Application Note

EEPROM Emulation Driver Guide for F29H85x

Alex Wasinger

ABSTRACT

Many applications require storing small quantities of system related data (calibration values, device configuration) in non-volatile memory so that it can be used or modified and reused even after power cycling the system. Electrically erasable programmable read-only memory, or EEPROM, is primarily used for this purpose. EEPROMs have the ability to erase and write individual bytes of memory many times over and the programmed locations retain the data over a long period even when the system is powered down. This application note and the associated code help to define a sector(s) of on-chip Flash memory as the emulated EEPROM and is transparently used by the application program for writing, reading, and modifying the data.

Project collateral and source code discussed in this application note can be found in F29H85x SDK 1.00.00.00 (or higher) at the following path: C:\ti\f29h85x-sdk_1_00_00_00\examples\driverlib\single_core\flash.

Table of Contents

List of Figures

Trademarks

Code Composer Studio™ is a trademark of Texas Instruments.
All trademarks are the property of their respective owners.

1 Introduction

F29H85x MCUs come with different configurations of Flash memory that is arranged in multiple sectors. Unfortunately, the technology used for the on-chip Flash memory does not allow for the addition of a traditional EEPROM to the chip. This functionality can be emulated, however, on C2000 MCUs, which have in-circuit programming ability for the flash memory. This application report demonstrates this functionality using sectors of the on-chip flash memory. Note that at least one whole flash sector is used for emulation, and as such is not available for storing application code.

2 Differences Between EEPROM and On-Chip Flash

EEPROMs are available in different capacities and connect with the host microcontrollers via a serial and sometimes parallel interface. The serial inter-integrated circuit (I2C) and serial peripheral interface (SPI) are quite popular due to the minimal number of pins/traces. EEPROMs can be programmed and erased electrically, and most of the serial EEPROMs allow byte-by-byte program or erase operations.

The biggest difference between EEPROM and Flash is in the erase operation: an EEPROM can erase any particular individual byte, while Flash must clear at least one entire sector.

Flash write and erase cycles are performed by applying time-controlled voltages to each cell. In the erase condition, each cell (bit) reads a logical 1. Therefore, every Flash location of a C2000 Real-Time Controller reads 0xFFFF when erased. By programming, the cell can be changed to logical 0. Any byte can be overwritten to change a bit from logical 1 to 0 (assuming the corresponding ECC has not been programmed); but not the other way around. The on-chip Flash memory on F29H85x MCUs parts require TI-supplied specific algorithms (Flash API) for erase and write operations.

Note

For the Flash erase/program/read times, see the Flash Parameters section in *Electrical Characteristics* section of the device-specific data manual.

3 Overview

This document describes two implementations of EEPROM emulation for F29H85x devices: Single-Unit and Ping-Pong. They each support two programming modes: Page-Programming Mode and 64-bit Programming Mode. The Single-Unit implementation will be described first, followed by Ping-Pong.

There are multiple user-configurable EEPROM variables supported by each implementation. These variables are detailed in Section 6.1.

3.1 Basic Concepts

In this implementation, the emulated EEPROM is comprised of at least one Flash Sector. Due to the block erase requirement of Flash, an entire Flash sector must be reserved for the EEPROM emulation. The size of a flash sector varies based on the C2000 part number.

Each EEPROM project has two programming modes: 64-bit mode and Page mode:

- **64-bit mode:** no pages or page tracking overhead, maximum EEPROM size of 64 bits
- **Page mode:** subdivides the chosen flash sector(s) into "pages" that store iterations of the EEPROM as it is updated
  - For example, a 2K x 16 flash sector can be divided into 32 pages, each with a size of 64 x 16.

The two implementations (Single-Unit and Ping-Pong) differ in how they handle the erase process when the designated Flash memory is full. These processes are described in the following sections.

3.2 Single-Unit Method

If using Single-Unit EEPROM Emulation, if the EEPROM unit is full and there is more data to be written, the EEPROM unit is erased and then new data is programmed to Flash. This behavior is depicted in the following figure:

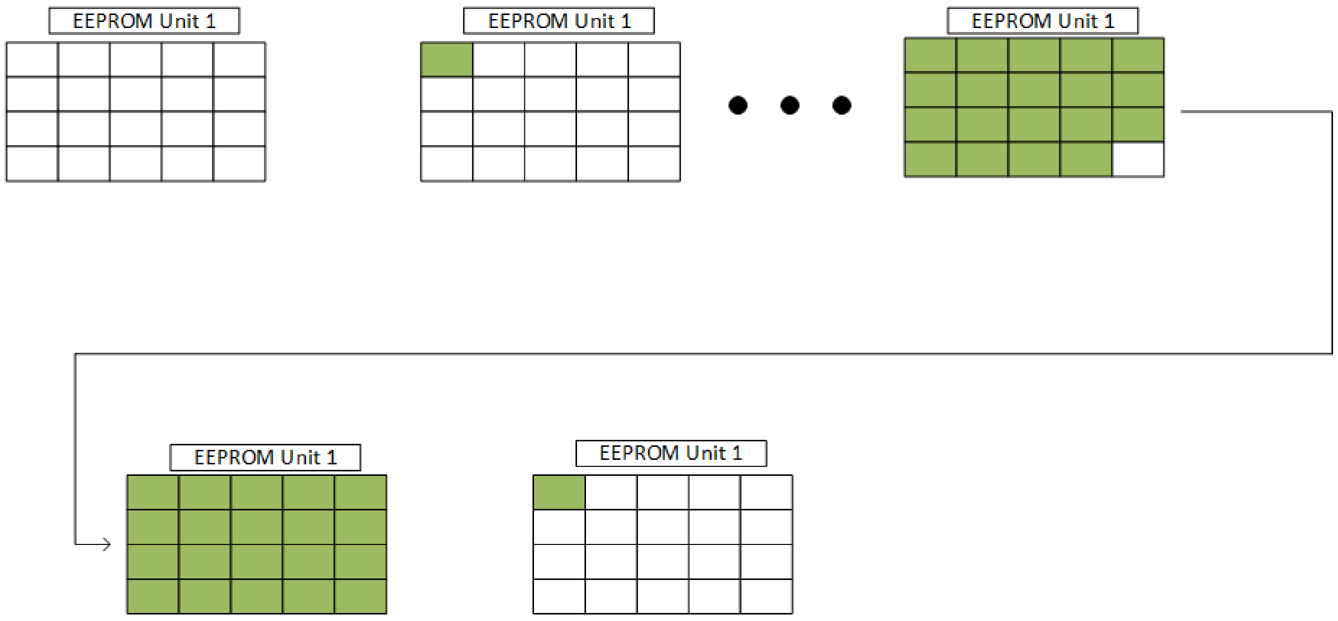


Figure 3-1. Single-Unit Behavior

3.3 Ping-Pong Method

If using Ping-Pong emulation, a second flash memory range is specified, and the emulation is divided into the **Active Unit** and the **Inactive Unit**. When the Active Unit is full and the EEPROM is updated, the unit designations will swap and the data is written to the new Active Unit. After the data is programmed, the Inactive Unit is safely erased without fear of data loss.

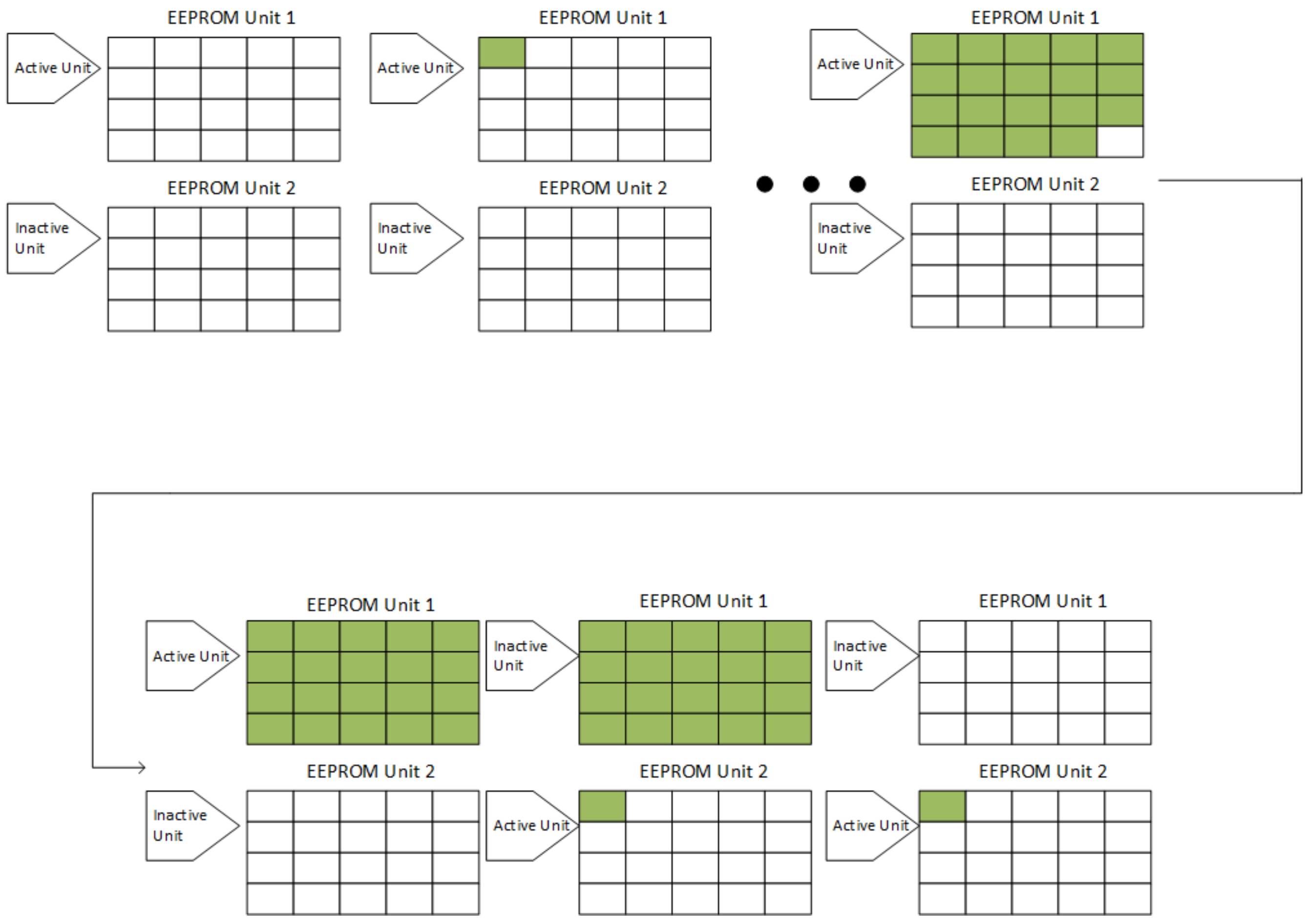


Figure 3-2. Ping Pong Behavior

3.4 Creating EEPROM Sections (Pages) and Page Identification

To support EEPROM emulation with varying data sizes (greater than 64 bits), the Flash sectors selected for emulation are divided into a format referred to as EEPROM Banks (not to be confused with Flash Banks) and Pages. This aspect of the emulation is the same for both the Single-Unit and Ping-Pong implementations.

The flash sector(s) chosen for emulation are subdivided into EEPROM banks, which are in turn divided into Pages. This is depicted in Figure 3-3.

Using this format allows the application to:

- Read back the data from the page written during the previous save
- Write the latest data to a new page
- Read from any previously stored data, if required by the application

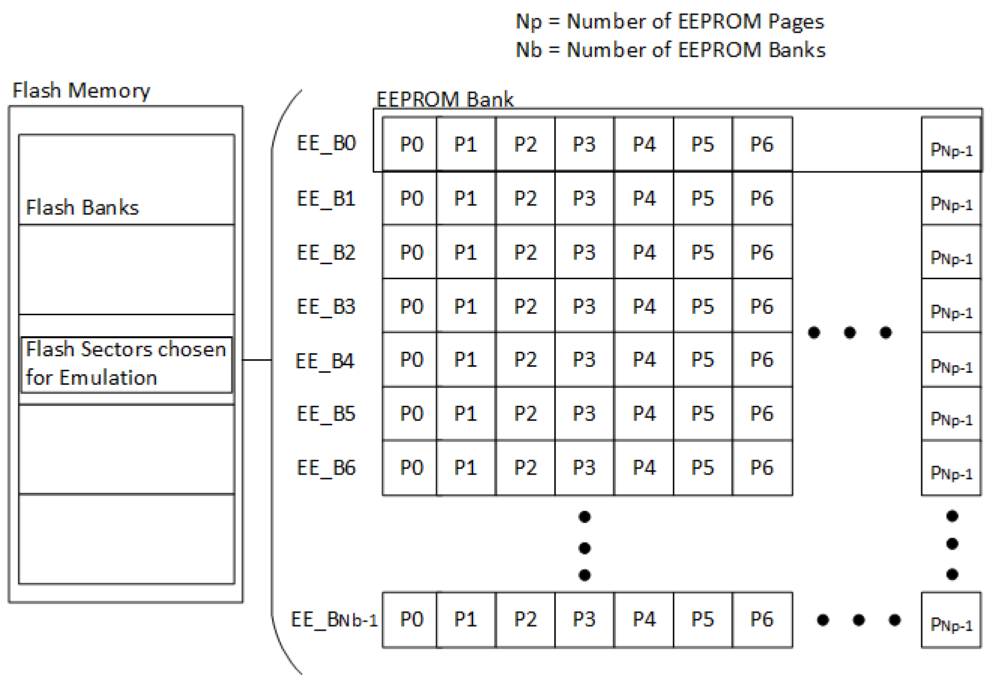


Figure 3-3. Bank Partitioning

The application tracks which bank and page is in use through status codes. For more information, seeSection 5.2.2.1.

To track which EEPROM banks are empty, current, or full the first 16 bytes (128 bits) of each bank are reserved. On switching to a new bank, the status of both the previous and new EEPROM banks is updated to reflect the new state.

Pages are handled in a similar manner: the first 16 bytes (128 bits) of each page is reserved to determine if it is empty, current, or used. Whenever new data is written to a Page, the status of both the previous and new Pages is updated.

To mark an EEPROM Bank or Page as current, the first 64 bits are written with the appropriate status code. To mark an EEPROM Bank or Page as full, the second 64 bits are written with the appropriate status code.

As seen in Page Layout, all pages contain an eight-word page status and a configurable amount of data space. Page 0 is slightly different as it contains the EEPROM bank status as well. Although only Pages 0 and 1 are shown, it should be noted that Page 2 through Page (N-1) are identical to Page 1.

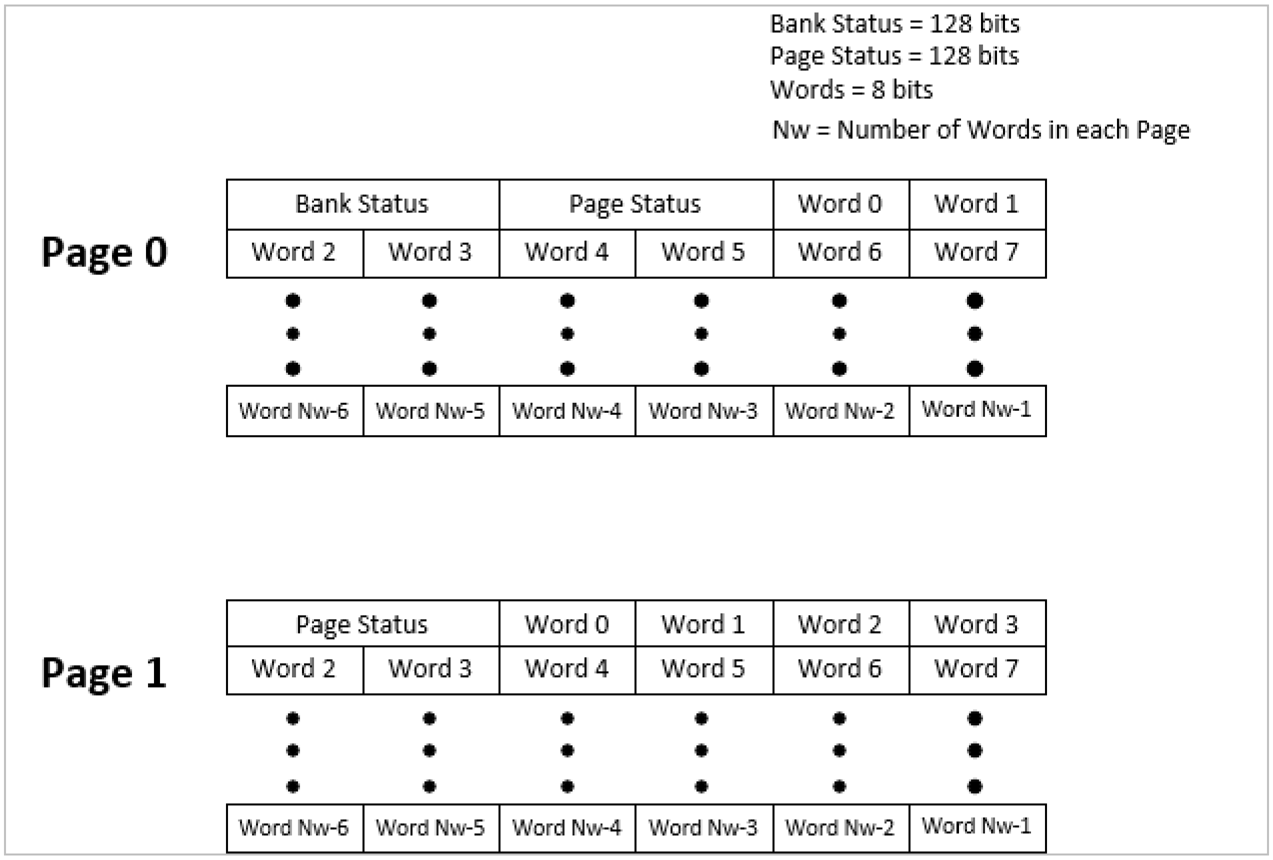


Figure 3-4. Page Layout

4 Software Description

The software provided with this application report includes EEPROM Emulation source code for the F29H85 Real-Time Controller with an example project demonstrating how to utilize the source code.

This software provides basic EEPROM functionality: write, read, and erase. At least one sector of Flash memory is used to emulate EEPROM. This sector(s) is broken into several EEPROM banks and pages, each containing status words to determine the validity of the data as described above.

This code uses the Header Files and Flash API libraries provided for the F29H85x. The example code can be found within the F29H85x SDK Directory. The full path is: f29h85x-sdk_1_00_00_00\examples\driverlib\single_core\flash

4.1 Software Functionality and Flow

The device must first go through its initialization code to initialize clocks, peripherals, and so forth. The initialization functions used are provided with the header library files included in the project. Further information regarding this sequence can be read in the documentation provided with the header files.

Once this is complete, the Flash API initialization and parameters are set to prepare for Flash programming. The Flash API library requires a few files and certain initialization/setup to function properly. The complete list of required steps can be found in the *F29H85x Flash API Reference Guide*.

Next, the EEPROM Configuration specified by the user in EEPROM_Config.h will be checked for validity and certain variables used by the Flash API will be configured. More details can be found in User-Configuration and Section 6.2.1.2.

At this point, programming can begin. The user data is written to flash and is subsequently read back. This software flow, particularly the initialization portion, should be followed by most applications, as some Flash API functions need to be copied to internal RAM before programming can begin.

The example project provided follows this software flow shown in Software Flow. To learn more about the functions shown in the diagram, navigate to the appropriate section in the document.

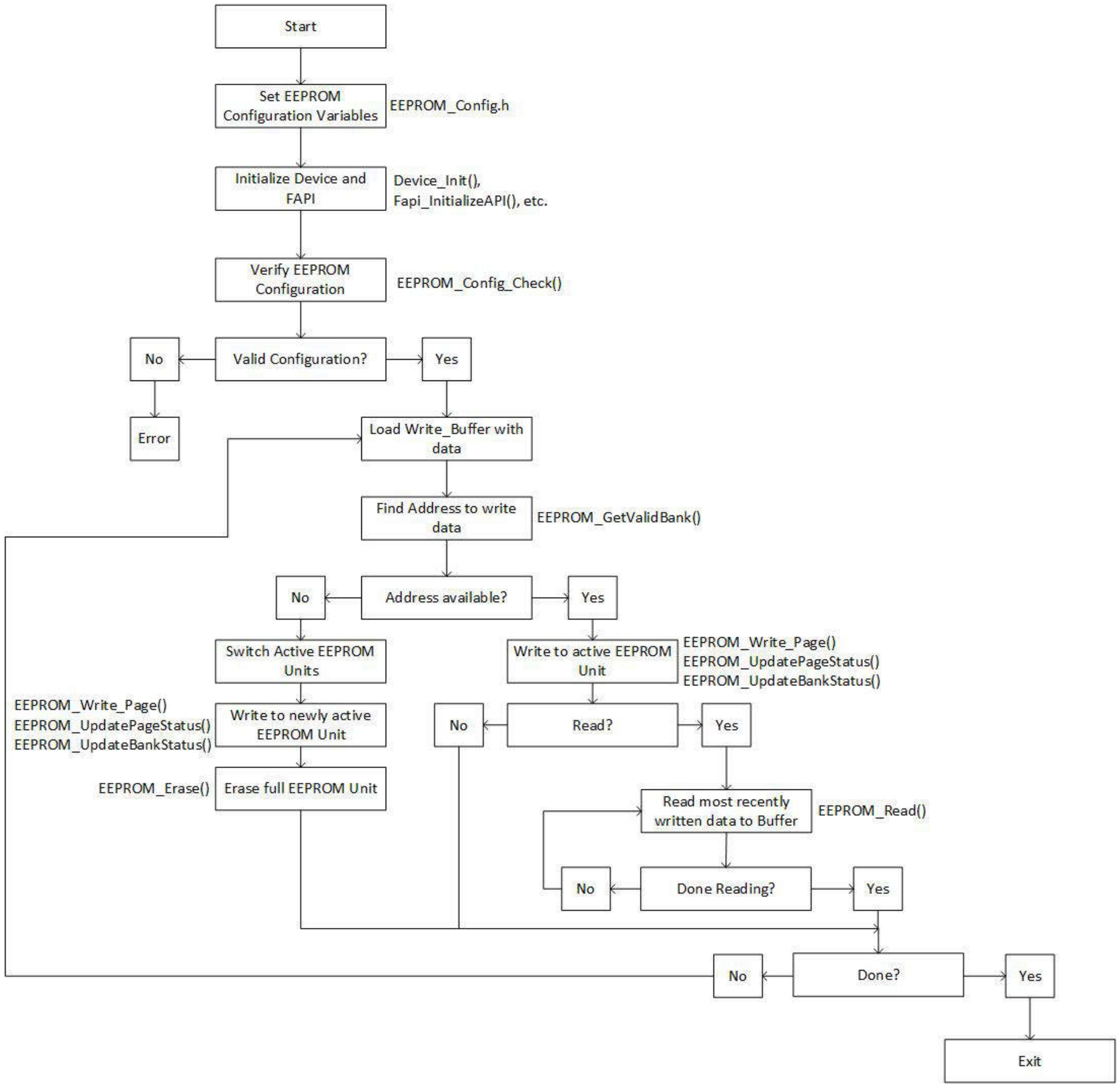

Start
Set EEPROM Configuration Variables
EEPROM_Config.h
Initialize Device and FAPI
Device_Init(), Fapi_InitializeAPI(), etc.
Verify EEPROM Configuration
EEPROM_Config_Check()
Valid Configuration?
No
Error
Yes
Load Write_Buffer with data
Find Address to write data
EEPROM_GetValidBank()
Address available?
No
Switch Active EEPROM Units
Write to newly active EEPROM Unit
EEPROM_Write_Page()
EEPROM_UpdatePageStatus()
EEPROM_UpdateBankStatus()
Erase full EEPROM Unit
EEPROM_Erase()
Yes
Write to active EEPROM Unit
EEPROM_Write_Page()
EEPROM_UpdatePageStatus()
EEPROM_UpdateBankStatus()
Read?
No
Yes
Read most recently written data to Buffer
EEPROM_Read()
Done Reading?
No
Yes
Done?
No
Yes
Exit

Figure 4-1. Software Flow

5 Single-Unit Emulation

Single-Unit EEPROM emulation uses only one set of Flash sectors, and thus has no inactive EEPROM unit. It's behavior is documented below, with the primary differences from Ping-Pong located in the EEPROM_Erase() function.

5.1 User Configuration

The implementation detailed in this document allows you to configure several variables for EEPROM Emulation. These variables are found within EEPROM_Config.h and F29H85x_EEPROM.c.

5.1.1 EEPROM_Config.h

This header file contains definitions that allow the user to change various aspects of EEPROM configuration. These aspects include:

- Choose between Page Mode and 64-Bit Mode, these are mutually exclusive

```
//#define _64_BIT_MODE 1
#define PAGE_MODE 1
```

- Choose how many EEPROM Banks to emulate.

```
#define NUM_EEPROM_BANKS 8
```

- Choose how many EEPROM Pages within each EEPROM Bank

```
#define NUM_EEPROM_PAGES 3
```

- Choose the size of the data space contained within each EEPROM Page (unit is bytes). Although any size can be specified, the size will be adjusted to the closest multiple of eight that is greater than or equal to the number given. For example, a specified size of six bytes per page will be programmed as eight bytes per page, with the last two being treated as 0xFFFF. This is to comply with Flash requirements (8 bits of ECC is programmed for every 64-bit aligned Flash memory address).

```
#define DATA_SIZE 64
```

5.1.2 F29H85x_EEPROM.c

Choose which Flash Sectors to use for EEPROM emulation. The sectors chosen (if multiple) should be contiguous and in order from least to greatest. Insert only the First and Last sectors to be used for EEPROM. For example, to use sectors 1-10, insert {1,10}. To use only sector 1, insert {1,1}.

```
uint32_t FIRST_AND_LAST_SECTOR[2] = {1,1};
```

A valid configuration has the following properties:

- Includes only sectors that exist on the device
- Does not create an overlap in the Write/Erase Protection Masks between the two units
  - The F29H85x Flash API requires Write/Erase Protection Masks to be configured before programming Flash Memory. Details about the proper configuration of these masks can be found in the *F29H85x Flash API Reference Guide* (SPRUJE7).

More details about invalid or dangerous configurations can be found in Section 5.2.1.2.

Users can additionally enable blank check after erase:

```
uint8_t Erase_Blank_Check = 1;
```

5.2 EEPROM Functions

Listed below are functions required for the implementations. They are found in the F29H85x_EEPROM.c or F29H85x_EEPROM_Example.c files.

Initialization + Setup

- Configure_Device()
- EEPROM_Config_Check()

Page Mode

- EEPROM_GetValidBank(uint8_t ReadFlag)
- EEPROM_UpdateBankStatus()
- EEPROM_UpdatePageStatus()
- EEPROM_UpdatePageData(uint8_t* Write_Buffer)
- EEPROM_Write_Page(uint8_t* Write_Buffer)

64-Bit Mode

- EEPROM_64_Bit_Mode_Check_EOS()
- EEPROM_Write_64_Bits(uint8_t Num_Bytes, uint8_t* Write_Buffer)

Both

- EEPROM_Erase()
- EEPROM_Read(uint8_t* Read_Buffer)

Utility

- EEPROM_Write_Buffer(uint8_t* address, uint8_t* write_buffer)
- Erase_Bank()
- Set_Protection_Masks()
- Configure_Protection_Masks(uint32_t* Sector_Numbers, uint32_t Num_EEPROM_Sectors)
- Fill_Buffer(uint8_t* status_buffer, int buffer_len, uint8_t value)
- ClearFSMStatus()

Each of these functions is discussed in detail in the subsequent sections.

5.2.1 Initialization and Setup Functions

5.2.1.1 Configure_Device

This function encapsulates all standard device and FlashAPI setup.

First, it initializes the device and its peripherals.

```
Device_init();
Flash_initModule(3);

Device_initGPIO();

Interrupt_initModule();

Interrupt_initVectorTable();

__asm(" ENINT")
```

Then, it requests the flash semaphore and initializes the Flash API with the chosen user configuration.

```
HWREG(SSUGEN_BASE + SSU_O_FLSEMREQ ) =  1;
while ((HWREG( SSUGEN_BASE + SSU_O_FLSEMSTAT) & SSU_FLSEMSTAT_CPU_M)!= (0x1<<SSU_FLSEMSTAT_CPU_S));

u32UserFlashConfig = Fapi_getUserConfiguration(BankType, FOTAStatus);

Fapi_SetFlashCPUConfiguration(u32UserFlashConfig);

oReturnCheck = Fapi_initializeAPI((Fapi_FmcRegistersType*) FLASHCONTROLLER1_BASE, 200);

if(oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}
```

5.2.1.2 EEPROM_Config_Check

The EEPROM_Config_Check() function performs general error-checking and configures Write/Erase protection masks required by the Flash API. This function should be called before programming or reading from the emulated EEPROM Unit(s).

First, the function verifies that the Flash Bank selected for EEPROM Emulation is valid. On F29H85x, only the data flash is currently supported.

```
if (FLASH_BANK_SELECT != C29FlashBankFR4RP0StartAddress)
{
    return 0xFFFF;
}
```

Second, the validity of Flash Sectors selected for emulation is examined. This checks for:

- If more Flash sectors have been selected for emulation than exist on the device, and if at least one flash sector has been selected

```
uint32_t NUM_EEPROM_SECTORS_1 = FIRST_AND_LAST_SECTOR[1] - FIRST_AND_LAST_SECTOR[0] + 1;

NUM_EEPROM_SECTORS = NUM_EEPROM_SECTORS_1;

if (NUM_EEPROM_SECTORS > NUM_FLASH_SECTORS || NUM_EEPROM_SECTORS == 0)
{
    return 0xEEEE;
}
```

- Invalid combinations of the first and last Sectors selected for emulation

```
if (NUM_EEPROM_SECTORS > 1)
{
    if (FIRST_AND_LAST_SECTOR[1] <= FIRST_AND_LAST_SECTOR[0])
    {
        return 0xEEEE;
    }

    if (FIRST_AND_LAST_SECTOR[1] > NUM_FLASH_SECTORS - 1 || FIRST_AND_LAST_SECTOR[1] < 1)
    {
        return 0xEEEE;
    }
}
else if (FIRST_AND_LAST_SECTOR[0] > NUM_FLASH_SECTORS - 1 ||
         FIRST_AND_LAST_SECTOR[1] > NUM_FLASH_SECTORS - 1)
{
    return 0xEEEE;
}
```

If using Page Mode, the following will also be checked:

- If the total size of EEPROM Banks + Pages will fit in the Flash Sectors selected

```
Bank_Size = WRITE_SIZE_BYTES*2 + ((EEPROM_PAGE_DATA_SIZE + WRITE_SIZE_BYTES*2) * NUM_EEPROM_PAGES);

uint32 Available_Words = NUM_EEPROM_SECTORS * FLASH_SECTOR_SIZE;

if (Bank_Size * NUM_EEPROM_BANKS > Available_Words)
{
    return 0xCCCC;
}
```

It will also warn you with the appropriate code if one of the following conditions is detected:

- If space for one or more EEPROM Banks is left in Flash after configuring EEPROM Bank and Page size

```
if (Available_Words - (Bank_Size * NUM_EEPROM_BANKS ) >= Bank_Size)
{
    Warning_Flags += 1;
}
```

- If each page consists of less than or equal to 64 bits (8 bytes) (this wastes space as the 64-Bit Mode could be used for the same effect, without the overhead of status codes)

```
if (DATA_SIZE <= WRITE_SIZE_BYTES)
{
    Warning_Flags += 2;
}
```

- If using sectors in the 32-127 range and not using all eight sectors allocated to a single bit in the write protection mask, a warning is issued. Any unused sectors within group cannot be properly be protected from erase. For more information on write protection masks, see the *F29H85x Flash API Reference Guide*.

```
if (FIRST_AND_LAST_SECTOR[0] > 31 && FIRST_AND_LAST_SECTOR[1] > 31)
{

    if (NUM_EEPROM_SECTORS < 8)
    {
        Warning_Flags += 4;
    }


    else if ((FIRST_AND_LAST_SECTOR[0] % 8) != 0 || (FIRST_AND_LAST_SECTOR[1] + 1) % 8 != 0)
    {
        Warning_Flags += 4;
    }

}

else if (FIRST_AND_LAST_SECTOR[1] > 31 && (FIRST_AND_LAST_SECTOR[1] + 1) % 8 != 0)
{
    Warning_Flags += 4;
}
```

5.2.2 Page Mode Functions

5.2.2.1 EEPROM_GetValidBank

The EEPROM_GetValidBank() function provides functionality for finding the current EEPROM bank and page. This function is called by both the EEPROM_Write_Page() and EEPROM_Read() functions. Figure 5-1 shows the overall flow required to search for current EEPROM bank and page.

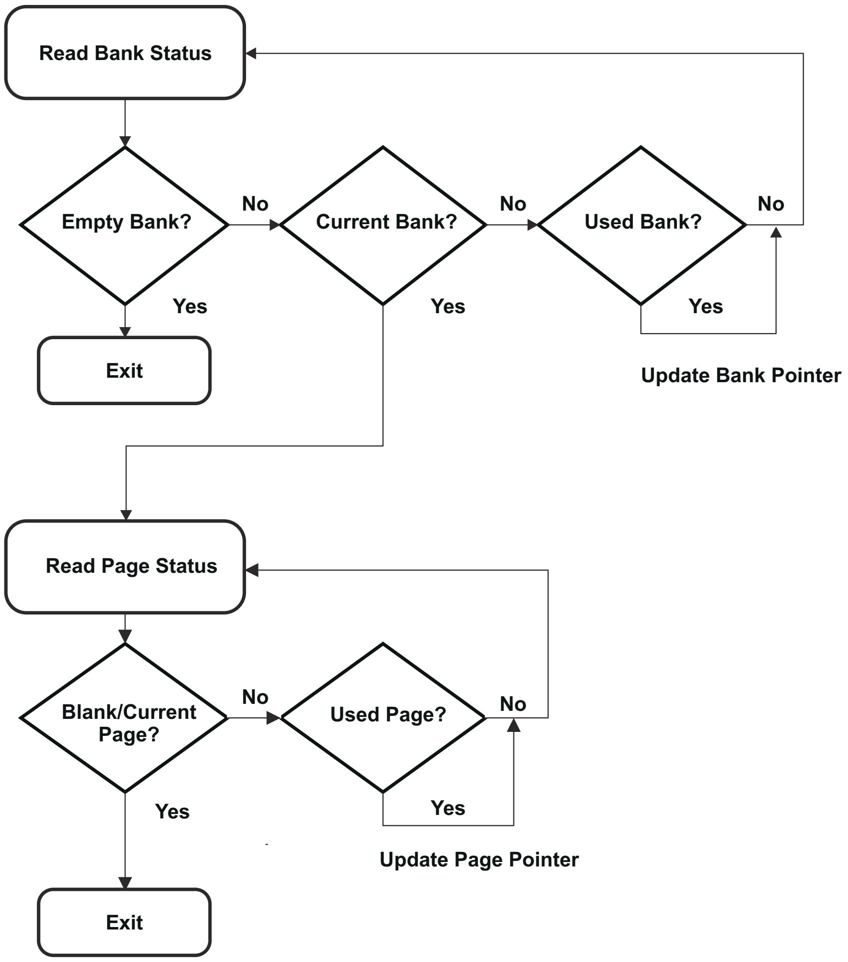


Figure 5-1. GetValidBank Flow

When entering this function, the EEPROM bank and page pointers are set to the beginning of the first sector specified in FIRST_AND_LAST_SECTOR:

```
RESET_BANK_POINTER;
RESET_PAGE_POINTER;
```

The addresses for these pointers are defined the EEPROM_Config.h file for the specific device and EEPROM configuration being used.

Next, the current EEPROM bank is found. As Figure 5-1 shows, there are three different states that a EEPROM bank can have: Empty, Current, and Used.

An Empty EEPROM Bank is signified by the 128 status bits all being 1s (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF). A Current EEPROM Bank is signified by the first 64 bits being set to 0x5A5A5A5A5A5A5A5A, with the remaining 64 bits set to 1. A Used EEPROM Bank is signified by all 128 bits being set to 0x5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A. These values can be changed if desired.

First, the program checks if the current EEPROM bank is Empty. If yes, the EEPROM bank has not been used and no further search is needed.

```
if (Bank_in_Use == EMPTY_BANK)
{
    Bank_Counter = i;
    return;
}
```

If an empty EEPROM bank is not encountered, the program checks if the bank is marked as Current. If yes, the EEPROM bank counter is updated and the page pointer is set to the first page of the EEPROM bank, to be adjusted further in a later loop. The loop is then exited as no further search is needed.

```
if (Bank_in_Use == CURRENT_BANK && Bank_Full != CURRENT_BANK)
{
    Bank_Counter = i;
    Page_Pointer = Bank_Pointer + WRITE_SIZE_BYTES*2;
    break;
}
```

Lastly, the program checks if the EEPROM bank is Used. If so, the EEPROM bank pointer is updated to the next EEPROM bank, and the loop continues.

```
if (Bank_in_Use == CURRENT_BANK && Bank_Full == CURRENT_BANK)
{
    Bank_Pointer += Bank_Size;
}
```

After the current EEPROM bank has been located, the current page can be found. There are three different states a page can have: Blank, Current, and Used.

An empty Page is identified by all 128 status bits being 1 (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF). A Current EEPROM Bank is identified by the first 64 bits being set to 0xA5A5A5A5A5A5A5A5, with the remaining 64 bits set to 1. A Used EEPROM Page is identified by all 128 bits being set to 0xA5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5. These status codes can be changed in EEPROM_Config.h if desired.

First, the program checks if the page is Blank or Current. If so, the correct page has been found and the loop is exited.

```
if (Page_in_Use == BLANK_PAGE)
{
    Page_Counter = i;
    break;
}

if (Page_in_Use == CURRENT_PAGE && Page_Full != CURRENT_PAGE)
{
    Page_Counter = i + 1;
    break;
}
```

If not Blank or Current, then the page must be used, and the page pointer is updated to the next page to test its status.

```
Page_Pointer += WRITE_SIZE_BYTES*2 + EEPROM_PAGE_DATA_SIZE;
```

At this point, the current EEPROM bank and page have been found and the calling function can continue. As a final step, this function will check if all EEPROM banks and pages have been used. If so, the last bank and page are marked as full for completeness, and the sector(s) used for emulation are erased.

```
if ((!ReadFlag) && (Bank_Counter == NUM_EEPROM_BANKS - 1) && (Page_Counter == NUM_EEPROM_PAGES))
{
    EEPROM_UpdatePageStatus();
    EEPROM_UpdateBankStatus();
    EEPROM_Erase();
}
```

This check is performed by testing the EEPROM bank and page counters. The number of EEPROM banks and pages indicating a full EEPROM will depend on the application configuration. These counters are set when testing for the current EEPROM banks and pages as shown in the code snippets above.

To prevent premature erasure when reading from a full EEPROM unit, this check is not made when the Read_Flag is set.

As show above, if the memory is full, the last page and bank are marked as Used for completeness before the EEPROM_Erase() functions are called and the bank and page pointers are reset.

5.2.2.2 EEPROM_UpdateBankStatus

The EEPROM_UpdateBankStatus() function provides functionality for updating the EEPROM bank status. This function called from the EEPROM_Write_Page() function. The EEPROM bank status is first read to determine how to proceed.

```
uint8_t Current_Bank_Status = *(Bank_Pointer);
```

If this status indicates the EEPROM bank is empty, the status is changed to current and programmed.

```
if (Current_Bank_Status == EMPTY_BANK)
{
    Fill_Buffer(Bank_Status, status_buffer_len, CURRENT_BANK);

    EEPROM_Write_Buffer(Bank_Pointer, Bank_Status);

    Page_Counter = 0;
    Page_Pointer = Bank_Pointer + WRITE_SIZE_BYTES*2;
}
```

If the status is not empty, the next check is for a full EEPROM bank. In this case, the current EEPROM bank's status will be updated to full and the next bank's status will be set to current to allow programming of the next EEPROM bank. Lastly, the page pointer is updated to the first page of the new EEPROM bank.

```
else if (Current_Bank_Status == CURRENT_BANK && Page_Counter == NUM_EEPROM_PAGES)
{
    Fill_Buffer(Bank_Status, status_buffer_len, CURRENT_BANK);

    EEPROM_Write_Buffer(Bank_Pointer + WRITE_SIZE_BYTES, Bank_Status);

    Bank_Pointer += Bank_Size;

    if (Bank_Counter == NUM_EEPROM_BANKS - 1 && Page_Counter == NUM_EEPROM_PAGES)
    {
        return;
    }

    Fill_Buffer(Bank_Status, status_buffer_len, CURRENT_BANK);
    EEPROM_Write_Buffer(Bank_Pointer, Bank_Status);
    Page_Counter = 0;
    Page_Pointer = Bank_Pointer + WRITE_SIZE_BYTES*2;

}
```

5.2.2.3 EEPROM_UpdatePageStatus

The EEPROM_UpdatePageStatus() function provides functionality for updating the previous page's status. This function is called from the EEPROM_Write_Page() function. The page status is first read to determine how to proceed.

```
uint8_t Current_Page_Status = *(Page_Pointer);
```

If this status indicates that the page is blank, the function exits. The page status will be updated in the EEPROM_Write_Page() function. Otherwise, the page status is updated to show it is full and the page pointer is incremented to prepare to program the next page:

```
if (Current_Page_Status == BLANK_PAGE)
{
    return;
}

Fill_Buffer(Page_Status, status_buffer_len, CURRENT_PAGE);

EEPROM_Write_Buffer(Page_Pointer + WRITE_SIZE_BYTES, Page_Status);

Page_Pointer += EEPROM_PAGE_DATA_SIZE + WRITE_SIZE_BYTES*2;
```

5.2.2.4 EEPROM_UpdatePageData

The EEPROM_UpdatePageData() function provides functionality for updating the EEPROM page data. This function is called from the EEPROM_Write_Page() function and programs the page to flash in 64-bit increments.

```
uint32_t i, Page_Offset;
for(i = 0; i < EEPROM_PAGE_DATA_SIZE / WRITE_SIZE_BYTES; i++)
{
    Page_Offset = WRITE_SIZE_BYTES*2 + (WRITE_SIZE_BYTES*i);
    EEPROM_Write_Buffer(Page_Pointer + Page_Offset, Write_Buffer + (i*WRITE_SIZE_BYTES));
}
```

If the programming was successful, the page is marked as current and the Empty_EEPROM flag is cleared.

```
Fill_Buffer(Page_Status, status_buffer_len, CURRENT_PAGE);
EEPROM_Write_Buffer(Page_Pointer, Page_Status);

Empty_EEPROM = 0;
```

5.2.2.5 EEPROM_Write_Page

The EEPROM_Write_Page() function provides the functionality for programming the data to Flash. It leverages the Flash API directly and makes several function calls within to prepare for data programming. The functions called are listed below:

- EEPROM_GetValidBank()
- EEPROM_UpdatePageStatus()
- EEPROM_UpdateBankStatus()
- EEPROM_UpdatePageData()

Each of the above functions are described in detail in their respective sections. To begin, the current EEPROM bank and page are found. Then the previous page is marked as used and the EEPROM bank status is updated if the bank changed. Finally, the data is programmed during the EEPROM page data update.

```
EEPROM_GetValidBank(0);
EEPROM_UpdatePageStatus();
EEPROM_UpdateBankStatus();
EEPROM_UpdatePageData(Write_Buffer);
```

5.2.3 64-Bit Mode Functions

5.2.3.1 EEPROM_64_Bit_Mode_Check_EOS

The EEPROM_64_Bit_Mode_Check_EOS() determines if the EEPROM unit is full and erases it if so.

First, the end address of EEPROM is set according to the device being used and the configuration. The END_OF_SECTOR directive is set in the EEPROM_Config.h file.

```
uint8_t* End_Address = (uint8_t*) END_OF_SECTOR;
```

Next, the EEPROM bank pointer is compared to the end address. If writing 8 bytes beginning at the current EEPROM bank pointer would go beyond the End Address, this indicates the EEPROM unit is full. At this point, the EEPROM unit is erased, performs a blank check and the EEPROM Bank Pointer is reset to the beginning of the EEPROM Unit.

```
if (Bank_Pointer > End_Address - WRITE_SIZE_BYTES)
{
    EEPROM_Erase();
}
```

5.2.3.2 EEPROM_Write_64_Bits

The EEPROM_Write_64_Bits() function provides functionality for programming eight bytes to memory. The first parameter, Num_Bytes, allows the user to specify how many valid words are written. The data words are assigned to the first indices of the Write_Buffer. If less than eight bytes are specified in the function call, missing bytes are filled with 0xFF. This is done to comply with ECC requirements.

First, the program checks if the EEPROM is full.

```
EEPROM_64_Bit_Mode_Check_EOS();
```

Next, the Write Buffer is filled with 1s if less than 8 bytes are specified.

```
uint8_t i;
for (i = Num_Bytes; i < WRITE_SIZE_BYTES; i++)
{
    Write_Buffer[i] = 0xFF;
}
```

Then, the data is programmed and the pointer is incremented to the next empty location

```
EEPROM_Write_Buffer(Bank_Pointer, Write_Buffer);

Empty_EEPROM = 0;

Bank_Pointer += WRITE_SIZE_BYTES;
```

Note

This function cannot be used until RESET_BANK_POINTER has been executed to set the pointer. Executing before this will produce unknown results.

5.2.4 Functions Used in Both Modes

5.2.4.1 EEPROM_Erase

The EEPROM_Erase() function provides functionality for erasing the sector(s) used for emulation and resetting the bank and page pointers. At least one entire sector must be erased, partial erase is not supported. Before erasing, ensure that stored data is no longer needed/valid. In the Single Unit implementation, this function is only called when all EEPROM Banks and Pages are full.

The function configures the Write/Erase Protection masks for the EEPROM unit, then calls the Erase_Bank function before resetting the pointers to the start of the bank.

```
Set_Protection_Masks();
Erase_Bank();
RESET_BANK_POINTER;
RESET_PAGE_POINTER;
```

5.2.4.2 EEPROM_Read

The EEPROM_Read() function provides functionality for reading the most recently written data and storing that data into a temporary buffer. This function can be used for debug purposes or to read stored data at runtime. The behavior differs in Page Mode vs 64-bit mode. In general, the most recently written data (page or 64-bits) are stored in the Read_Buffer.

The function verifies that data has been written to EEPROM by checking the Empty_EEPROM flag. If attempting to read data before any has been written, the values read into the buffer are invalid and an error is thrown.

```
if (Empty_EEPROM)
{
   Sample_Error();
}
```

Page Mode: If data has been written, the current EEPROM Bank and Page are found and then the Read Buffer is filled.

```
EEPROM_GetValidBank(1);

Page_Pointer += WRITE_SIZE_BYTES*2;

uint32 i;
for (i = 0; i < DATA_SIZE; i++)
{
    Read_Buffer[i] = *(Page_Pointer++);
}
```

64-Bit Mode: The function verifies data has been written to EEPROM by checking the Empty_EEPROM flag. If attempting to read data before any has been written, the values read into the buffer are invalid and an error is thrown. If data has been written, the pointer is moved back by four addresses (64-bits total) and the Read Buffer is filled with the data.

```
Bank_Pointer -= WRITE_SIZE_BYTES;
uint32_t i;
for (i = 0; i < WRITE_SIZE_BYTES; i++)
{
    Read_Buffer[i] = *(Bank_Pointer++);
}
```

5.2.5 Utility Functions

5.2.5.1 EEPROM_Write_Buffer

EEPROM_Write_Buffer() takes a pointer to an address in flash the write location) and a pointer to a 64-bit write buffer as input. It calls all the necessary FlashAPI functions to commit the write buffer to flash at the specified address.

It begins by clearing the FSM status and setting the proper protection masks.

```
Fapi_StatusType oReturnCheck;
Fapi_FlashStatusType  oFlashStatus;
Fapi_FlashStatusWordType oFlashStatusWord;

ClearFSMStatus();

Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTA, WE_Protection_A_Mask);
```

```
Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTB, WE_Protection_B_Mask);
```

Then the data from the write buffer is programmed to flash.

```
oReturnCheck = Fapi_issueProgrammingCommand((uint32_t*) address, (uint8_t*) write_buffer,
        WRITE_SIZE_BYTES, 0, 0, Fapi_AutoEccGeneration, u32UserFlashConfig);

while (Fapi_checkFsmForReady((uint32_t) address, u32UserFlashConfig) == Fapi_Status_FsmBusy);
```

Finally, it checks for any programming errors and verifies the correct data was written.

```
if (oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}

oFlashStatus = Fapi_getFsmStatus((uint32_t) address, u32UserFlashConfig);
if (oFlashStatus != 3)
{
    FMSTAT_Fail();
}

oReturnCheck = Fapi_doVerify((uint32_t*) address, VERIFY_LEN, (uint32_t*) write_buffer,
        &oFlashStatusWord, 0, u32UserFlashConfig);

if (oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}
```

5.2.5.2 Erase_Bank

The Erase_Bank function leverages the Flash API to erase a full EEPROM Unit. This function is only a wrapper around Flash API, and the protection masks are set in the EEPROM_Erase() functions.

It begins by clearing the FSM status and copying the protection masks into Flash API.

```
ClearFSMStatus(FLASH_BANK_SELECT, u32UserFlashConfig);

Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTA, WE_Protection_A_Mask);

Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTB, WE_Protection_B_Mask);
```

Then, it erases the flash and checks for programming errors.

```
oReturnCheck = Fapi_issueBankEraseCommand((uint32_t*) FLASH_BANK_SELECT, 0, u32UserFlashConfig);

while(Fapi_checkFsmForReady((uint32_t) FLASH_BANK_SELECT, u32UserFlashConfig) ==
Fapi_Status_FsmBusy);

if (oReturnCheck != Fapi_Status_Success)
    Sample_Error();

oFlashStatus = Fapi_getFsmStatus((uint32_t) FLASH_BANK_SELECT, u32UserFlashConfig);
if (oFlashStatus != 3)
{
    FMSTAT_Fail();
}
```

Finally, if Erase_Blank_Check is set, a blank check is performed.

```
if (Erase_Blank_Check)
{
    uint32_t address = FLASH_BANK_SELECT + FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT][0] *
FLASH_SECTOR_SIZE;
    Fapi_FlashStatusWordType oFlashStatusWord;
    oReturnCheck = Fapi_doBlankCheck((uint32_t*) address, BLANK_CHECK_LEN, &oFlashStatusWord, 0,
                u32UserFlashConfig);
    if (oReturnCheck != Fapi_Status_Success)
    {
        Sample_Error();
    }
}
```

5.2.5.3 Set_Protection_Masks

Set_Protection_Masks() is a simple wrapper around Configure_Protection_Masks() that updates the global mask variables (WE_Protection_A_Mask and WE_Protection_B_Mask) with the correct values for the current active unit.

```
uint64_t WE_Protection_AB_Mask = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR,
                                                            NUM_EEPROM_SECTORS);

WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32_t)WE_Protection_AB_Mask;
WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;
```

5.2.5.4 Configure_Protection_Masks

The Configure_Protection_Masks provides functionality to disable Write/Erase protection for any sector selected for EEPROM Emulation. This is done by calculating the appropriate Masks to pass to the Fapi_setupBankSectorEnable function. It requires two parameters, a pointer to the selected Flash Sector numbers, and the number of Flash Sectors to be emulated. For more information on the implementation of the Fapi_setupBankSectorEnable function, see the *F29H85x Flash API Reference Guide*.

The return value of this function is used to disable Write/Erase protection in Flash Sectors selected for EEPROM Emulation.

```
uint64 Protection_Mask_Sectors = 0;

if (Num_EEPROM_Sectors > 1)
{
    uint64_t Unshifted_Sectors;
    uint8_t Shift_Amount;

    // All sectors use Mask A
    if (Sector_Numbers[0] < 32 && Sector_Numbers[1] < 32)
    {
        // Push 1 out to 1 past the number of sectors
        Unshifted_Sectors = (uint64_t) 1 << Num_EEPROM_Sectors;
        // Subtract 1 --> now we have all 1s for the sectors we want
        Unshifted_Sectors -= 1;
        // Shift over by start location and OR with master
        Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);
    }
    // All sectors use Mask B
    else if (Sector_Numbers[0] > 31 && Sector_Numbers[1] > 31)
    {
        Shift_Amount = ((Sector_Numbers[1] - 32) / 8) - ((Sector_Numbers[0] - 32) / 8) + 1;
        Unshifted_Sectors = (uint64) 1 << Shift_Amount;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << ((Sector_Numbers[0] - 32)/8));
        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;
    }
    // Mix of Masks A and B
    else
    {
        // Configure Mask B
        Shift_Amount = ((Sector_Numbers[1] - 32)/8) + 1;
        Unshifted_Sectors = (uint64) 1 << Shift_Amount;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= Unshifted_Sectors;

        // Zero out the bottom half so we can configure Mask A
        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;

        // Configure Mask A
        Unshifted_Sectors = (uint64) 1 << ((32 - Sector_Numbers[0]) + 1);
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);
    }
}
// Only using 1 sector
else
{
    // Mask A
    if(Sector_Numbers[0] < 32)
    {
        Protection_Mask_Sectors |= ((uint64) 1 << Sector_Numbers[0]);
    }
    // Mask B
    else
    {
        Protection_Mask_Sectors |= ((uint64) 1 << ((Sector_Numbers[0] - 32)/8));
        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;
    }
}

return Protection_Mask_Sectors;
```

5.2.5.5 Fill_Buffer

Fill_Buffer() is a very basic helper function to fill up the various status and write buffers. It takes a target buffer, its length, and a value as input and fills the buffer with that value.

```
uint8_t i;
for (i = 0; i < buffer_len; i++)
{
    buffer[i] = value;
}
```

5.2.5.6 ClearFSMStatus

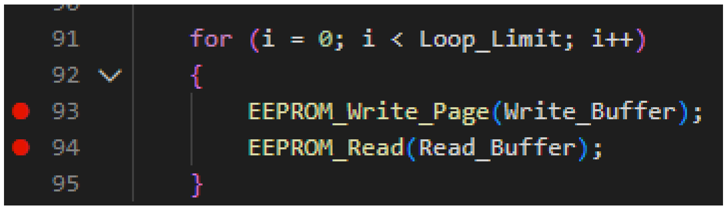
The ClearFSMStatus() function is responsible for clearing the status of the previous flash operation. This function must be used as-is.

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_StatusType  oReturnCheck;

while (Fapi_checkFsmForReady(u32StartAddress, u32UserFlashConfig) != Fapi_Status_FsmReady){}
oFlashStatus = Fapi_getFsmStatus(u32StartAddress, u32UserFlashConfig);

oReturnCheck = Fapi_issueAsyncCommand(u32StartAddress, u32UserFlashConfig, Fapi_ClearStatus);

while (Fapi_getFsmStatus(u32StartAddress, u32UserFlashConfig) != 0) {}

if(oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}
```

5.3 Testing Example

The examples provided were tested with F29H859TU8. To properly test the example, the memory window and breakpoints need to be utilized within Code Composer Studio™. The following steps were followed to program and test the project:

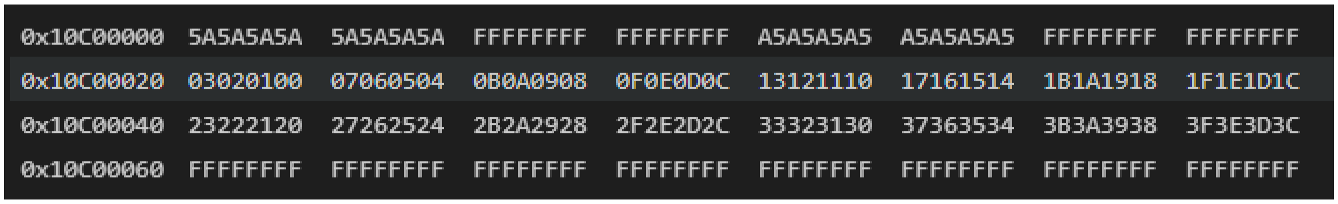
1. Connect the F29H859TU8 to the PC via USB and an XDS110 Debug Probe with JTAG connection.
2. Connect a 5V DC power supply to the board.
3. Start Code Composer Studio and open the F29H85x_EEPROM_Example.pjt.
4. Build the project by selecting Project -> Build Project.
5. Launch the project by right-clicking it in the explorer, then selecting "Debug Project".
6. Set breakpoints to properly view data written to and read from the memory within the memory window as shown in Break Points.



```
for (i = 0; i < Loop_Limit; i++)
{
    EEPROM_Write_Page(Write_Buffer);
    EEPROM_Read(Read_Buffer);
}
```

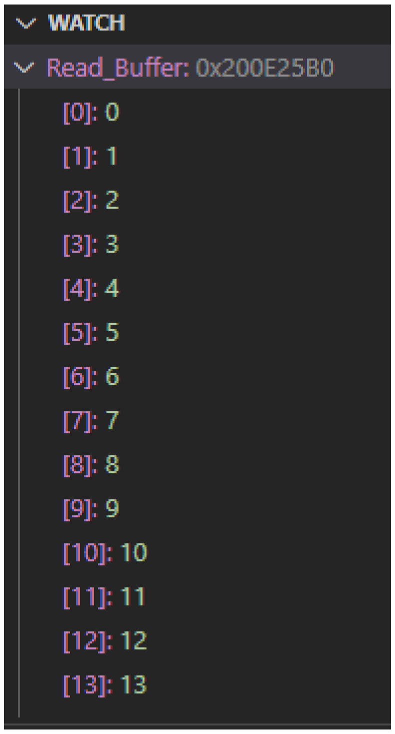
Figure 5-2. Break Points

7. Run to the first breakpoint and open the Memory Browser (View -> Memory Browser) to view the data. Bank_Pointer can be used to watch the data written and Read_Buffer to watch the data being read back from the memory. This is shown in Write to EEPROM and Read Data.



```
0x10C00000  5A5A5A5A  5A5A5A5A  FFFFFFFF  FFFFFFFF  A5A5A5A5  A5A5A5A5  FFFFFFFF  FFFFFFFF
0x10C00020  03020100  07060504  0B0A0908  0F0E0D0C  13121110  17161514  1B1A1918  1F1E1D1C
0x10C00040  23222120  27262524  2B2A2928  2F2E2D2C  33323130  37363534  3B3A3938  3F3E3D3C
0x10C00060  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

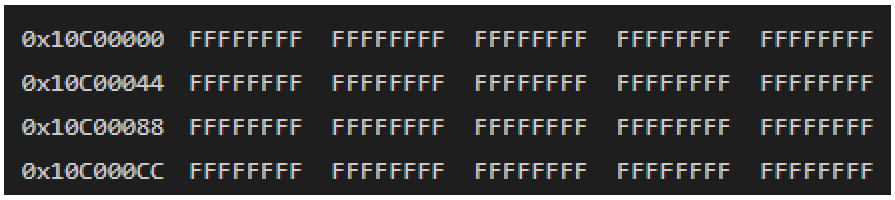
Figure 5-3. Write to EEPROM



```
WATCH
Read_Buffer: 0x200E25B0
  [0]: 0
  [1]: 1
  [2]: 2
  [3]: 3
  [4]: 4
  [5]: 5
  [6]: 6
  [7]: 7
  [8]: 8
  [9]: 9
  [10]: 10
  [11]: 11
  [12]: 12
  [13]: 13
```

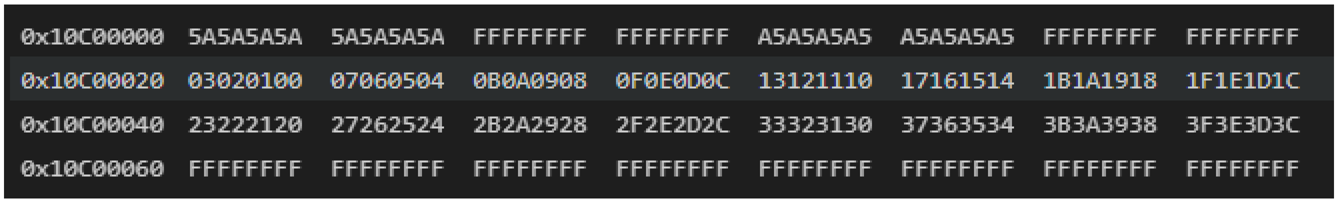
Figure 5-4. Read Data

8. Continue running from breakpoint to breakpoint until the program has finished or EEPROM is full.
9. Once EEPROM is full, you will see the new data written to the previously inactive unit and the full EEEPROM will be erased. Shown in Erase full EEPROM Unit and Write to EEPROM.



```
0x10C00000  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
0x10C00044  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
0x10C00088  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
0x10C000CC  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

Figure 5-5. Erase Full EEPROM Unit



```
0x10C00000  5A5A5A5A  5A5A5A5A  FFFFFFFF  FFFFFFFF  A5A5A5A5  A5A5A5A5  FFFFFFFF  FFFFFFFF
0x10C00020  03020100  07060504  0B0A0908  0F0E0D0C  13121110  17161514  1B1A1918  1F1E1D1C
0x10C00040  23222120  27262524  2B2A2928  2F2E2D2C  33323130  37363534  3B3A3938  3F3E3D3C
0x10C00060  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

Figure 5-6. Write to EEPROM

10. This process can be repeated as necessary.

The preceding steps were used to test the Page Mode configuration. The 64-bit mode configuration can also be tested with the same procedure. To enable 64-bit mode, change the definition in the EEPROM_Config.h file by un-commenting the _64_BIT_MODE directive and commenting out the PAGE_MODE directive.

6 Ping-Pong Emulation

In this section, the Ping-Pong implementation is discussed. To review the behavior of this implementation, see Ping Pong Behavior.

6.1 User-Configuration

The implementation detailed in this document allows you to configure several variables for EEPROM Emulation. These variables are found within EEPROM_PingPong_Config.h and F29H85x_EEPROM_PingPong.c.

6.1.1 EEPROM_PingPong_Config.h

EEPROM_PingPong_Config.h contains definitions that allow the user to change various aspects of EEPROM configuration. These aspects include:

- Choose between Page Mode and 64-Bit Mode.

```
//#define _64_BIT_MODE 1
#define PAGE_MODE 1
```

- Choose how many EEPROM Banks to emulate.

```
#define NUM_EEPROM_BANKS 4
```

- Choose how many EEPROM Pages within each EEPROM Bank

```
#define NUM_EEPROM_PAGES 5
```

- Choose the size of the data space contained within each EEPROM Page (unit is bytes). Although any size can be specified, the size will be adjusted to the closest multiple of eight that is greater than or equal to the number given. For example, a specified size of six bytes per page will be programmed as eight bytes per page, with the last two being treated as 0xFFFF. This is to comply with Flash requirements (8 bits of ECC is programmed for every 64-bit aligned Flash memory address).

```
#define DATA_SIZE 64
```

6.1.2 F29H85x_EEPROM_PingPong.c

Within F29H85x_EEPROM_PingPong.c, users can choose which Flash Sectors to use for EEPROM emulation. The sectors chosen (if multiple) should be contiguous and in order from least to greatest. Insert only the First and Last sectors to be used for EEPROM. For example, to use sectors 1-10, insert {1,10}. To only use sector 1, insert {1,1}.

```
uint32 FIRST_AND_LAST_SECTOR[2][2] = {{0,0},{1,1}};
```

A valid configuration has the following properties:

- Imply a valid and consistent amount of sectors between the two EEPROM units
- Only include a sector(s) that exist on the device
- Not create an overlap in the Write/Erase Protection Masks between the two units
  - The F29H85x Flash API requires Write/Erase Protection Masks to be configured before programming Flash Memory. Details about the proper configuration of these masks can be found in the *F29H85x Flash API Reference Guide* (SPRUJE7).

More details about invalid or dangerous configurations can be found in Section 6.2.1.2.

Users can additionally enable blank check after erase and choose which EEPROM unit to begin emulation in.

```
uint8_t EEPROM_ACTIVE_UNIT = 0;
uint8_t Erase_Blank_Check  = 1;
```

If set to 0, the first set of Flash Sectors in FIRST_AND_LAST_SECTOR will be the Active EEPROM Unit first, and the second set will be Inactive EEPROM unit at first. If set to 1, the opposite will be true.

6.2 EEPROM Functions

Listed below are all functions required for the implementations. They are found in the F29H85x_EEPROM_PingPong.c or F29H85x_EEPROM_PingPong_Example.c files.

Initialization and Setup Functions

- Configure_Device()
- EEPROM_Config_Check()

Page Mode Functions

- EEPROM_GetValidBank(uint8_t ReadFlag)
- EEPROM_UpdateBankStatus()
- EEPROM_UpdatePageStatus()
- EEPROM_UpdatePageData(uint8_t* Write_Buffer)
- EEPROM_Write_Page(uint8_t* Write_Buffer)

64-bit Mode Functions

- EEPROM_64_Bit_Mode_Check_EOS()
- EEPROM_Write_64_Bits(uint8_t Num_Bytes, uint8_t* Write_Buffer)

Used in Both

- EEPROM_Erase_Inactive_Unit()
- EEPROM_Read()
- EEPROM_Erase_All()

Utility Functions

- EEPROM_Write_Buffer(uint8_t* address, uint8_t* write_buffer)
- Erase_Bank()
- Configure_Protection_Masks(uint32_t* Sector_Numbers, uint32_t Num_EEPROM_Sectors)
- Set_Protection_Masks()
- Fill_Buffer()
- ClearFSMStatus(uint32_t u32StartAddress, uint32_t u32UserFlashConfig)

The description of each of these functions is discussed in detail in the subsequent sections.

6.2.1 Initialization and Setup Functions

6.2.1.1 Configure_Device

This function encapsulates all standard device and FlashAPI setup.

First, it initializes the device and its peripherals.

```
Device_init();
Flash_initModule(3);

Device_initGPIO();

Interrupt_initModule();

Interrupt_initVectorTable();

__asm(" ENINT")
```

Then, it requests the flash semaphore and initializes the Flash API with the chosen user configuration.

```
HWREG(SSUGEN_BASE + SSU_O_FLSEMREQ ) =  1;
while ((HWREG( SSUGEN_BASE + SSU_O_FLSEMSTAT) & SSU_FLSEMSTAT_CPU_M)!= (0x1<<SSU_FLSEMSTAT_CPU_S));

u32UserFlashConfig = Fapi_getUserConfiguration(BankType, FOTAStatus);

Fapi_SetFlashCPUConfiguration(u32UserFlashConfig);

oReturnCheck = Fapi_initializeAPI((Fapi_FmcRegistersType*) FLASHCONTROLLER1_BASE, 200);

if(oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}
```

6.2.1.2 EEPROM_Config_Check

The EEPROM_Config_Check() function provides general error-checking and configures Write/Erase protection masks required by the Flash API. This function should be called before programming or reading from the emulated EEPROM Unit(s).

First, the function verifies that the Flash Bank selected for EEPROM Emulation is valid. Only the data bank is a valid selection on F29x.

```
if (FLASH_BANK_SELECT != C29FlashBankFR4RP0StartAddress)
{
    return 0xFFFF;
}
```

Second, the validity of Flash Sectors selected for emulation is examined. This function checks for:

- FIRST_AND_LAST_SECTOR indicating two different numbers of Flash Sectors between the two units

  ```
  uint32_t NUM_EEPROM_SECTORS_1 = FIRST_AND_LAST_SECTOR[0][1] - FIRST_AND_LAST_SECTOR[0][0] + 1;
  uint32_t NUM_EEPROM_SECTORS_2 = FIRST_AND_LAST_SECTOR[1][1] - FIRST_AND_LAST_SECTOR[1][0] + 1;

  if (NUM_EEPROM_SECTORS_1 != NUM_EEPROM_SECTORS_2)
  {
      return 0xEEEE;
  }
  ```

- More Flash Sectors selected for emulation than available within the Flash Bank

  ```
  if (NUM_EEPROM_SECTORS > NUM_FLASH_SECTORS || NUM_EEPROM_SECTORS == 0)
  {
      return 0xEEEE;
  }
  ```

- Invalid combinations for First and Last Sectors selected for emulation

```
if (NUM_EEPROM_SECTORS > 1)
{
    if (FIRST_AND_LAST_SECTOR[0][1] <= FIRST_AND_LAST_SECTOR[0][0])
    {
        return 0xEEEE;
    }

    if (FIRST_AND_LAST_SECTOR[1][1] <= FIRST_AND_LAST_SECTOR[1][0])
    {
        return 0xEEEE;
    }

    if (FIRST_AND_LAST_SECTOR[0][1] > NUM_FLASH_SECTORS - 1 || FIRST_AND_LAST_SECTOR[0][1] < 1)
    {
        return 0xEEEE;
    }

    if (FIRST_AND_LAST_SECTOR[1][1] > NUM_FLASH_SECTORS - 1 || FIRST_AND_LAST_SECTOR[1][1] < 1)
    {
        return 0xEEEE;
    }
}
else if (FIRST_AND_LAST_SECTOR[0][0] > NUM_FLASH_SECTORS - 1 ||
         FIRST_AND_LAST_SECTOR[1][0] > NUM_FLASH_SECTORS - 1)
{
        return 0xEEEE;
}
```

- Overlapping Sectors between the two units

```
if (FIRST_AND_LAST_SECTOR[0][0] <= FIRST_AND_LAST_SECTOR[1][1] &&
    FIRST_AND_LAST_SECTOR[1][0] <= FIRST_AND_LAST_SECTOR[0][1])
{
    return 0xEEEE;
}
```

- If using Page Mode, check if total size of EEPROM Banks + Pages will fit in the Flash Sectors selected.

```
Bank_Size = WRITE_SIZE_BYTES*2 +
            ((EEPROM_PAGE_DATA_SIZE + WRITE_SIZE_BYTES*2) * NUM_EEPROM_PAGES);

uint32_t Available_Words = NUM_EEPROM_SECTORS * FLASH_SECTOR_SIZE;

if (Bank_Size * NUM_EEPROM_BANKS > Available_Words)
{
    return 0xCCCC;
}
```

- Verify that the two EEPROM units do not have overlapping protection masks

```
uint64_t WE_Protection_AB_Sectors_Unit_0 = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR[0],
                                                                      NUM_EEPROM_SECTORS);
uint64_t WE_Protection_AB_Sectors_Unit_1 = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR[1],
                                                                      NUM_EEPROM_SECTORS);
if (WE_Protection_AB_Sectors_Unit_0 & WE_Protection_AB_Sectors_Unit_1)
{
    return 0xEEEE;
}
```

If one of the following nonfatal conditions is detected, a warning is issued. Each flag corresponds to a bit in the return value of the function:

- Space for one or more EEPROM Banks is left in Flash after configuring EEPROM Bank and Page size

```
if (Available_Words - (Bank_Size * NUM_EEPROM_BANKS ) >= Bank_Size)
{
    Warning_Flags += 1;
}
```

- If each page consists of less than or equal to 64 bits (8 bytes) (this wastes space as the 64-Bit Mode could be used without the need for Status Codes)

```
if (EEPROM_PAGE_DATA_SIZE <= WRITE_SIZE_BYTES)
{
    Warning_Flags += 2;
}
```

If using sectors in the 32-127 range (for F29H85x devices) and not using all eight sectors allocated to a single bit in the Write/Erase Protection Mask, a warning is issued. Any unused sectors within the eight designed by a single bit cannot be properly be protected from erase. For more information on how the Write/Erase Protection Masks correspond to sectors, see the *F29H85x Flash API Reference Guide*

```
uint8_t i;
for (i = 0; i < 2; i++)
{
    // If using any sectors > 31
    if (FIRST_AND_LAST_SECTOR[i][1] > 31)
    {
        // If all sectors are > 31 (use protection mask B)
        if (FIRST_AND_LAST_SECTOR[i][0] > 31)
        {
            // If using < 8 sectors (will be clearing more than necessary)
            if (NUM_EEPROM_SECTORS < 8)
            {
                Warning_Flags += 4;
                break;
            }
            // if sector isn't a multiple of 8 (will be clearing more than necessary)
            else if ((FIRST_AND_LAST_SECTOR[i][0] % 8) != 0 ||
                     (FIRST_AND_LAST_SECTOR[i][1] + 1) % 8 != 0)
            {
                Warning_Flags += 4;
                break;
            }
        }
        // if sector isn't a multiple of 8 (will be clearing more than necessary)
        else if ((FIRST_AND_LAST_SECTOR[i][1] + 1) % 8 != 0)
        {
            Warning_Flags += 4;
            break;
        }
    }
}
```

6.2.2 Page Mode Functions

6.2.2.1 EEPROM_GetValidBank

The EEPROM_GetValidBank() function provides functionality for finding the current EEPROM bank and page. This function is called by both the EEPROM_Write_Page() and EEPROM_Read() functions. GetValidBank Flow shows the overall flow required to search for current EEPROM bank and page.

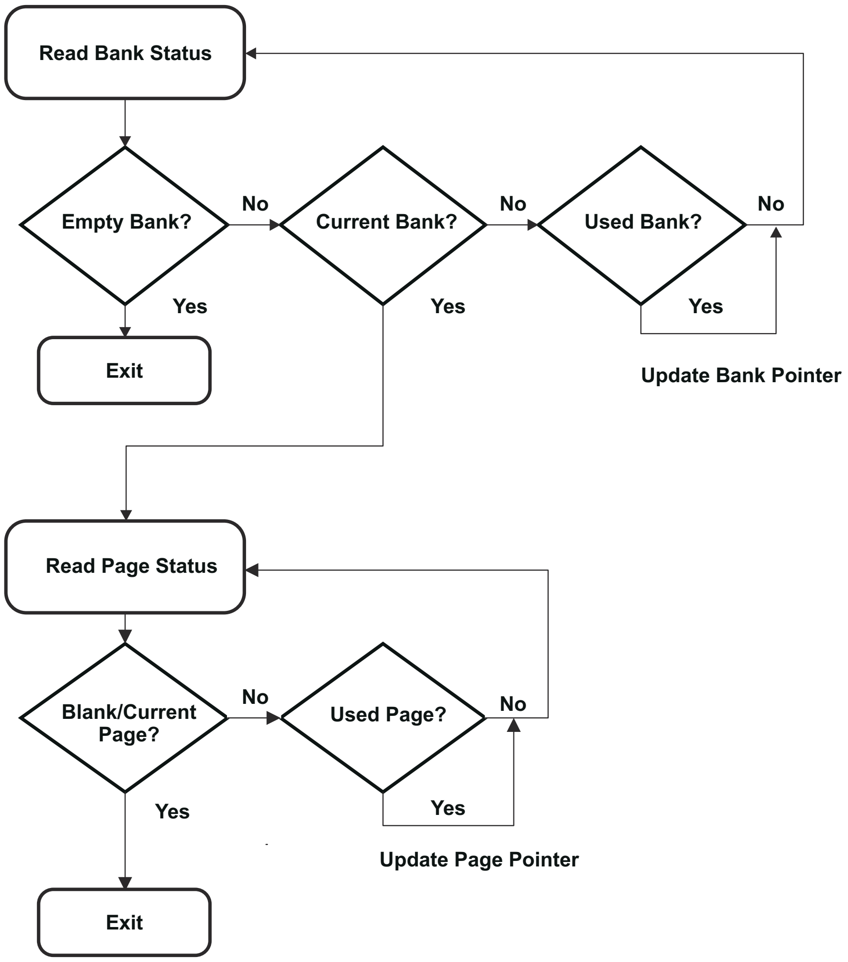


Figure 6-1. GetValidBank Flow

When entering this function, the EEPROM bank and page pointers are set to the beginning of the first sector specified in FIRST_AND_LAST_SECTOR:

```
RESET_BANK_POINTER;
RESET_PAGE_POINTER;
```

The addresses for these pointers are defined the EEPROM_Config.h file for the specific device and EEPROM configuration being used.

Next, the current EEPROM bank is found. As GetValidBank Flow shows, there are three different states that an EEPROM bank can have: Empty, Current, and Used.

An Empty EEPROM Bank is signified by the 128 status bits all being 1s (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF). A Current EEPROM Bank is signified by the most significant 64 bits being set to 0x5A5A5A5A5A5A5A5A, with the remaining 64 bits set to 1 (0x5A5A5A5A5A5A5A5AFFFFFFFFFFFFFFFF. A Used EERPOM Bank is signified by all 128 bits being set to 0x5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A. These values can be changed if desired.

First, the program checks for an Empty EEPROM bank. If this status is encountered, the EEPROM bank has not been used and no further search is needed.

```
if (Bank_in_Use == EMPTY_BANK)
{
    Bank_Counter = i;
    return;
}
```

If an Empty EEPROM Bank is not encountered, the Current status is checked for next. If present, the EEPROM bank counter is updated accordingly and the page pointer is set to the first page of the EEPROM bank to enable testing for the current page. The loop is then exited as no further search is needed.

```
if (Bank_in_Use == CURRENT_BANK && Bank_Full != CURRENT_BANK)
{
    Bank_Counter = i;
    Page_Pointer = Bank_Pointer + WRITE_SIZE_BYTES*2;
    break;
}
```

Lastly, a check for the Used status is made. In this case, the EEPROM bank has been used and the EEPROM bank pointer is updated to the next EEPROM bank to test its status.

```
if (Bank_in_Use == CURRENT_BANK && Bank_Full == CURRENT_BANK)
{
    Bank_Pointer += Bank_Size;
}
```

After the current EEPROM bank has been found, the current page needs to be found. there are three different states that a page can have: Empty, Current, and Used.

An Empty Page is signified by the 128 status bits all being 1s (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF). A Current Page is signified by the most significant 64 bits being set to 0xA5A5A5A5A5A5A5A5, with the remaining 64 bits set to 1 (0xA5A5A5A5A5A5A5A5FFFFFFFFFFFFFFFF). A Used Page is signified by all 128 bits being set to 0xA5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5. These values can be changed if desired.

The Blank and Current statuses are looked for first. If either of these are the current state of the page, the correct page is found and the loop is exited as further searching is not needed.

```
if (Page_in_Use == BLANK_PAGE)
{
    Page_Counter = i;
    break;
}

if (Page_in_Use == CURRENT_PAGE && Page_Full != CURRENT_PAGE)
{
    Page_Counter = i + 1;
    break;
}
```

If the page status is neither of these, the only other possibility is a Used Page. In this case, the page pointer is updated to the next page to test its status.

```
Page_Pointer += WRITE_SIZE_BYTES*2 + EEPROM_PAGE_DATA_SIZE;
```

At this point, the current EEPROM bank and page is found and the calling function can continue. As a final step, this function will check if all EEPROM banks and pages have been used. In this case, the sector needs to be erased. The last bank and page in the full unit are marked as used for completeness, active units are switched, Write/Erase Protection masks are reconfigured, and the Erase_Inactive_Unit flag is set.

```
if (!ReadFlag && Bank_Counter == NUM_EEPROM_BANKS - 1 && Page_Counter == NUM_EEPROM_PAGES)
{
    EEPROM_UpdatePageStatus();
    EEPROM_UpdateBankStatus();

    EEPROM_ACTIVE_UNIT ^= 1;
    Set_Protection_Masks();
    Erase_Inactive_Unit = 1;

   RESET_BANK_POINTER;
   RESET_PAGE_POINTER;
}
```

This check is performed by testing the EEPROM bank and page counters. The number of EEPROM banks and pages indicating a full EEPROM depends on the application. These counters are set when testing for the current EEPROM banks and pages as shown in the code snippets above. However, this check is not made when the Read_Flag is set. This is to prevent premature erasure of the inactive EEPROM unit when reading from a full EEPROM unit.

6.2.2.2 EEPROM_UpdateBankStatus

The EEPROM_UpdateBankStatus() function provides functionality for updating the EEPROM bank status. This function called from the EEPROM_Write_Page() function. The EEPROM bank status is first read to determine how to proceed.

```
uint8_t Current_Bank_Status = *(Bank_Pointer);
```

If this status indicates the EEPROM bank is empty, the status is changed to Current and programmed.

```
Fill_Buffer(Bank_Status, status_buffer_len, CURRENT_BANK);

EEPROM_Write_Buffer(Bank_Pointer, Bank_Status);

Page_Counter = 0;
Page_Pointer = Bank_Pointer + WRITE_SIZE_BYTES*2;
```

If the status is not empty, the next check is for a full EEPROM bank. In this case, the current EEPROM bank's status will be updated to show full and the next bank's status will be set to Current to allow programming of the next EEPROM bank. Lastly, the page pointer is updated to the first page of the new EEPROM bank.

```
Fill_Buffer(Bank_Status, status_buffer_len, CURRENT_BANK);

EEPROM_Write_Buffer(Bank_Pointer + WRITE_SIZE_BYTES, Bank_Status);

Bank_Pointer += Bank_Size;

if (Bank_Counter == NUM_EEPROM_BANKS - 1)
{
    return;
}

Fill_Buffer(Bank_Status, status_buffer_len, CURRENT_BANK);
EEPROM_Write_Buffer(Bank_Pointer, Bank_Status);
Page_Counter = 0;
Page_Pointer = Bank_Pointer + WRITE_SIZE_BYTES*2;
```

6.2.2.3 EEPROM_UpdatePageStatus

The EEPROM_UpdatePageStatus() function provides functionality for updating the previous page's status. This function is called from the EEPROM_Write_Page() function. The page status is first read to determine how to proceed.

```
uint8_t Current_Page_Status = *(Page_Pointer);
```

If this status indicates that the page is blank, the function exits. This status will be updated in the EEPROM_Write_Page() function. Otherwise, the page status is updated to show it is full and the page pointer is incremented to prepare to program the next page:

```
if (Current_Page_Status == BLANK_PAGE)
{
    return;
}

Fill_Buffer(Page_Status, status_buffer_len, CURRENT_PAGE);
EEPROM_Write_Buffer(Page_Pointer + WRITE_SIZE_BYTES, Page_Status);

Page_Pointer += EEPROM_PAGE_DATA_SIZE + WRITE_SIZE_BYTES*2;
```

6.2.2.4 EEPROM_UpdatePageData

The EEPROM_UpdatePageData() function provides functionality for updating the EEPROM page data. This function is called from the EEPROM_Write_Page() function.

The data from the write buffer is written to flash 64-bits at a time, with the offset being computed each loop iteration.

```
uint32_t i, Page_Offset;
for(i = 0; i < EEPROM_PAGE_DATA_SIZE / WRITE_SIZE_BYTES; i++)
{
    Page_Offset = WRITE_SIZE_BYTES*2 + (WRITE_SIZE_BYTES*i);
    EEPROM_Write_Buffer(Page_Pointer + Page_Offset, Write_Buffer + (i*WRITE_SIZE_BYTES));
}
```

If the programming is successful, the page is marked as current and the Empty_EEPROM flag is cleared. The code is shown below:

```
Fill_Buffer(Page_Status, status_buffer_len, CURRENT_PAGE);
EEPROM_Write_Buffer(Page_Pointer, Page_Status);

Empty_EEPROM = 0;
```

After a successful write, the function checks if the inactive EEPROM unit needs to be erased. If so, it erases the unit and clears the flag.

```
if (Erase_Inactive_Unit)
{
    EEPROM_Erase_Inactive_Unit();
    Erase_Inactive_Unit = 0;
}
```

6.2.2.5 EEPROM_Write_Page

The EEPROM_Write_Page() function provides the functionality for programming the data to Flash. It leverages the Flash API directly and makes several function calls within to prepare for data programming. The functions called are listed below:

- EEPROM_GetValidBank()
- EEPROM_UpdatePageStatus()
- EEPROM_UpdateBankStatus()
- EEPROM_UpdatePageData()

Each of the above functions are described in detail in their respective sections. To begin, the current EEPROM Bank and Page are found. After the current EEPROM Bank and Page are found, the Page Status of the previous Page is updated and the EEPROM Bank status is updated if a new EEPROM Bank is being used. Next, the actual programming occurs during the EEPROM page data update.

```
EEPROM_GetValidBank(0);
EEPROM_UpdatePageStatus();
EEPROM_UpdateBankStatus();
EEPROM_UpdatePageData(Write_Buffer);
```

6.2.3 64-Bit Mode Functions

6.2.3.1 EEPROM_64_Bit_Mode_Check_EOS

The EEPROM_64_Bit_Mode_Check_EOS() provides functionality for determining if the EEPROM unit is full and assigning the proper address, if required. If a full EEPROM unit is detected, the unit is flagged for erasure and the pointers are moved to the other, clean, unit.

First, the end address of EEPROM is retrieved according to the device being used and the configuration. The END_OF_SECTOR directive is set in the EEPROM_Config.h file.

```
uint8_t* End_Address = (uint8_t*) END_OF_SECTOR;
```

Next, the EEPROM bank pointer is compared to the end address. If writing 64 new bits would go beyond the end address, the unit is full and needs to be erased. The active EEPROM unit is switched, new Write/Protection masks are configured, the Erase_Inactive_Unit flag is set, and the EEPROM EEPROM Bank Pointer is reset to the beginning of the newly active EEPROM unit.

```
if (Bank_Pointer > End_Address - WRITE_SIZE_BYTES)
{
    EEPROM_ACTIVE_UNIT ^= 1;
    Set_Protection_Masks();
    Erase_Inactive_Unit = 1;
    RESET_BANK_POINTER;
}
```

6.2.3.2 EEPROM_Write_64_Bits

The EEPROM_Write_64_Bits() function provides functionality for programming 64 bits (8 bytes) to memory. The first parameter, Num_Bytes, allows the user to specify how many valid bytes will be written. A 64-bit minimum write is required by ECC. If less than 8 bytes, the data will be padded with 0xFF until a 64 bit buffer is achieved. The data bytes should be assigned to the first 8 locations of the Write_Buffer to be used by the Fapi_issueProgrammingCommand() function.

First, a full EEPROM unit is tested for.

```
EEPROM_64_Bit_Mode_Check_EOS();
```

Next, the Write Buffer is filled with 1s if less than 8 bytes are specified.

```
uint8_t i;
for (i = Num_Bytes; i < WRITE_SIZE_BYTES; i++)
{
    Write_Buffer[i] = 0xFF;
}
```

Next, data is programmed and the pointer is incremented to the next location to program data.

```
EEPROM_Write_Buffer(Bank_Pointer, Write_Buffer);

Empty_EEPROM = 0;

Bank_Pointer += WRITE_SIZE_BYTES;
```

Once programming is complete, the Erase_Inactive_Unit flag is checked. If set, the inactive unit is erased and the flag reset.

```
if (Erase_Inactive_Unit)
{
    EEPROM_Erase_Inactive_Unit();
    Erase_Inactive_Unit = 0;
}
```

Note

This function cannot be used until RESET_BANK_POINTER has been executed to set the pointer. Executing before could produce unknown results.

6.2.4 Functions Used in Both Modes

6.2.4.1 EEPROM_Erase_Inactive_Unit

The EEPROM_Erase_Inactive_Unit() function provides functionality for erasing the inactive sector(s) used for emulation. At least one entire sector must be erased as partial erase is not supported. Before erasing, you must ensure that stored data is no longer needed/valid. In the Ping Pong implementation, this function is only called when all EEPROM banks and pages in one EEPROM unit are used and data is successfully written to the other EEPROM unit. The function begins by re-calculating the Write/Erase Protection masks for the inactive (full) EEPROM unit, and then calls the Erase_Bank function.

```
EEPROM_ACTIVE_UNIT ^= 1;
Set_Protection_Masks();

Erase_Bank();

EEPROM_ACTIVE_UNIT ^= 1;
Set_Protection_Masks();
```

6.2.4.2 EEPROM_Read

The EEPROM_Read() function provides functionality for reading the most recently written data and storing it into a temporary buffer. This function can be used for debug purposes or to read stored data at runtime. The behavior differs in Page Mode vs 64-bit mode. In general, the most recently written data (page or 64-bits) are stored in the Read_Buffer.

First, the function verifies that data has been written to EEPROM by checking the Empty_EEPROM flag. If attempting to read data before any has been written, the values read into the buffer are invalid and an error is thrown.

```
if (Empty_EEPROM)
{
   Sample_Error();
}
```

Page Mode: If data has been written, the current EEPROM Bank and Page are found and then the Read Buffer is filled.

```
EEPROM_GetValidBank(1);

Page_Pointer += WRITE_SIZE_BYTES*2;

uint32_t i;
for (i = 0; i < DATA_SIZE; i++)
{
    Read_Buffer[i] = *(Page_Pointer++);
}
```

64-Bit Mode: The pointer is moved back by eight addresses (64 bits total) and the Read Buffer is filled with the data.

```
Bank_Pointer -= WRITE_SIZE_BYTES;
uint32 i;
for (i = 0; i < WRITE_SIZE_BYTES; i++)
{
    Read_Buffer[i] = *(Bank_Pointer++);
}
```

6.2.4.3 EEPROM_Erase_All

The EEPROM_Erase_All function is run at the start of the program, and computes the combined protection masks before erasing all sectors and resetting all pointers used in the emulation.

```
uint64_t WE_Protection_AB_Sectors_Unit_0 = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR[0],
                                                                    NUM_EEPROM_SECTORS);

uint64_t WE_Protection_AB_Sectors_Unit_1 = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR[1],
                                                                    NUM_EEPROM_SECTORS);

uint32_t Combined_WE_Protection_A_Sectors = (uint32_t)WE_Protection_AB_Sectors_Unit_0 |
        (uint32)WE_Protection_AB_Sectors_Unit_1;

uint32_t Combined_WE_Protection_B_Sectors = WE_Protection_AB_Sectors_Unit_0 >> 32 |
        WE_Protection_AB_Sectors_Unit_1 >> 32;

WE_Protection_A_Mask = 0xFFFFFFFF ^ Combined_WE_Protection_A_Sectors;
WE_Protection_B_Mask = 0x00000FFF ^ Combined_WE_Protection_B_Sectors;

Erase_Bank();

Set_Protection_Masks();

RESET_BANK_POINTER;
RESET_PAGE_POINTER;
```

6.2.5 Utility Functions

6.2.5.1 EEPROM_Write_Buffer

EEPROM_Write_Buffer() takes a pointer to an address in flash the write location) and a pointer to a 64-bit write buffer as input. It calls all the necessary FlashAPI functions to commit the write buffer to flash at the specified address.

It begins by clearing the FSM status and setting the proper protection masks.

```
Fapi_StatusType oReturnCheck;
Fapi_FlashStatusType  oFlashStatus;
Fapi_FlashStatusWordType oFlashStatusWord;

ClearFSMStatus();

Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTA, WE_Protection_A_Mask);

Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTB, WE_Protection_B_Mask);
```

Then the data from the write buffer is programmed to flash.

```
oReturnCheck = Fapi_issueProgrammingCommand((uint32_t*) address, (uint8_t*) write_buffer,
        WRITE_SIZE_BYTES, 0, 0, Fapi_AutoEccGeneration, u32UserFlashConfig);

while (Fapi_checkFsmForReady((uint32_t) address, u32UserFlashConfig) == Fapi_Status_FsmBusy);
```

Finally, it checks for any programming errors and verifies the correct data was written.

```
if (oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}

oFlashStatus = Fapi_getFsmStatus((uint32_t) address, u32UserFlashConfig);
if (oFlashStatus != 3)
{
    FMSTAT_Fail();
}

oReturnCheck = Fapi_doVerify((uint32_t*) address, VERIFY_LEN, (uint32_t*) write_buffer,
        &oFlashStatusWord, 0, u32UserFlashConfig);

if (oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}
```

6.2.5.2 Erase_Bank

The Erase_Bank function leverages the Flash API to erase a full EEPROM Unit. This function is only a wrapper around Flash API, and the protection masks are set in the EEPROM_Erase() functions.

It begins by clearing the FSM status and copying the protection masks into Flash API.

```
ClearFSMStatus(FLASH_BANK_SELECT, u32UserFlashConfig);

Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTA, WE_Protection_A_Mask);

Fapi_setupBankSectorEnable((uint32_t*) FLASH_BANK_SELECT, u32UserFlashConfig,
        FLASH_NOWRAPPER_O_CMDWEPROTB, WE_Protection_B_Mask);
```

Then, it erases the flash and checks for programming errors.

```
oReturnCheck = Fapi_issueBankEraseCommand((uint32_t*) FLASH_BANK_SELECT, 0, u32UserFlashConfig);

while(Fapi_checkFsmForReady((uint32_t) FLASH_BANK_SELECT, u32UserFlashConfig) ==
Fapi_Status_FsmBusy);

if (oReturnCheck != Fapi_Status_Success)
    Sample_Error();

oFlashStatus = Fapi_getFsmStatus((uint32_t) FLASH_BANK_SELECT, u32UserFlashConfig);
if (oFlashStatus != 3)
{
    FMSTAT_Fail();
}
```

Finally, if Erase_Blank_Check is set, a blank check is performed.

```
if (Erase_Blank_Check)
{
    uint32_t address = FLASH_BANK_SELECT + FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT][0] *
FLASH_SECTOR_SIZE;
    Fapi_FlashStatusWordType oFlashStatusWord;
    oReturnCheck = Fapi_doBlankCheck((uint32_t*) address, BLANK_CHECK_LEN, &oFlashStatusWord, 0,
                u32UserFlashConfig);
    if (oReturnCheck != Fapi_Status_Success)
    {
        Sample_Error();
    }
}
```

6.2.5.3 Configure_Protection_Masks

The Configure_Protection_Masks provides functionality to disable Write/Erase protection for any sector selected for EEPROM Emulation. This is done by calculating the appropriate Masks to pass to the Fapi_setupBankSectorEnable() function. It requires two parameters: a pointer to the selected Flash Sector numbers and the number of sectors used for emulation. For more information on the implementation of the Fapi_setupBankSectorEnable() function, see the *F29H85x Flash API Reference Guide*.

The return value of this function will be used to disable Write/Erase protection in Flash Sectors selected for EEPROM Emulation.

```
uint64_t Protection_Mask_Sectors = 0;

if (Num_EEPROM_Sectors > 1)
{
    uint64_t Unshifted_Sectors;
    uint8_t Shift_Amount;

    if (Sector_Numbers[0] < 32 && Sector_Numbers[1] < 32)
    {
       Unshifted_Sectors = (uint64_t) 1 << Num_EEPROM_Sectors;
       Unshifted_Sectors -= 1;
       Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);
    }
    else if (Sector_Numbers[0] > 31 && Sector_Numbers[1] > 31)
    {
        Shift_Amount = ((Sector_Numbers[1] - 32) / 8) - ((Sector_Numbers[0] - 32) / 8) + 1;
        Unshifted_Sectors = (uint64_t) 1 << Shift_Amount;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << ((Sector_Numbers[0] - 32)/8));
        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;
    }
    else
    {
        Shift_Amount = ((Sector_Numbers[1] - 32)/8) + 1;
        Unshifted_Sectors = (uint64_t) 1 << Shift_Amount;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= Unshifted_Sectors;

        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;

        Unshifted_Sectors = (uint64_t) 1 << ((32 - Sector_Numbers[0]) + 1);
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);
    }
}
else {
    if (Sector_Numbers[0] < 32)
    {
        Protection_Mask_Sectors |= ((uint64_t) 1 << Sector_Numbers[0]);
    }
    else
    {
        Protection_Mask_Sectors |= ((uint64_t) 1 << ((Sector_Numbers[0] - 32)/8));
        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;
    }
}

return Protection_Mask_Sectors;
```

6.2.5.4 Set_Protection_Masks

Set_Protection_Masks() is a simple wrapper around Configure_Protection_Masks() that updates the global mask variables (WE_Protection_A_Mask and WE_Protection_B_Mask) with the correct values for the current active unit.

```
uint64_t WE_Protection_AB_Mask = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR,
                                                            NUM_EEPROM_SECTORS);

WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32_t)WE_Protection_AB_Mask;
WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;
```

6.2.5.5 Fill_Buffer

Fill_Buffer() is a very basic helper function to fill up the various status and write buffers. It takes a target buffer, its length, and a value as input and fills the buffer with that value.

```
uint8_t i;
for (i = 0; i < buffer_len; i++)
{
    buffer[i] = value;
}
```

6.2.5.6 ClearFSMStatus

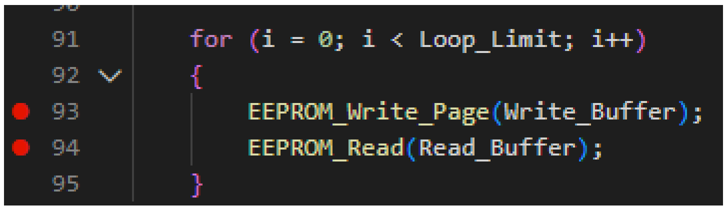
The ClearFSMStatus() function is responsible for clearing the status of the previous flash operation.

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_StatusType  oReturnCheck;

while (Fapi_checkFsmForReady(u32StartAddress, u32UserFlashConfig) != Fapi_Status_FsmReady){}
oFlashStatus = Fapi_getFsmStatus(u32StartAddress, u32UserFlashConfig);


oReturnCheck = Fapi_issueAsyncCommand(u32StartAddress, u32UserFlashConfig, Fapi_ClearStatus);
while (Fapi_getFsmStatus(u32StartAddress, u32UserFlashConfig) != 0);

if(oReturnCheck != Fapi_Status_Success)
{
    Sample_Error();
}
```

6.3 Testing Example

The examples provided were tested with F29H859TU8. To properly test the example, the memory window and breakpoints need to be utilized within Code Composer Studio. The following steps were followed to program and test the project.

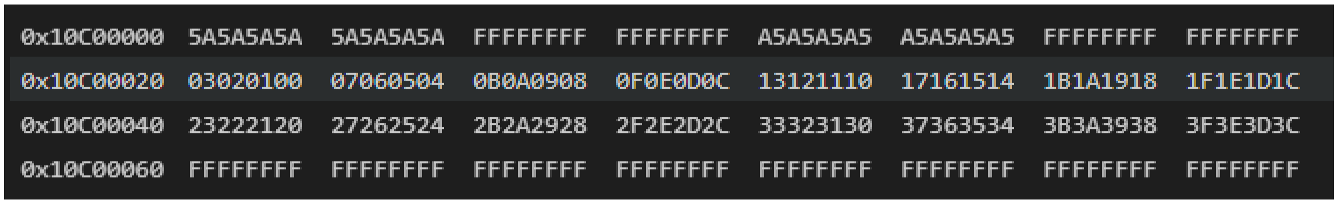
1. Connect the F29H859TU8 to the PC via USB and an XDS110 Debug Probe with JTAG connection.
2. Connect a 5V DC power supply to the board.
3. Start Code Composer Studio and open the F29H85x_EEPROM_PingPong_Example.pjt.
4. Build the project by selecting Project -> Build Project.
5. Launch the project by right-clicking it in the explorer, then selecting "Debug Project".
6. Set breakpoints to properly view data written to and read from the memory within the memory window as shown in Break Points.



```
91    for (i = 0; i < Loop_Limit; i++)
92    {
93        EEPROM_Write_Page(Write_Buffer);
94        EEPROM_Read(Read_Buffer);
95    }
```

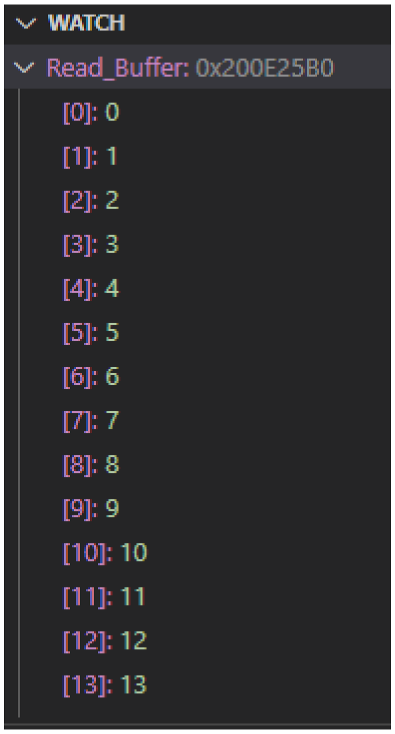
Figure 6-2. Break Points

7. Run to the first break-point and open the Memory Browser (View -> Memory Browser) to view the data. Bank_Pointer can be used to watch the data written and Read_Buffer to watch the data being read back from the memory. This is shown in Write to EEPROM Unit and Read Data.



```
0x10C00000  5A5A5A5A  5A5A5A5A  FFFFFFFF  FFFFFFFF  A5A5A5A5  A5A5A5A5  FFFFFFFF  FFFFFFFF
0x10C00020  03020100  07060504  0B0A0908  0F0E0D0C  13121110  17161514  1B1A1918  1F1E1D1C
0x10C00040  23222120  27262524  2B2A2928  2F2E2D2C  33323130  37363534  3B3A3938  3F3E3D3C
0x10C00060  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

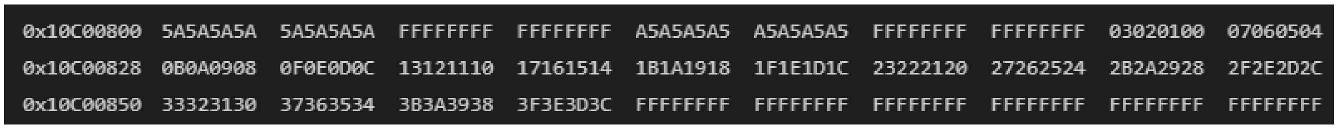
Figure 6-3. Write to EEPROM Unit



```
WATCH
Read_Buffer: 0x200E25B0
    [0]: 0
    [1]: 1
    [2]: 2
    [3]: 3
    [4]: 4
    [5]: 5
    [6]: 6
    [7]: 7
    [8]: 8
    [9]: 9
    [10]: 10
    [11]: 11
    [12]: 12
    [13]: 13
```

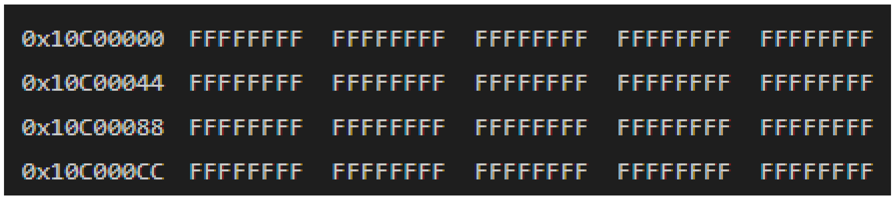
Figure 6-4. Read Data

8. Continue running from break-point to break-point until the program has finished or EEPROM is full.
9. Once EEPROM is full, you will see the new data written to the previously inactive unit and the full EEEPROM will be erased. Shown in Write to new EEPROM Unit and Erase full EEPROM Unit.



```
0x10C00800  5A5A5A5A  5A5A5A5A  FFFFFFFF  FFFFFFFF  A5A5A5A5  A5A5A5A5  FFFFFFFF  FFFFFFFF  03020100  07060504
0x10C00828  0B0A0908  0F0E0D0C  13121110  17161514  1B1A1918  1F1E1D1C  23222120  27262524  2B2A2928  2F2E2D2C
0x10C00850  33323130  37363534  3B3A3938  3F3E3D3C  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

Figure 6-5. Write to New EEPROM Unit



```
0x10C00000  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
0x10C00044  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
0x10C00088  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
0x10C000CC  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

Figure 6-6. Erase Full EEPROM Unit

10. This process can be repeated between the two EEPROM units as necessary.

The preceding steps were used to test the Page Mode configuration. The 64-Bit Mode configuration can also be tested with the same procedure. To enable 64-Bit Mode, change the definition in the EEPROM_PingPong_Config.h file by un-commenting the _64_BIT_MODE directive and commenting out the PAGE_MODE directive.

7 Application Integration

Applications needing this functionality need to include the EEPROM_Config.h and EEPROM.c files provided for the device. The Flash API and driverlib also needs to be included for the appropriate device. For example, for Single-Unit emulation on F29H85x, the following files are needed:

- F29H85x_EEPROM.c
- EEPROM_Config.h
- Device.c and device.h
- flash_programming_F29H85x.h
- F29H85x_NWFlashAPI_v21.00.lib
- driverlib.lib

Note

The Flash API is updated periodically with new revisions of silicon being released. To ensure functionality, the latest Flash API libraries should be used.

8 Flash API

The Flash API is resident and called for by the CPU for various Flash operations. The API library includes functions to erase, program, and verify the Flash array. The smallest amount of memory that can be erased at a time is a single sector. The program function can only change bits from a 1 to a 0 (assuming the corresponding ECC bits have not been written yet). Bits cannot be moved from a 0 back to a 1 by the programming function. The programming function operates on a single byte at a time, but 64-bits must be written every time to align with ECC requirements.

8.1 Flash API Checklist

This following section is taken from the *F29H85x Flash API Reference Guide* and describes the flow for using various API functions.

- After the device is first powered up, the Fapi_initializeAPI() function must be called before any other API function (except for the Fapi_getLibraryInfo() function) can be used. This procedure configures the Flash Wrapper based on the user specified operating system frequency.
- Before performing a Flash operation for the first time, the Fapi_setActiveFlashBank() function must be called.
- If the system operating frequency is changed after the initial call to the Fapi_initializeAPI() function, this function must be called once again before any other API function (except the Fapi_getLibraryInfo() function) can be used. This procedure updates the API internal state variables.

8.1.1 Flash API Do's and Do Not's

API Do's

- Execute the Flash API code from RAM or a Flash Bank not selected for EEPROM Emulation (some functions must be run from RAM).
- Configure the API for the correct CPU frequency of operation
- Follow the Flash API checklist to integrate the API into an application
- Configure the PLL as needed and pass the configured CPUCLK value to Fapi_initializeAPI() function.
- Configure waitstates as per the device-specific data manual before calling Flash API functions. The Flash API issues and errors if the waitstate configured by the application is not appropriate for the operating frequency of the application.
- Carefully review the API restrictions described in the *F29H85x Flash API Reference Guide*.

API Do Not's

- Do not execute Flash API from the same Flash Bank that is selected for emulation
- Do not configure interrupt service routines (ISRs) that result in read/fetch access from the Flash bank on which an erase/program operation is in progress. Flash API functions, user application functions that call Flash API functions, and any ISRs, must be executed from RAM or the flash bank on which there is no active erase/program operation in progress.
- Do not access the Flash bank on which the Flash erase/program operation is in progress

9 Source File Listing

File	Function	Description

F29H85x_EEPROM_PingPong.c	EEPROM_Config_Check()
	Configure_Protection_Masks()
	EEPROM_Write_Page()
	EEPROM_Read()
	EEPROM_Erase()
	Erase_Bank()
	EEPROM_GetValidBank()
	EEPROM_UpdateBankStatus()
	EEPROM_UpdatePageStatus()
	EEPROM_UpdatePageData()
	EEPROM_64_Bit_Mode_Check_EOS()
	EEPROM_Write_64_Bits()
	EEPROM_CheckStatus()
	ClearFSMStatus()
		Validate EEPROM configuration
		Configure bits for W/E Protection Masks
		Performs write operation
		Performs read operation
		Performs erase operation
		Performs erase operation
		Finds valid bank and page
		Updates bank status
		Updates pages status
		Updates page data
		Finds pointer for 64-bit operation and tests for full sector
		Programs 64-bits to flash
		Verify success of flash operation
		Clear flash state machine status

EEPROM_PingPong_Config.h		Contains function prototypes, global variables, includes flash API headers, pointer initialization, definition of constants and macros, enter user-configurable variables

F29H85x_EEPROM.c	EEPROM_Config_Check()
	Configure_Protection_Masks()
	EEPROM_Write_Page()
	EEPROM_Read()
	EEPROM_Erase()
	Erase_Bank()EEPROM_GetValidBank()
	EEPROM_UpdateBankStatus()
	EEPROM_UpdatePageStatus()
	EEPROM_UpdatePageData()
	EEPROM_64_Bit_Mode_Check_EOS()
	EEPROM__64_Bits()
	EEPROM_CheckStatus()
	ClearFSMStatus()
		Validate EEPROM configuration
		Configure bits for W/E Protection Masks
		Performs write operation
		Performs read operation
		Performs erase operation
		Finds valid bank and page
		Updates bank status
		Updates pages status
		Updates page data
		Finds pointer for 64-bit operation and tests for full sector
		Programs 64-bits to flash
		Verify success of flash operation
		Clear flash state machine status

EEPROM_Config.h		Contains function prototypes, global variables, includes flash API headers, pointer initialization, definition of constants and macros, enter user-configurable variables

10 Troubleshooting

Below are solutions to some common issues encountered by users when utilizing the EEPROM and EEPROM_PingPong projects.

10.1 General

Question: I cannot find the EEPROM and EEPROM_PingPong projects, where are they?:

Device	Build Configurations	Location
F29H85x	RAM, FLASH	f29h85x-sdk > examples > driverlib > single_core > flash

Question: What are the first things I should check if the EEPROM project encounters an error?

Answer:

- View the configuration file (EEPROM_Config.h, EEPROM_PingPong_Config.h) and check the provided options for the following: programming mode (64 bit vs. Page), number of EEPROM banks, number of EEPROM pages, and the data size of EEPROM pages. Also, check the main program file (EEPROM_Example.c, EEPROM_PingPong_Example.c) to see if the correct Flash Sector locations are being used for EEPROM Emulation. If an incorrect first and last sector value are provided, an error will occur and be seen in the EEPROM_Config_Check function. The EEPROM_Config_Check function will provide general information for error checking.
- Ensure that the protection masks are enabled or disabled for the appropriate sector(s) selected for EEPROM Emulation for your device. For more information, see the device's Flash API reference guide.
- One area of the program to check would be the linker command file - make sure all flash sections are aligned to 128-bit boundaries. In SECTIONS, add a comma and "ALIGN(8)" after each line where a section is allocated to flash.

11 Conclusion

This application report has proven that the F29H85x Real-Time Controller is capable of utilizing its internal Flash to emulate EEPROM. This allows for in-system storage and reduces the need for an external component. This is highly dependent on code size and whether or not an extra Flash sector is available for use. This document also provides designers with a ready-made driver using the Flash API library that accelerates and simplifies design.

12 References

- Texas Instruments: *F29H85x Flash API Reference Guide*
- Texas Instruments: *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual*