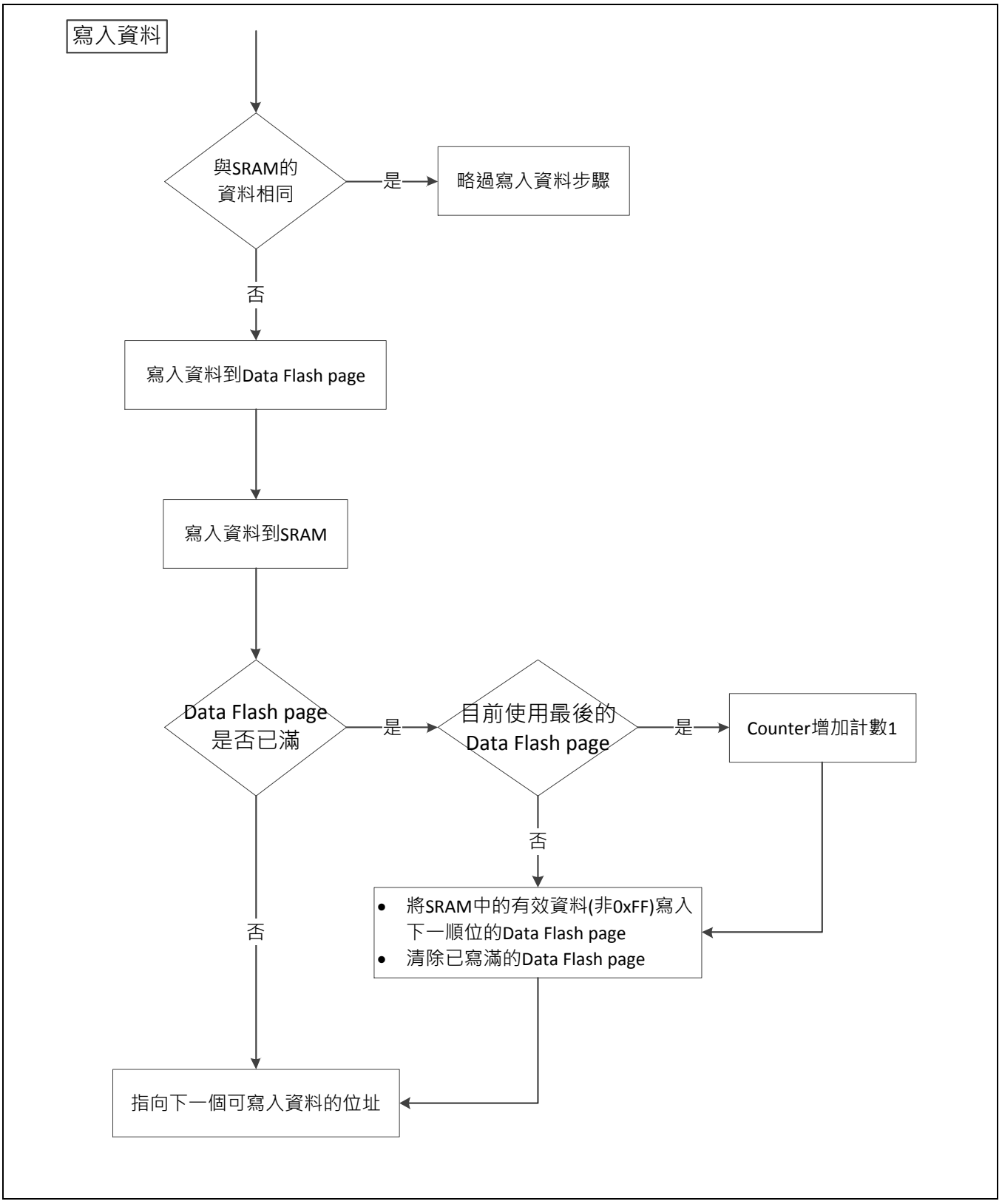


圖 2-3 寫入資料流程

2.4 讀取資料

讀取資料的時候，只需要將SRAM中的值讀出即可，不需要從Data Flash page中尋找最後更新的資料，可以大幅節省讀取資料需要的時間。詳細流程請參考圖 2-4，並請參考函式 `Read_Data()`。

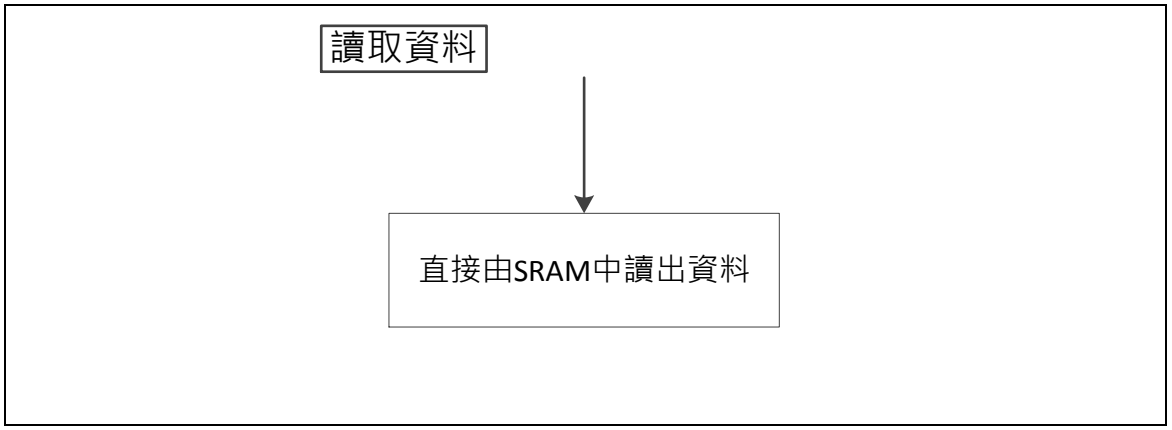


圖 2-4 讀取資料流程

2.5 讀取 Counter 值

使用者可以藉由讀取Counter的值了解目前Data Flash page已經使用多少次可擦寫循環，並推算目前剩餘的可靠擦寫次數。請參考函式`Get_Cycle_Counter()`。

2.6 資料讀寫範例

在主程式運行之前，首先需要初始化Data Flash。如圖 2-5所示，我們選擇組合兩個page的Data Flash來模擬EEPROM使用。初始化過程會先找出目前儲存有效資料的Data Flash page，Page 1，並將有效資料(非0xFF)放入SRAM。此範例中儲存的資料量為8個位元組，在SRAM中以陣列存放這些資料。最後指向下一個可以寫入資料的位址。

當使用者要寫入0x07[0x68] (位址[資料]) 的時候，如圖 2-6所示。首先會判斷與SRAM中的資料是否相同。由於目前SRAM中的值與0x68不相等，因此會將資料的位址和值寫入Data Flash的區塊，並且將值寫入對應位址0x07的SRAM陣列中。因為目前的Data Flash page並未寫滿，僅需要指向下一個可以寫入資料的位址即可。

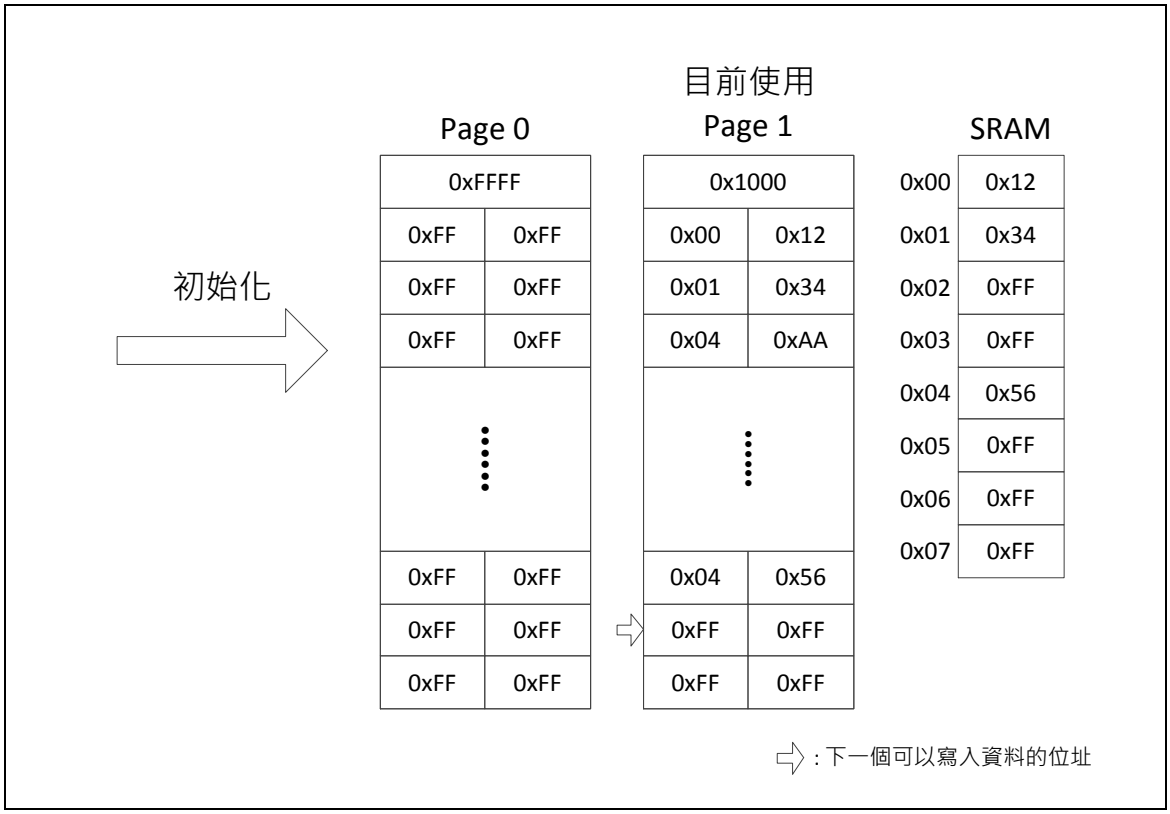


圖 2-5 資料讀寫流程範例 – 初始化

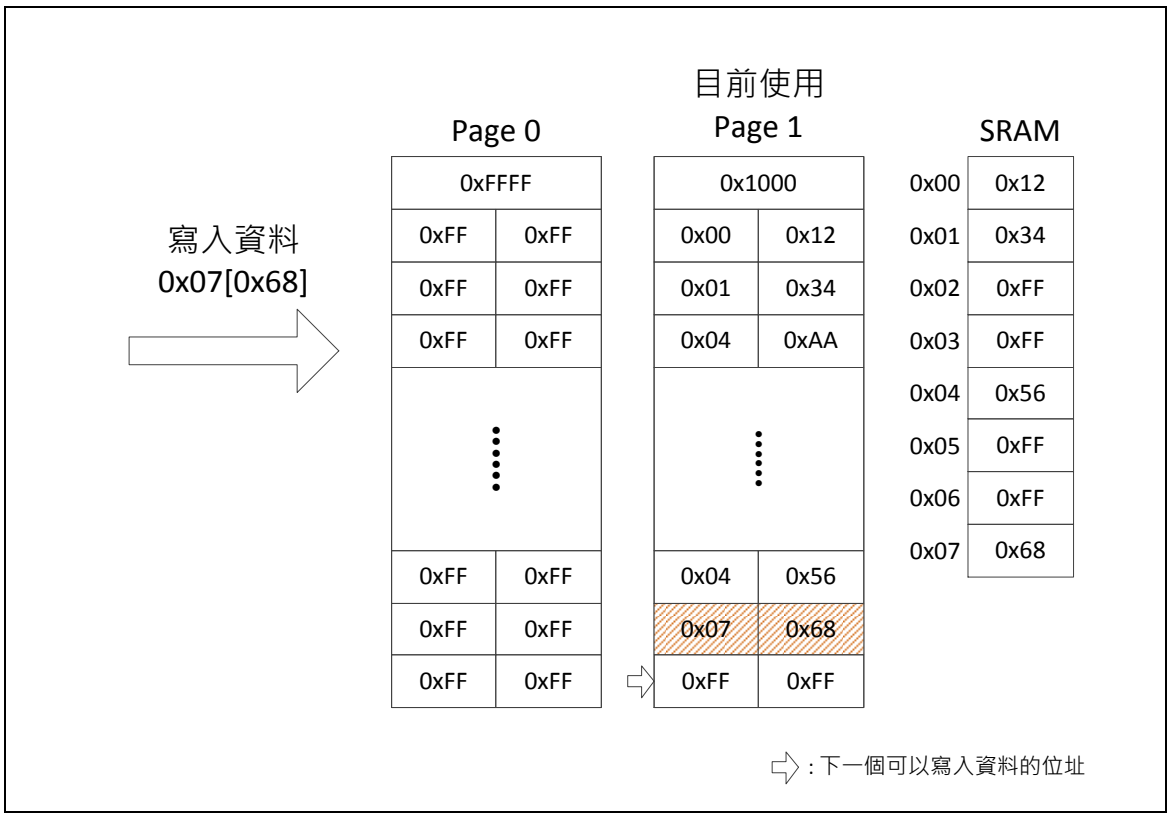


圖 2-6 資料讀寫流程範例 – 寫入新資料

如果使用者要寫入資料0x04[0x56]，如圖 2-7所示。由於目前SRAM中的值與0x56相等，因此略過寫入資料的步驟。

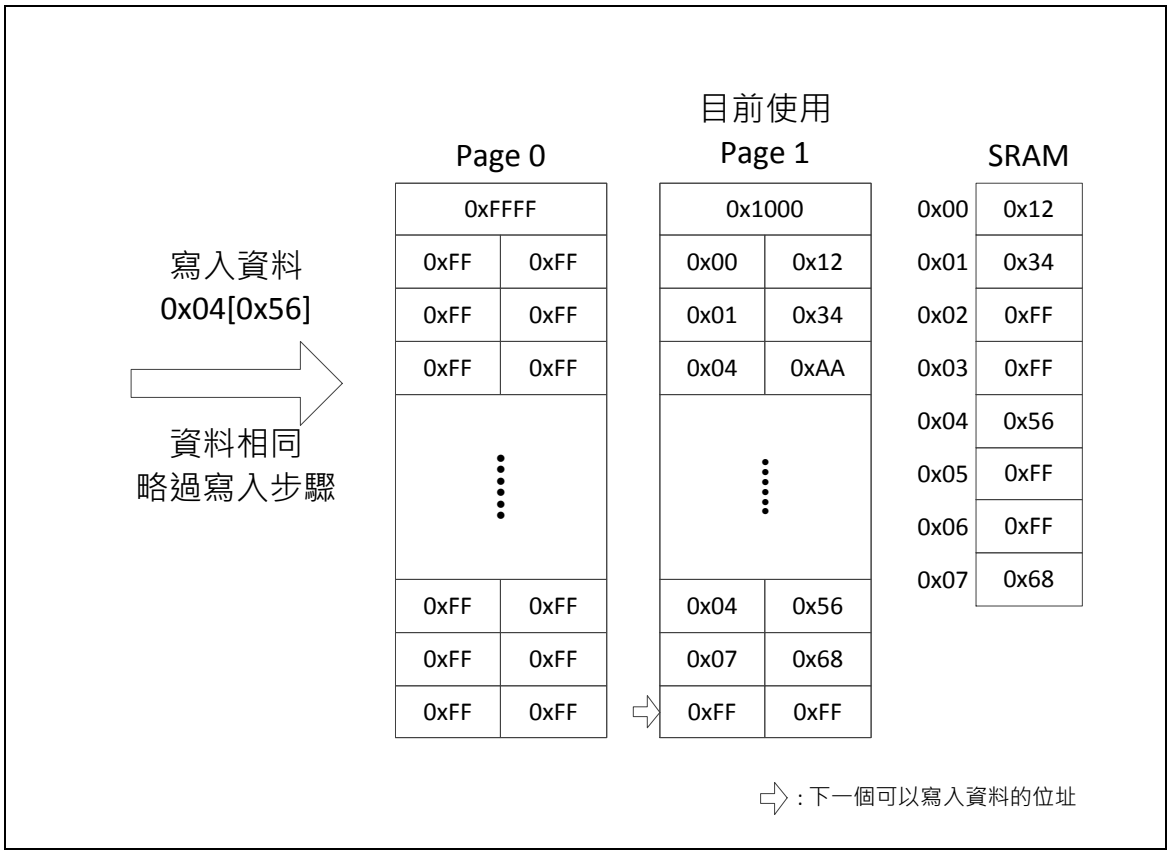


圖 2-7 資料讀寫流程範例 – 寫入相同資料

接著，當使用者要寫入資料0x00[0xFF]的時候，如圖 2-6所示。首先會判斷與SRAM中的資料是否相同。由於目前SRAM中的值與0xFF不相等，因此會將資料的位址和值寫入Data Flash的區塊，並且將值寫入對應位址0x00的SRAM陣列中。

這時目前的Data Flash page已經寫滿，需要將SRAM的資料寫到新的Data Flash page。而且目前使用的Data Flash page已經是可調用的最後一個Data Flash page，因此將Counter增加計數1。再將SRAM中有效的資料(非0xFF)寫入第一個調用的Data Flash page，所以僅將位址0x01、0x04、0x06、0x07的值寫入到新調用的Data Flash page。

完成資料寫入後，清除已寫滿的Data Flash page，最後指向下一個可以寫入資料的位址。

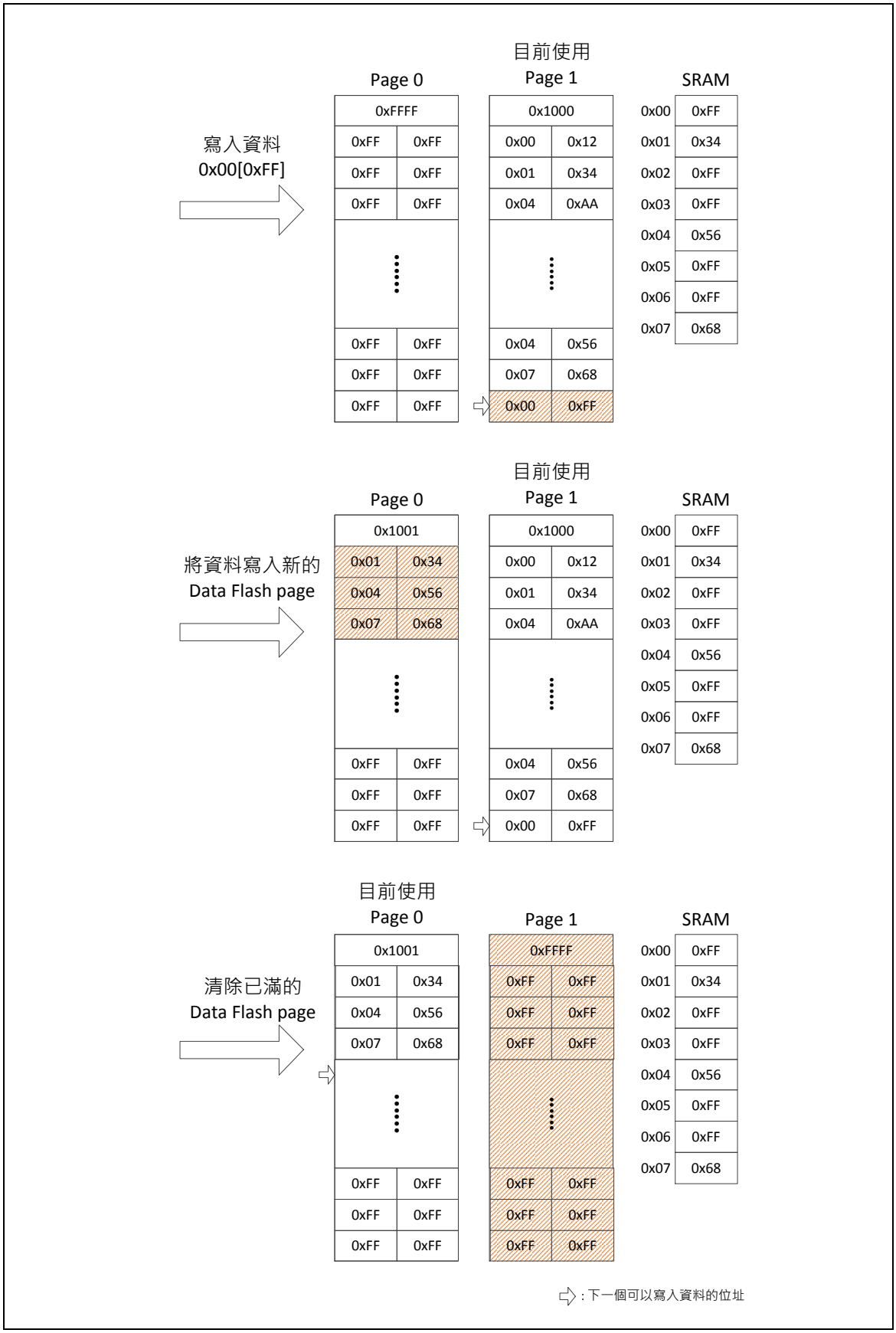
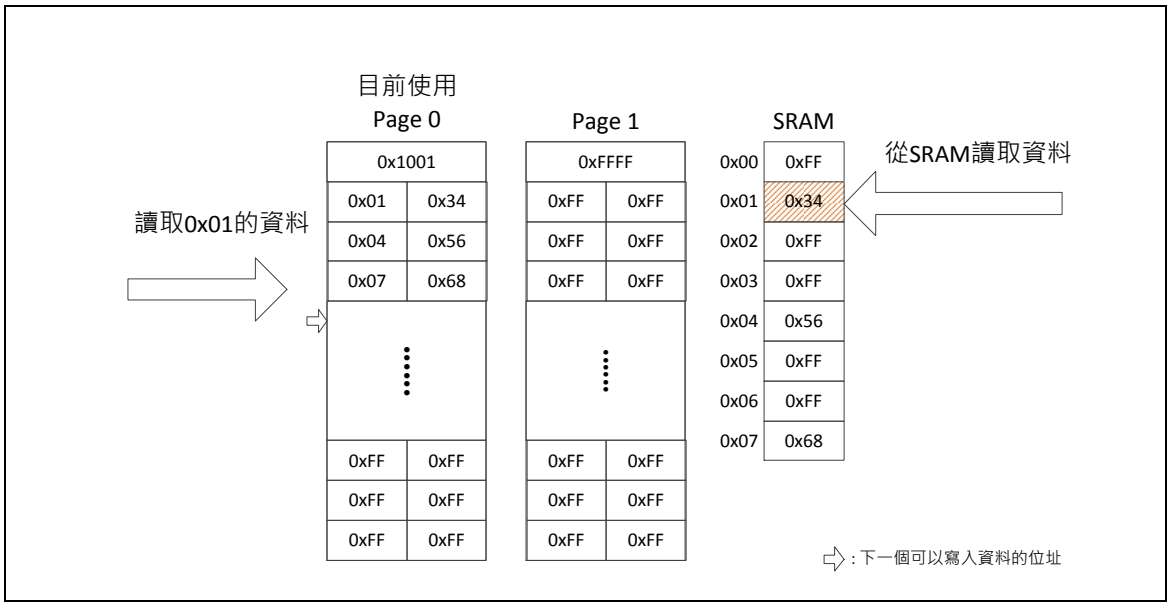

寫入資料
0x00[0xFF]
目前使用
Page 0
Page 1
SRAM
0xFFFF
0x1000
0x00 0x12
0x01 0x34
0x04 0xAA
0x04 0x56
0x07 0x68
0x00 0xFF
0x00 0xFF
0x01 0x34
0x02 0xFF
0x03 0xFF
0x04 0x56
0x05 0xFF
0x06 0xFF
0x07 0x68
將資料寫入新的
Data Flash page
0x1001
0x01 0x34
0x04 0x56
0x07 0x68
清除已滿的
Data Flash page
0xFF 0xFF
: 下一個可以寫入資料的位址

圖 2-8 資料讀寫流程範例 – 寫入新資料並調用下一個 Data Flash page

當使用者讀取資料的時候，只需要將SRAM中的值讀出即可。



2.7 可靠度計算

在計算可靠度的時候，我們以每筆資料的平均可靠擦寫次數做為評估基準。為了滿足EEPROM的使用性，每筆資料的平均可靠擦寫次數都需要在一百萬次以上。而每筆資料的平均可靠擦寫次數公式如下：

$$\text{每筆資料的平均可靠擦寫次數} = \frac{(S - C - D)}{D} \times P \times E$$

S: Data Flash page大小，在NuMicro® Cortex®-M0系列中為512位元組，在NuMicro® Cortex®-M4系列中為2048位元組。

C: Counter使用大小， 固定為2位元組。

D: 每筆資料使用大小 × 資料量，其中每筆資料使用大小固定為2位元組。

P: 使用的Data Flash page數量，由使用者選擇調用多少Data Flash page模擬EEPROM。

E: Data Flash page可靠的可擦寫次數。

以NuMicro® M051 DE系列為例，如果使用者調用4個Data Flash page，共2K位元組模擬成EEPROM存放8筆資料，則每筆資料的平均可靠擦寫次數為：

$$\text{每筆資料的平均可靠擦寫次數} = \frac{(512 - 2 - 2 \times 8)}{2 \times 8} \times 4 \times 20{,}000 = 2{,}470{,}000\text{次}$$

2.8 讀寫速度

不同的資料量以及所調用的Data Flash page數量，如表 2-1所示。由於使用SRAM存放資料，在讀取資料時僅需要低於1微秒的時間，就可以取出資料。而在調用新的Data Flash page的時候，直接將SRAM存放的資料存入可以將所需要的時間大幅縮小，不到100毫秒就可以完成。

HCLK = 50MHz	資料量8筆 調用4 page Data Flash	資料量20筆 調用4 page Data Flash	資料量20筆 調用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()` 並且 `Manage_Next_Page()`	20400	20753	20753
`Read_Data()`	0.88	0.88	0.88

單位：微秒

表 2-1 讀寫速度

2.9 降低需要的 Data Flash page 數量

2.9.1 NuMicro® Cortex®-M0 系列

對於NuMicro® Cortex®-M0系列，如圖 1-1所示，如果使用者需要存放的資料量增加，為了滿足要求的可靠擦寫次數，需要的Data Flash page數量會大量增加。

舉例來說，如果使用者有100筆資料需要儲存，為了滿足200萬次的可靠擦寫次數，使用者必須使用65個Data Flash page，共32.5K位元組模擬成EEPROM。但是儲存50筆資料，僅需要25個Data Flash page，共12.5K位元組模擬成EEPROM。因此，如果使用者將100筆資料分成2個50筆的資料群，僅需要50個Data Flash page，共25K位元組模擬成EEPROM即可。如表 2-2所示。

資料量	Data Flash page數量
50	25
100	65
50 + 50	50

表 2-2 資料量大小與 Data Flash page 數量比較 - NuMicro® Cortex®-M0 系列

因此，我們建議使用者可以將要存放的資料量分成數個較小的資料群，利用較少的Data Flash page數量就可以達到需要的可靠擦寫次數。

2.9.2 NuMicro® Cortex®-M4 系列

對於NuMicro® Cortex®-M4系列，如圖 1-2所示，如果使用者需要存放的資料量增加，為了滿足要求的可靠擦寫次數，需要增加的Data Flash page數量並不會大量增加。

舉例來說，如果使用者有100筆資料需要儲存，為了滿足200萬次的可靠擦寫次數，使用者必須使用11個Data Flash page，共22K位元組模擬成EEPROM。但是儲存50筆資料，仍然需要6個Data Flash page，共12K位元組模擬成EEPROM。如表 2-3所示。

資料量	Data Flash page數量
50	6
100	11
50 + 50	12

表 2-3 資料量大小與 Data Flash page 數量比較 - NuMicro® Cortex®-M4 系列

因此，使用者可以直接操作，不需將要資料量分成數個較小的資料群。

3 結論

本篇提出一個新的機制，能夠組合兩個page以上的Data Flash來模擬EEPROM使用，並且利用SRAM加速資料讀寫的速度，同時滿足超過百萬次可靠的擦寫次數。

經由特別寫入的Counter值，使用者可以了解Data Flash page目前已經使用過的可擦寫次數，有效掌握剩餘的可靠擦寫次數，進一步推算產品的可用年限。

對於NuMicro® Cortex®-M0系列，我們建議使用者可以將要存放的大量資料量分成數個較小的資料群，利用較少的Data Flash page數量達到需要的可靠擦寫次數。而對於NuMicro® Cortex®-M4系列，使用者可以直接操作，不需將要資料量分成數個較小的資料群。

4 附錄：範例程式碼

Error! Reference source not found.提供函式庫的程式碼，使用者可以應用在自己的程式中；4.1.2提供範例程式碼，以NuMicro® M051 DE系列為例，使用4個Data Flash page模擬EEPROM儲存8筆資料。在主程式中將不斷寫入數值，直到Counter值等於2萬。

4.1 函式庫

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 範例程式碼

```
#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 版本歷史

日期	版本	描述
2015.09.11	1.00	1. 初次發佈

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.