The following document contains information on Cypress products. Although the document is marked with the name "Spansion" and "Fujitsu", the company that originally developed the specification, Cypress will continue to offer these products to new and existing customers.

Continuity of Specifications
There is no change to this document as a result of offering the device as a Cypress product. Any changes that have been made are the result of normal document improvements and are noted in the document history page, where supported. Future revisions will occur when appropriate, and changes will be noted in a document history page.

Continuity of Ordering Part Numbers
Cypress continues to support existing part numbers. To order these products, please use only the Ordering Part Numbers listed in this document.

For More Information
Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress
Cypress (NASDAQ: CY) delivers high-performance, high-quality solutions at the heart of today's most advanced embedded systems, from automotive, industrial and networking platforms to highly interactive consumer and mobile devices. With a broad, differentiated product portfolio that includes NOR flash memories, F-RAM™ and SRAM, Traveo™ microcontrollers, the industry's only PSoC® programmable system-on-chip solutions, analog and PMIC Power Management ICs, CapSense® capacitive touch-sensing controllers, and Wireless BLE Bluetooth® Low-Energy and USB connectivity solutions, Cypress is committed to providing its customers worldwide with consistent innovation, best-in-class support and exceptional system value.

F²MC-16LX FAMILY
16-BIT MICROCONTROLLER
MB90F897

EEPROM EMULATION WITH DUAL OPERATION FLASH

APPLICATION NOTE

FUJITSU

Revision History

Date	Issue
2003-07-09	MWi, First version
2003-07-21	MWi, V1.1, upgraded
2005-12-28	MWi, V1.2, `FWR0/1` defines in code removed

This document contains 17 pages.

Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH restricts its warranties and its liability for **all products delivered free of charge** (eg. software include or header files, application examples, target boards, evaluation boards, engineering samples of IC's etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling. **Note, all these products are intended and must only be used in an evaluation laboratory environment**.

1. Fujitsu Microelectronics Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days form the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Microelectronics Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.

2. Should a Product turn out to be defect, Fujitsu Microelectronics Europe GmbH´s entire liability and the customer´s exclusive remedy shall be, at Fujitsu Microelectronics Europe GmbH´s sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Microelectronics Europe GmbH in original packing and without further defects resulting from the customer´s use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Microelectronics Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Microelectronics Europe GmbH.

3. To the maximum extent permitted by applicable law Fujitsu Microelectronics Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.

4. To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH´s and its suppliers´ liability is restricted to intention and gross negligence.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES

To the maximum extent permitted by applicable law, in no event shall Fujitsu Microelectronics Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect

Contents

0 Introduction

This application note describes how to implement an in-circuit EEPROM emulation with the Dual Operation Flash in the MB90F897.

1 EEPROM Emulation

EEPROM EMULATION EXAMPLE OF THIS APPLICATION NOTE

1.1 Purpose

Often it is useful in an application that some data have to be stored to an "solid-state memory", so that it is retained after power-off of the system. Usually EEPROMs are used for that purpose.

With a (Single operation) Flash device the emulation of an EEPROM is quite complex, because the Flash Memory cannot be cleared/programmed while code is executed in it. A programming software has to be moved to the RAM area first, and then the Flash content can be manipulated.

The Dual Operation Flash offers the ability to update one of the two Flash-banks while code is executed in the other bank. Therefore there is no need to use a program code which is transferred to the RAM first.

There exists one little difference between an EEPROM and Flash Memory: In a Flash Memory a single cell cannot be cleared – only Memory areas (the so called sectors) can be erased.

To emulate an EEPROM behavior nevertheless, a little trick has to be done. Let's say we need 4K of an EEPROM Memory. For this we have to use *two* 4K-Flash-Sectors. To (over-write) an area the unused sector is erased and the content of the last used sector is copied to the other, omitting the area for the new data. Finally this data is written to the omitted "gap".

This application note gives an example how to implement this.

1.2 Flash Memory Structure of MB90F897

The flash memory in the MB90F897 is divided into two banks, which are divided into sectors as follows:

	Flash memory	CPU address
Upper bank	SA9 (4 KB)	FFFFFF$_H$ FFF000$_H$
	SA8 (4 KB)	FFEFFF$_H$ FFE000$_H$
	SA7 (4 KB)	FFDFFF$_H$ FFD000$_H$
	SA6 (4 KB)	FFCFFF$_H$ FFC000$_H$
	SA5 (16 KB)	FFBFFF$_H$ FF8000$_H$
	SA4 (16 KB)	FF7FFF$_H$ FF4000$_H$
Lower bank	SA3 (4 KB)	FF3FFF$_H$ FF3000$_H$
	SA2 (4 KB)	FF2FFF$_H$ FF2000$_H$
	SA1 (4 KB)	FF1FFF$_H$ FF1000$_H$
	SA0 (4 KB)	FF0FFF$_H$ FF0000$_H$

1.3 Used Algorithm

For the EEPROM emulation the sectors SA0 and SA1 of the lower Flash bank of the MB90F897 is used, while the code is executed in the upper Flash bank.

1.3.1 Initializing

For the very first operation both sectors SA0, SA1 have to be erased. Later this initialization must not be performed, because it would mean losing all stored "EEPROM data".

SA0 is then selected as the valid sector by writing `0x0001` to the uppermost cell (`0xFF0FFE`).

1.3.2 Writing data

For writing new data to the "EEPROM", the content of the valid sector is copied to the other, except the area, which will be used for this new data. After this the new data is written to the gap and the non-valid sector becomes the valid one.

Note that the data is written word-wise.

The functionality is shown in the illustration below:

Flow of the EEPROM Emulation

1. Erase

Valid Sector

2. Copy   2. Copy

New data

3. Write

Non-valid Sector

4. Make valid
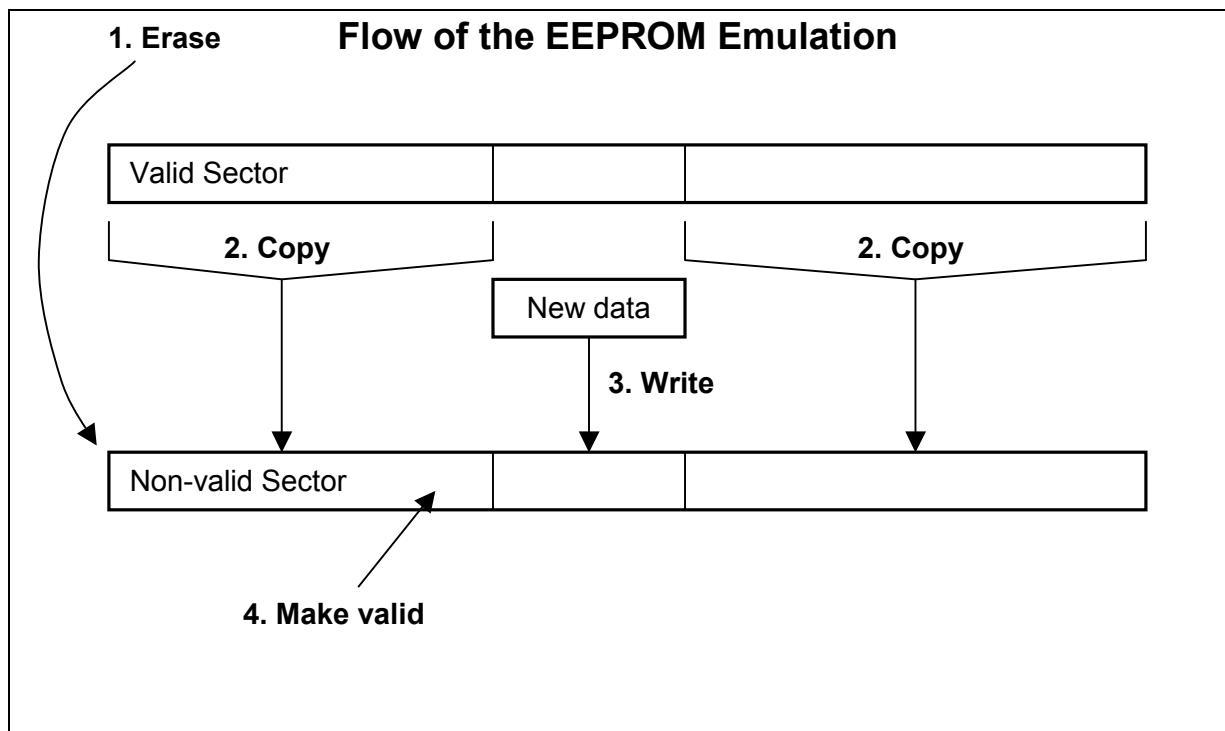
2 Programming Example

EXAMPLE CODE OF HOW TO PROGRAMM A EEROM EMULATION

2.1 Definitions

The following definitions should be done to make the code more readable:

```
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                             */
/*                  (C) Fujitsu Microelectronics Europe GmbH                 */

 /*---------------------------------------------------------------------
   MAIN.C
   - description
   - EEPROM Emulation Demonstration with Dual Flash

 /*-------------------------------------------------------------------*/

// sequence addresses for SA1
#define seq_1AAA ((__far volatile unsigned int*)0xFF1AAA)
#define seq_1554 ((__far volatile unsigned int*)0xFF1554)

// sequence addresses for SA2
#define seq_2AAA ((__far volatile unsigned int*)0xFF2AAA)
#define seq_2554 ((__far volatile unsigned int*)0xFF2554)

// sector SA1 start address
#define SA1       ((__far unsigned int*)0xFF1000)
#define SA1W      0xFF1000
// sector SA2 start address
#define SA2       ((__far unsigned int*)0xFF2000)
#define SA2W      0xFF2000

// valid sector flag (SA1)
#define act1      ((__far unsigned int*)0xFF1FFE)
// valid sector flag (SA2)
#define act2      ((__far unsigned int*)0xFF2FFE)

#define DQ7       0x0080   // data polling flag
#define DQ5       0x0020   // time limit exceeding flag
```

2.2 Low Level Drivers

The following code gives an example of the basic Flash Memory operations:

```
/----------------- low level driver ---------------------------

int sec_erase(__far unsigned int *adr)  // Erases sector SAn
{
      unsigned char flag;

      flag = 0;

      FMCS_WE = 1;                       // programming enable         ▼
```

```
        FWR0 = 0x03FF;                      // write enable (all sectors)        ▲

        if (adr < SA2)                      // which sector?
        {
                *seq_1AAA = 0xAA;           // send erase command to SA1
                *seq_1554 = 0x55;
                *seq_1AAA = 0x80;
                *seq_1AAA = 0xAA;
                *seq_1554 = 0x55;
        }
        else
        {
                *seq_2AAA = 0xAA;           // send erase command to SA2
                *seq_2554 = 0x55;
                *seq_2AAA = 0x80;
                *seq_2AAA = 0xAA;
                *seq_2554 = 0x55;
        }
        *adr = 0x0030;                      // erase!

        while(flag == 0)
        {
                if((*adr & DQ7) == DQ7)     // Toggle bit
                {
                        flag = 1;           // successful erased
                }
                if((*adr & DQ5) == DQ5)     // time out
                        if((*adr & DQ7) == DQ7)
                        {
                                flag = 1;   // successful erased
                        }
                        else
                        {
                                flag = 2;   // timeout error
                        }
        }
        FMCS_WE = 0;

        return flag;
}


int write(__far unsigned int *adr,
          unsigned int data)       // Writes a word to sector SAn address
{
        unsigned char flag;

        flag = 0;

        FMCS_WE = 1;                        // programming enable
        FWR0 = 0x03FF;                      // write enable (all sectors)

        if (adr < SA2)                      // which sector?
        {
                *seq_1AAA = 0xAA;           // sends write command to SA1
                *seq_1554 = 0x55;
                *seq_1AAA = 0xA0;
        }
        else
        {
                *seq_2AAA = 0xAA;           // sends write command to SA2
                *seq_2554 = 0x55;
                *seq_2AAA = 0xA0;
        }
        *adr     = data;                    // send data to the pointed address   ▼
```

```
        while(flag == 0)
        {
                if((*adr & DQ7) == (data & DQ7))  // Toggle bit
                {
                        flag = 1;               // successful programmed
                }
                if((*adr & DQ5) == DQ5)     // time out
                        if((*adr & DQ7) == (data & DQ7))
                        {
                                flag = 1;   // successful programmed
                        }
                        else
                        {
                                flag = 2;   // timeout error
                        }
        }
        FMCS_WE = 0;                             // reset programming enable flag

        return(flag);
}
```

2.3 EEPROM Routines

The next code boxes contain the code for the usage of the Flash Memory as an EEPROM as described in the last chapter.

```
//----------------- "EEPROM" functions --------------------

unsigned char init_eeprom (void)
{
   unsigned char err = 0;

   if (sec_erase (SA1) == 2) err = 1;   // erase SA1
   if (sec_erase (SA2) == 2) err = 2;   // erase SA2
   if (write (act1, 1) == 2) err = 3;   // mark SA1 as valid sector

   return err;
}

unsigned char sec_copy (unsigned int start_adr, unsigned int end_adr)
{
   unsigned char err = 0;
   unsigned int addr, dummy;
   unsigned int __far *address;

   if (*act1 == 1)                                  // SA1 valid?
   {
      if (sec_erase(SA2) == 2) err=1;          // erase SA2

         if (err == 0)               // copy complete sector except
         {                               // [start_adr...end_adr]  and "actual flag"
            for (addr = 0; addr < 0x0FFE; addr = addr + 2)
            {
               if (((addr < start_adr) | (addr > end_adr)) & (err == 0))
               {
                  address = (unsigned int __far*)(SA1W + addr);
                  dummy = *address;
                  if (write ((unsigned int __far*)(SA2W + addr), dummy) == 2)
                    err = 1;
               }
```

```
        }                                                         ▲

        if (write (act2, 1) == 2) err = 1;    // make SA2 valid
        if (write (act1, 0) == 2) err = 1;    // SA1 not valid anymore
    }
}
else if (*act2 == 1)                          // SA2 valid?
{
    if (sec_erase(SA1) == 2) err=1;           // erase SA1

    if (err == 0)                    // copy complete sector except
    {                                // [start_adr...end_adr] and "actual flag"
        for (addr = 0; addr < 0x0FFE; addr = addr + 2)
        {
            if (((addr < start_adr) | (addr > end_adr)) & (err == 0))
            {
                address = (unsigned int __far*)(SA2W + addr);
                dummy = *address;
                if (write ((unsigned int __far*)(SA1W + addr), dummy) == 2)
                    err = 1;
            }
        }

        if (write (act1, 1) == 2) err = 1;    // make SA1 valid
        if (write (act2, 0) == 2) err = 1;    // SA2 not valid anymore
    }
}
else err = 1;                                 // error

    return err;
}

unsigned char write_eeprom(unsigned int start_adr, unsigned int end_adr)
{
    unsigned char err, i;
    unsigned int adr;

    err = sec_copy(start_adr, end_adr);       // Copy data from actual sector

    if (err != 1)
    {
        adr = start_adr;
        i = 0;

        if (*act1 == 1)                    // Write new data to new valid sector
        {
            while ((err == 0) & (adr < end_adr))
            {
                if (write((unsigned int __far*)(SA1W + adr), datafield[i]) == 2)
                    err = 1;
                adr = adr + 2;
                i++;
            }
        }
        else if (*act2 == 1)               // Write new data to new valid sector
        {
            while ((err == 0) & (adr < end_adr))
            {
                if (write((unsigned int __far*)(SA2W + adr), datafield[i]) == 2)
                    err = 1;
                adr = adr + 2;
                i++;
            }
        }
        else
        {                                                         ▼
```

```
         err = 1;                                                              ▲
      }
   }

   return err;
}

unsigned int read_eeprom(unsigned int adr)     // Read data from valid sector
{
   unsigned int rd;
   unsigned int __far *address;

   if (*act1 == 1)
   {
      address = (unsigned int __far*)(SA1W + adr);
      rd = *address;
   }
   else
   {
      address = (unsigned int __far*)(SA2W + adr);
      rd = *address;
    }

   return rd;
}
```

2.3.1 C Functions

- `init_eeprom()` Initializes and erases both EEPROM-sectors.

- `sec_copy(startadr, endadr)` Copies complete sector from actual to non-valid, remaining a gap between *startadr* and *endadr*. Marks non-valid sector to actual.

- `write_eeprom(startadr, endadr)` Calls `sec_copy`. Writes data from *startadr* to *endadr* to EEPROM.

- `read_eeprom(adr)` Reads a single word from *adr* of actual sector.

Note, if you want to write `n` words to the EEPROM beginning at `startaddress`, please use the following formula for the `endaddress`:

$$endaddress = startaddress + 2 \cdot n - 1$$

Important:

`startaddress` must be an even value number.

2.4 Using Driver and EEPROM routines in a program

The following demonstration C code shows how to use the routines above. Note, that the function `error()` stands for an error handler, which is not shown here.

```c
#include "mb90385.h"                        // for mb90f897

unsigned int datafield[16];                 // data field to write in "EEPROM"

void main(void)
{
      unsigned char i;
      unsigned int a;

      if (init_eeprom() != 0) error();      // Init EEPROM

      for (i = 0; i < 16; i++)              // Fill data field
      {
            datafield[i] = i * 99;
      }

      if (write_eeprom(0, 31) == 1) error(); // write all 16 words

      for (i = 0; i < 16; i++)              // read back data for checking
      {
            a = read_eeprom(i * 2);
            if ( a != (i * 99))
            {
                  error();
            }
      }

      for (i = 0; i < 8; i++)               // new data (8 Words)
      {
            datafield[i] = i * 77;
      }

      if (write_eeprom(4, 19) == 1) error(); // partly overwrite old data
                                            //   (inserting new data)

      // check all data
      if (read_eeprom(0)  !=    0) error();  // old
      if (read_eeprom(2)  !=   99) error();  // old
      if (read_eeprom(4)  !=    0) error();  // new, old: 198
      if (read_eeprom(6)  !=   77) error();  // new, old: 297
      if (read_eeprom(8)  !=  154) error();  // new, old: 396
      if (read_eeprom(10) !=  231) error();  // new, old: 495
      if (read_eeprom(12) !=  308) error();  // new, old: 594
      if (read_eeprom(14) !=  385) error();  // new, old: 693
      if (read_eeprom(16) !=  462) error();  // new, old: 792
      if (read_eeprom(18) !=  539) error();  // new, old: 891
      if (read_eeprom(20) !=  990) error();  // old
      if (read_eeprom(22) != 1089) error();  // old
      if (read_eeprom(24) != 1188) error();  // old
      if (read_eeprom(26) != 1287) error();  // old
      if (read_eeprom(28) != 1386) error();  // old
      if (read_eeprom(30) != 1485) error();  // old
```

This demonstration program first initializes the EEPROM (erasing all two sectors). This step should only be done at the very first execution. You may write data to a certain address, which determines that the EEPROM contains data. After a power-on the content of this address can be checked and thus the initializing can be skipped.

After initializing a field of 16 words is generated and then written to the EEPROM beginning at address "0". Note, that the address has to be stated as byte, so that the 16 words reach from `0x0000` to `0x001F` (31_d). These data words are checked by reading them back afterwards.

Following the addresses `0x0004` to `0x0012` (18_d) are overwritten by 8 new data words.

Finally all data words from `0x0000` to `0x001F` (31_d) are checked by reading them back.

2.5 How to use the EEPROM functions in an own application (Function Summary)

The RAM buffer for the EEPROM data should be declared global, so that no pointers has to be used as arguments in the function calls. This eases the handling. Use the following declaration

`unsigned int datafield[MAXDATA];`

With `MAXDATA` the buffer size is determined.

The initialization at the very first program execution (or erasing the EEPROM) is done by the function:

`unsigned char init_eeprom (void)`
"0" is returned if the initialization was successful.

To write to the EEPROM the `datafield` has to be filled with the desired data and the function

`unsigned char write_eeprom(unsigned int start_adr, unsigned int end_adr)`

writes the desired data to `start_adr` till `end_adr`. Please note that always beginning from `datafield[0]` the data is written.

Also note, if you want to write `n` words to the EEPROM beginning at `start_adr`, please use the following formula for the `end_adr`:

$$\text{end\_adr} = \text{start\_adr} + 2 \bullet n - 1$$

Important: `start_adr` must be an even value number.

Reading from the EEPROM realizes the function:

`unsigned int read_eeprom(unsigned int adr)`

3   Time effort

THIS CHAPTER GIVES AN OVERVIEW OF THE TIME EFFORT

3.1   Background

Because for each new data a whole sector has to be erased and copied, the time effort is higher than using a standard EEPROM.

The execution time can be decreased by using not the whole sector. The erase time will not be affected, but the copy time.

3.2   Timing

A 4K sector erase takes about 115 ms and a sector copy 85 ms. This means a copy transfer rate of 85 ms / 2048 words = 41.5 µs/word at 16 MHz CPU speed.

So for each data writing a time effort of about 115 ms + 85 ms = 200 ms is taken if a whole sector is used.

4 Other Usage

THIS CHAPTER GIVES AN OVERVIEW OF OTHER EEPROM CONCEPTS

4.1 O-Buffer (Ring-Buffer) Concept 1

With some modifications a circular buffer (O-Buffer or Ring-Buffer) can be realized. The following graphic illustrates such kind of buffer:
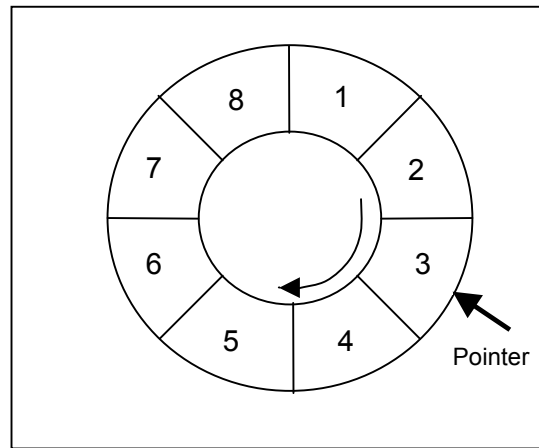

O-Buffer

In this example the buffer is divided into 8 sectors. Each buffer sector has the same size. A pointer points to the last written sector and is increased by 1 if the sector is filled with new data. When the pointer points to 8 and data is written the pointer moves on to 1 and this sector is overwritten in the next write process.

One concept is that the EEPROM functions have to be modified a bit so that the valid flag now is the pointer and a fix data length has to be used. Additionally a small pointer update routine has to be written and added to the EEPROM functions. The value of the pointer can be used to calculate the actual address.
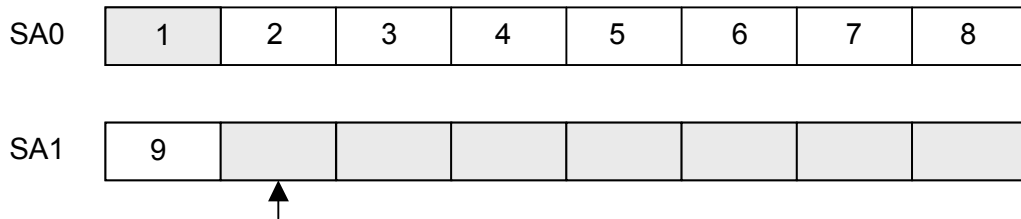
4.2 O-Buffer Concept 2

The O-Buffer concept 1 has the disadvantage that for each buffer update a whole Flash Sector is erased. This takes time and slows the performance of the MCU. A better concept is the fill one Sector with data. If the end of the Sector is reached, another Sector is filled. From this time on a Sector Erase is only needed if the end of a Sector is reached. The following graphics illustrate this concept (The arrow means the pointer):
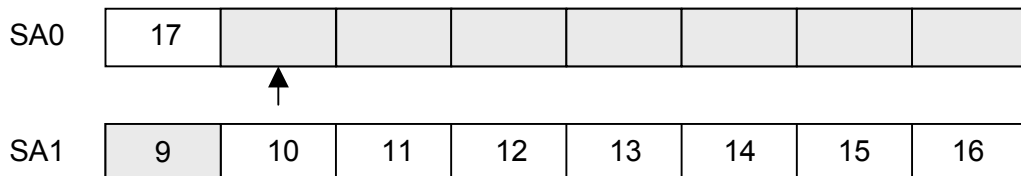

1. Both Sector empty

SA0

SA1
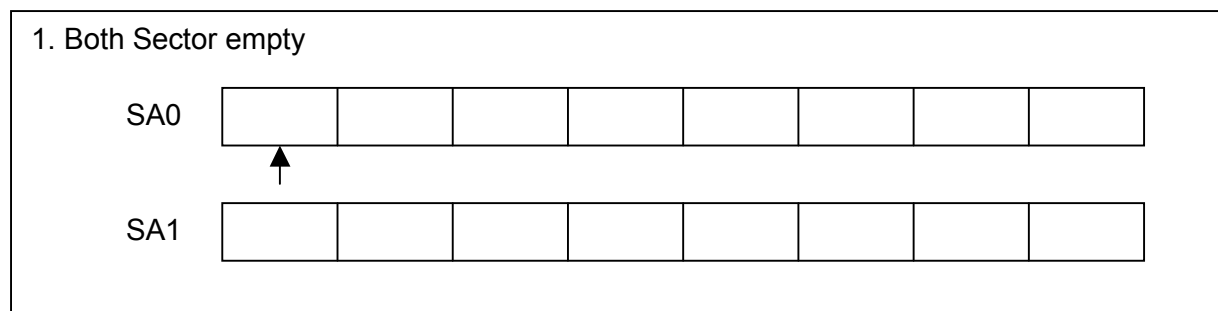
2. Sector SA0 is filled with data, end of Sector is reached.

| SA0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| SA1 | | | | | | | | |

---
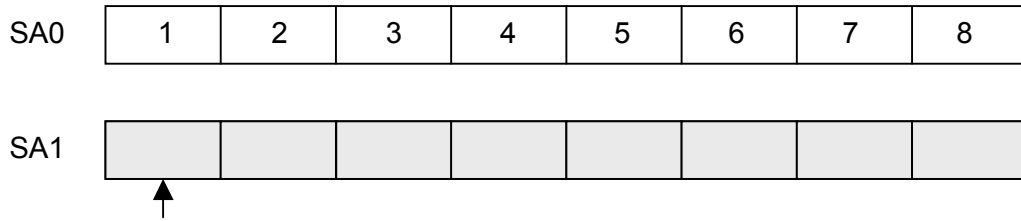
3. Data-9 writing continues with SA1.

| SA0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| SA1 | 9 | | | | | | | |

---

4. SA1 is full and data-17 writing is pending → SA0 is erased and data is written.

| SA0 | 17 | | | | | | | |

| SA1 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

---

3. SA0 is full and data-25 writing is pending → SA1 is erased and data is written..

| SA0 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |

| SA1 | 25 | | | | | | | |
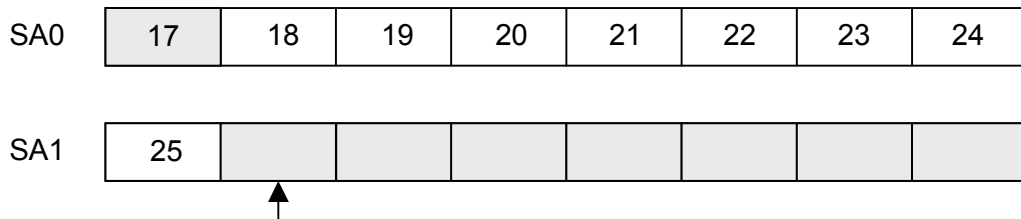
… and so on.

Note, that the read_eeprom function must not return the invalid data sections (gray boxes). So the pointer determines from which sector is read.

In this example a time-consuming sector erase is performed after all 8 writings to the ring buffer (without the "initial phase").