Application Note

EEPROM Emulation for Generation 3 C2000 Real-Time Controllers

Vamsikrishna Gudivada, Skyler Baumer, Charles Roberson

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 report 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 report can be found in C2000Ware v5.02.00.00 (or higher) at the following path(s): C2000Ware_5_02_00_00/driverlib/f28p65x/examples/c28x/flash/ , C2000Ware_5_02_00_00/driverlib/f28003x/examples/flash/

Table of Contents

1 Introduction....2
2 Difference Between EEPROM and On-Chip Flash....2
3 Overview....3
3.1 Basic Concept....3
3.2 Single-Unit Method....3
3.3 Ping-Pong Method....4
3.4 Creating EEPROM Sections (Pages) and Page Identification....4
4 Software Description....6
4.1 Software Functionality and Flow....6
5 Ping-Pong Emulation....8
5.1 User-Configuration....8
5.2 EEPROM Functions....9
5.3 Testing Example....26
6 Single-Unit Emulation....29
6.1 User-Configuration....29
6.2 EEPROM Functions....30
6.3 Testing Example....45
7 Application Integration....47
8 Adapting to Other Gen 3 C2000 MCUs....47
9 Flash API....48
9.1 Flash API Checklist....48
10 Source File Listing....49
11 Troubleshooting....50
11.1 General....50
12 Conclusion....50
13 References....50
14 Revision History....51

List of Figures

Figure 3-1. Single-Unit Behavior....3
Figure 3-2. Ping Pong Behavior....4

Figure 3-3. Bank Partitioning........ 5
Figure 3-4. Page Layout........ 6
Figure 4-1. Software Flow........ 7
Figure 5-1. GetValidBank Flow........ 17
Figure 5-2. Break Points........ 26
Figure 5-3. Write to EEPROM Unit........ 27
Figure 5-4. Read Data........ 27
Figure 5-5. Write to new EEPROM Unit........ 27
Figure 5-6. Erase full EEPROM Unit........ 28
Figure 5-7. Write to original EEPROM unit........ 28
Figure 5-8. Erase full EEPROM Unit........ 28
Figure 6-1. GetValidBank Flow........ 37
Figure 6-2. Break Points........ 45
Figure 6-3. Write to EEPROM........ 46
Figure 6-4. Read Data........ 46
Figure 6-5. Erase full EEPROM Unit........ 46
Figure 6-6. Write to EEPROM........ 47

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

Generation 3 C2000 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 adding a traditional EEPROM on the chip. Some designers use an external EEPROM part for such non-volatile storage. The good news is that Flash memory is a specific type of EEPROM and all Generation 3 C2000 MCUs have in-circuit programming ability for the Flash memory. This application report makes use of this facility and allows using sectors of on-chip Flash as EEPROM by emulating the EEPROM functionality within the limitations of the Flash memory. Note that at least one Flash sector is entirely used as an emulated EEPROM; therefore, it is not available for the application code.

Note

Generation 3 C2000 MCUs include: TMS320F2837x, TMS320F2838x, TMS320F28004x, TMS320F28002x, TMS320F28003x, TMS320F280013x, TMS320F280015x, TMS320F28P65x.

2 Difference 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 major difference between EEPROM and Flash operations is seen in the erase operation. The EEPROM does not require a sector erase operation. One can erase a particular byte requiring the specified time. However, the smallest unit of an erase operation in Flash is one sector.

Flash erase and write cycles are performed by applying time-controlled voltages to each cell. In the erase condition, each cell (bit) reads logical 1. Therefore, every Flash location of a C2000 Real-Time Controller reads 0xFFFF when erased. Through programming, the cell can be changed to logical 0. Any word can be overwritten to change a bit from logical 1 to 0 (assuming corresponding ECC has not been programmed); but not the other way around. The on-chip Flash memory on Generation 3 C2000 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 of the device-specific data manual.

3 Overview

The implementation described in this document supports both Single-Unit and Ping-Pong EEPROM Emulation. Single-Unit and Ping-Pong implementations both support two modes, Page-Programming and 64-Bit Programming mode. Ping-Pong Emulation will be described first, and Single-Unit will be described subsequently.

There are multiple user-configurable EEPROM variables supported by this implementation. These variables are detailed in Section 5.1.

3.1 Basic Concept

In this implementation, the emulated EEPROM is comprised of at least one Flash Sector. Due to the block erase requirement of Flash, a Flash sector has to be entirely reserved for the EEPROM Emulation. Based on the C2000 part number, the size of the Flash sector will vary. The area of the Flash sector is divided into a number of smaller sections and is referred to as a Page. For example, a 2K x 16 flash sector can be divided into 32 pages, each with a size of 64 x 16.

The data to be saved is first written in a buffer in RAM. Then, using the in-circuit programming facility of Generation 3 C2000 MCUs, the data is written to the first page in the selected sector(s) for EEPROM Emulation. The next time data is written to Flash, it will be written to the next page. This process continues until the last page in the selected sector is written. Upon reaching the last page, there are two ways to continue. If using Single-Unit EEPROM emulation, see the Single-Unit Emulation behavior to see how this is handled. If using Ping Pong EEPROM Emulation, see Ping-Pong Method to see how this is handled.

In addition to the Page Programming concept described above, there is also support for 64-Bit Programming. In this mode, the sector(s) is not broken into EEPROM banks and pages. 64-Bit Programming is discussed further in Section 5.2.10 and Section 5.2.11 .

3.2 Single-Unit Method

If using Single-Unit EEPROM Emulation, Figure 3-1 shows the implemented behavior.

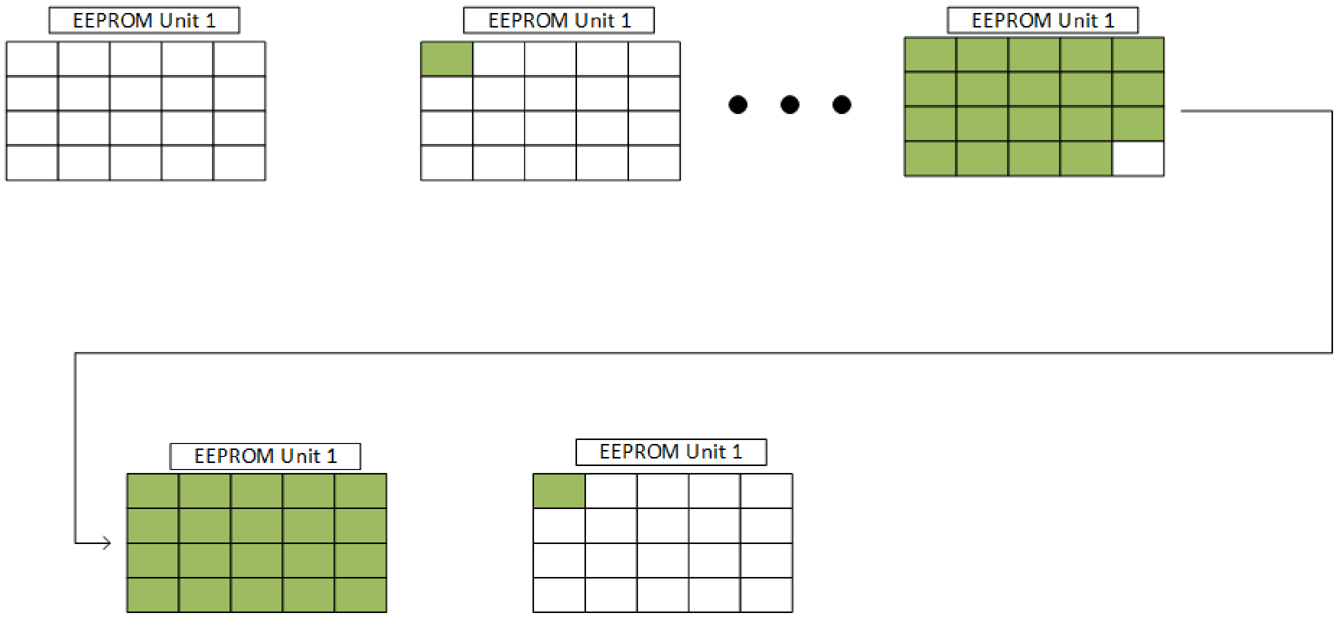


Figure 3-1. Single-Unit Behavior

If the EEPROM unit is full and there is more data to be written, the EEPROM unit is erased and the new data is programmed to Flash. This process can be repeated as necessary.

3.3 Ping-Pong Method

If using the Ping Pong method, the following diagram shows the implemented behavior. As shown in Figure 3-2, there are two EEPROM units made up of selected Flash Sectors. One is marked as an Active Unit, and the other is marked as the Inactive Unit. To begin, data is written to the Active Unit.

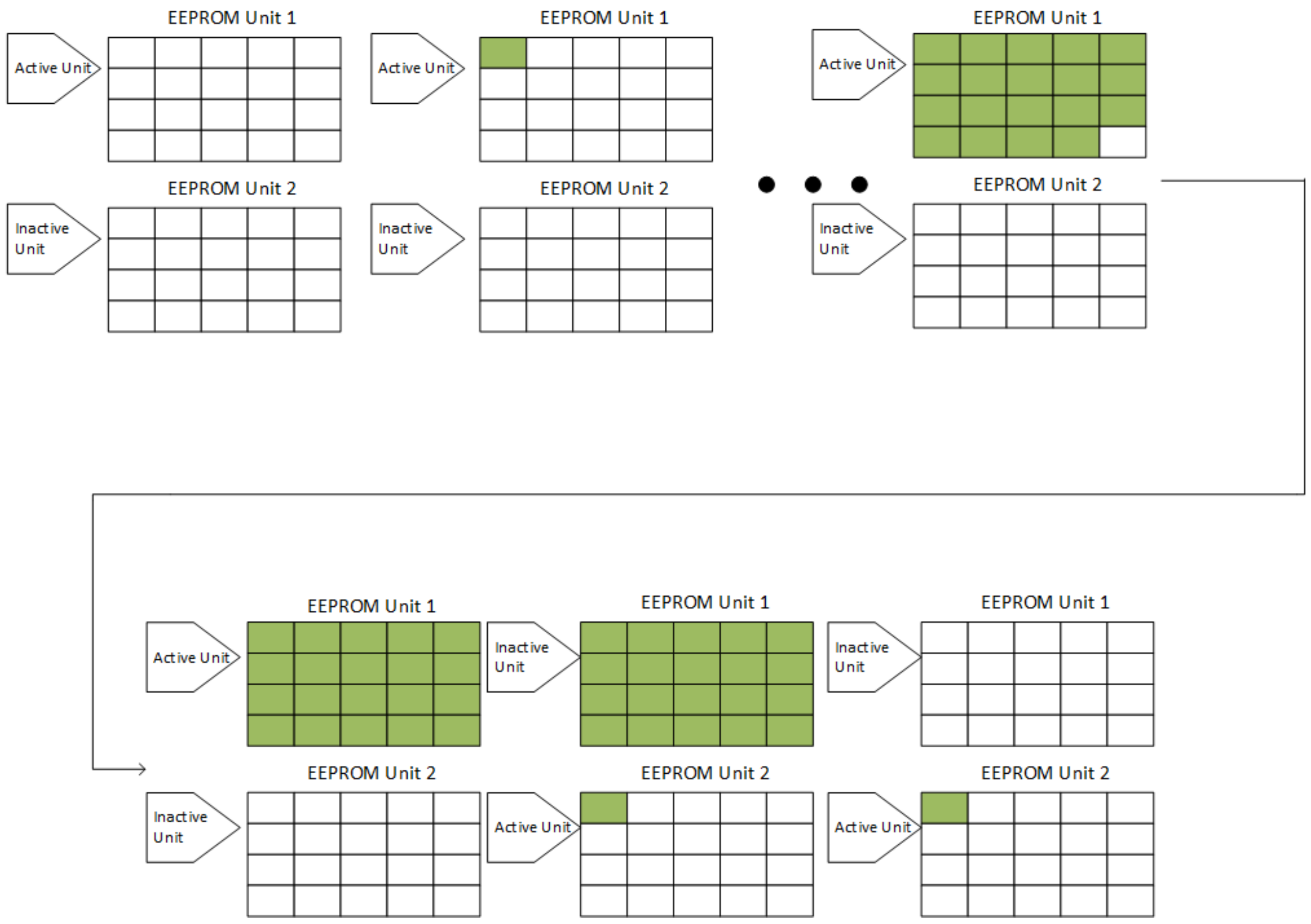


Figure 3-2. Ping Pong Behavior

If the Active Unit is full and there is more data to be written, the Active and Inactive EEPROM units will switch. Therefore, the previously Active Unit (full Unit) will be marked as Inactive, and the previously Inactive Unit (empty Unit) will be marked as Active. Subsequently, the data will be written to the newly Active EEPROM Unit. After the data is successfully programmed to the Active EEPROM Unit, the Inactive EEPROM Unit is erased. This method ensures that there is a fall-back options for the last successfully written data in case of any failure during the erase or program operation when the currently active EEPROM Unit is full.

This process can be repeated as necessary.

3.4 Creating EEPROM Sections (Pages) and Page Identification

In order to support EEPROM emulation with varying data sizes (other 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. First, the Flash sector (or set of chosen Flash Sectors) is divided into EEPROM banks. Each EEPROM bank is further divided into Pages. This partitioning is shown in Bank Partitioning. The partitioning of EEPROM Banks and Pages is the same for Single-Unit and Ping-Pong emulation.

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

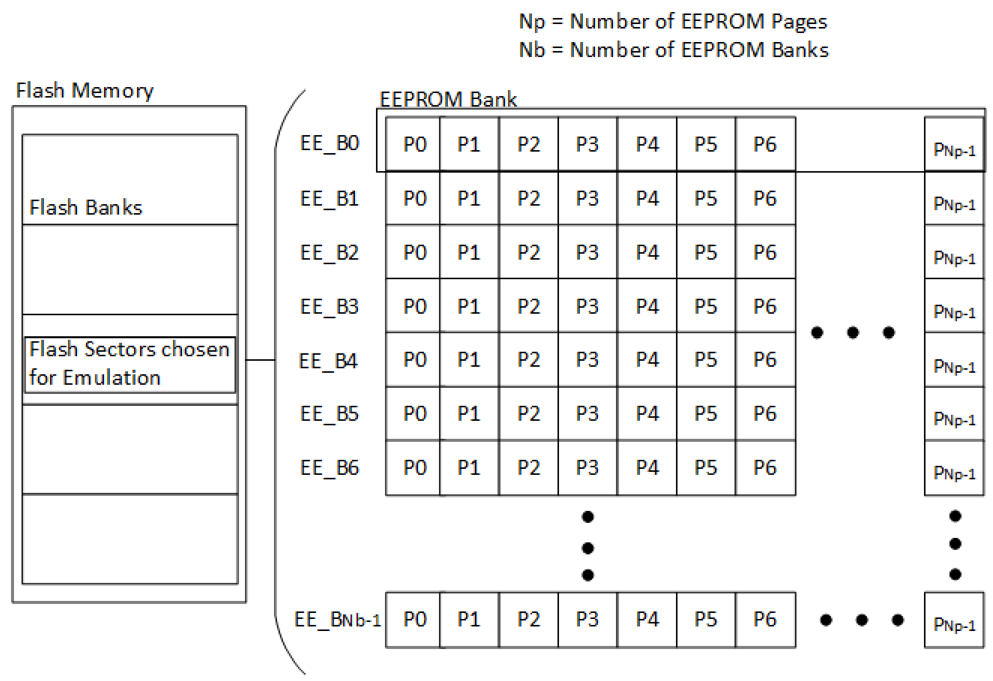


Figure 3-3. Bank Partitioning

The first eight 16-bit words of each EEPROM bank are reserved for EEPROM bank status information and the first eight words of each page are reserved for page status information. Every time a new set of data is written to a page, the status location of the last page and the next page are modified. When a new EEPROM bank is used the Bank Status of the last and current EEPROM banks are updated. Both the EEPROM Bank and Page status words differentiate between the current EEPROM Bank/Page and a used EEPROM Bank/Page in the same way. 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 latter 64-bits are written with the appropriate status code. More details about the status codes can be found in the EEPROM_GetValidBank.

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. Only Pages 0 and 1 are shown, but it should be noted that Page 2 through Page (N-1) are identical to Page 1.

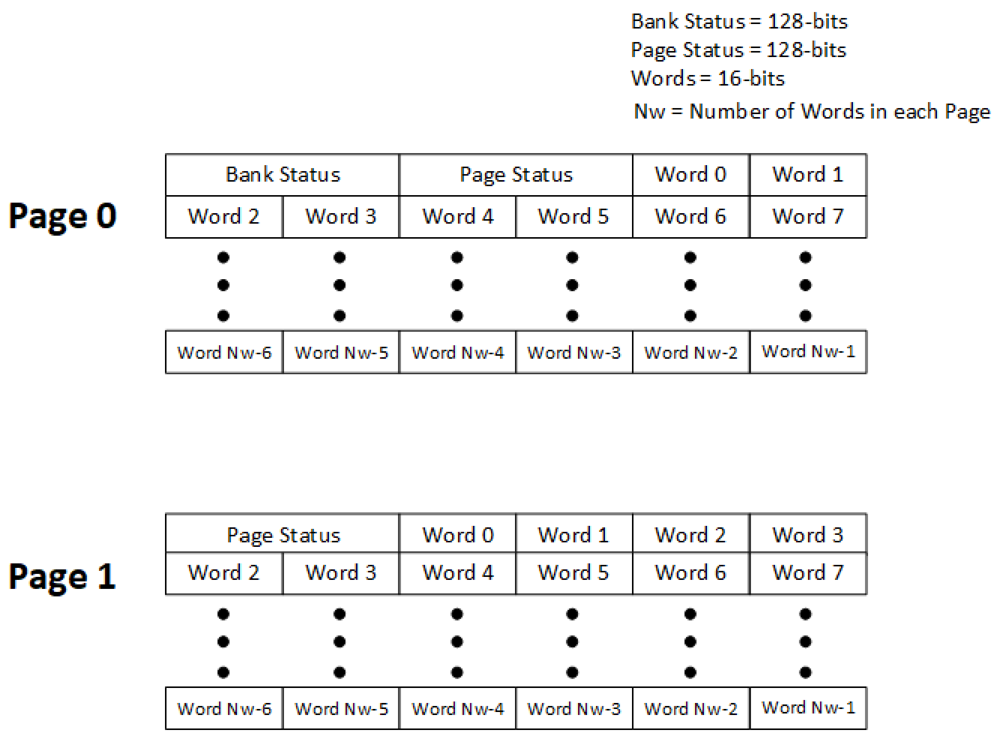


Figure 3-4. Page Layout

4 Software Description

The software provided with this application report includes EEPROM Emulation source code for the F28P65x Generation 3 C2000 Real-Time Controller with an example project demonstrating how to utilize the source code. Some aspects of the code may differ based on the device being used. For an example, see the Adapting to Other Gen 3 C2000 MCUs. The rest of the guide is designed for the F28P650DK9 devices, with highlighted comparisons to the F280039C devices where applicable.

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 F28P65x. The example code can be found within the C2000Ware Directory. The full path is: C2000Ware_5_02_00_00/driverlib/f28p65x/examples/c28x/flash

For comparisons to the F28003x devices, the example code can be found within the C2000Ware Directory. The full path is: C2000Ware_5_02_00_00/driverlib/f28003x/examples/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 the functions 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 F28P65x 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 5.2.1.

At this point, programming can begin. First, data needs to be captured to program. After programming this data, the read functionality reads the last set of data that was programmed into the Flash. This software flow should be followed by most applications, especially the initialization portion 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 their appropriate section in the document.

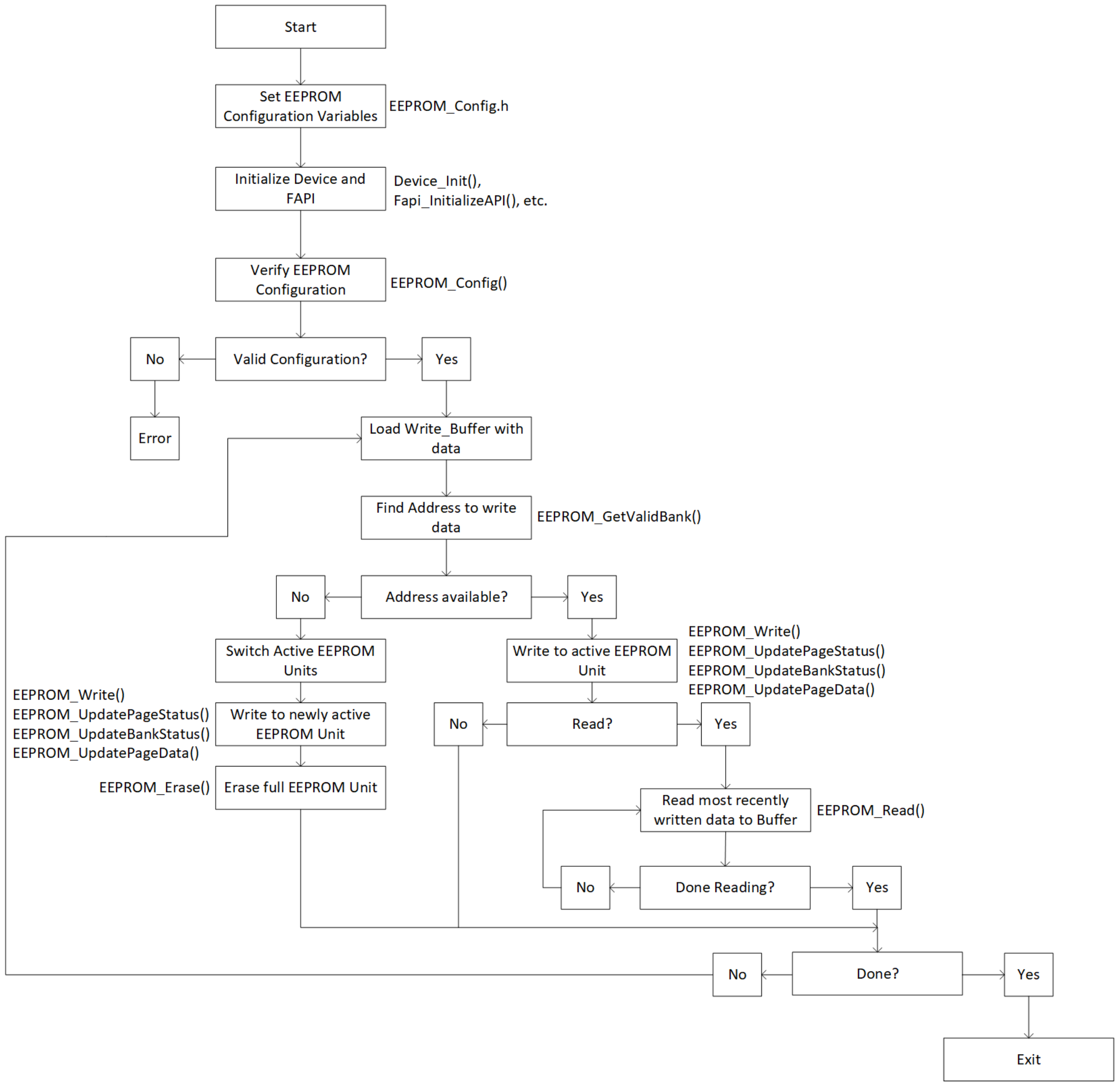

Start
Set EEPROM Configuration Variables
EEPROM_Config.h
Initialize Device and FAPI
Device_Init(), Fapi_InitializeAPI(), etc.
Verify EEPROM Configuration
EEPROM_Config()
No
Valid Configuration?
Yes
Error
Load Write_Buffer with data
Find Address to write data
EEPROM_GetValidBank()
No
Address available?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

Figure 4-1. Software Flow

5 Ping-Pong Emulation

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

5.1 User-Configuration

The implementation detailed in this document allows the user to configure several variables for EEPROM Emulation. These variables are mainly found within EEPROM_PingPong_Config.h, but two are contained in F28P65x_EEPROM_PingPong.c. The configurable variables will be discussed below in the section dedicated to the appropriate file.

5.1.1 EEPROM_PingPong_Config.h

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

- Define which device variant is being used. This allows EEPROM emulation in Flash Banks not common to all devices.

```
// Un-comment appropriate definition if one of the following variants is being used
#define F28P65xDKx 1
//#define F28P65xSKx 1
//#define F28P65xSHx 1
```

- Choose between Page Mode and 64-Bit Mode.

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

- Choose which Flash Bank to use for emulation. **The Flash API and program are stored/run from Flash Bank 0 by default, so it cannot be used for EEPROM Emulation. In general, the Flash API and program should be stored/run from a different Flash bank than the ones used for EEPROM emulation.**

```
#define FLASH_BANK_SELECT FlashBank1StartAddress
```

- Define the Flash Sector size (unit is 16-bit words). This will vary based on the device being used, reference the appropriate data sheet for details.

```
#define FLASH_SECTOR_SIZE F28P65x_FLASH_SECTOR_SIZE
```

- Define how many Flash sectors are in a Flash Bank. This will vary based on the device being used, reference the appropriate data sheet for details.

```
#define NUM_FLASH_SECTORS F28P65x_NUM_FLASH_SECTORS
```

- 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 3
```

- Choose the size of the data space contained within each EEPROM Page (unit is 16-bit words). Although any size can be specified, the size will be adjusted to the closest multiple of four that is greater than or equal to the size specified. For example, a specified size of six 16-bit words per page will be programmed as eight 16-bit words per page, with the last two being treated as 0xFFFF. This is to comply with Flash requirements (8-bit ECC is programmed for every 64-bit aligned Flash memory address).

```
#define DATA_SIZE 64
```

5.1.2 F28P65x_EEPROM_PingPong.c

Within F28P65x_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}. 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 F28P65x 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 *F28P65x Flash API Reference Guide*.

More details about invalid or dangerous configurations can be found in Section 5.2.1.

```
uint16 FIRST_AND_LAST_SECTOR[2][2] = {{1,1},{39,39}};
```

Additionally, you can choose which set of Flash Sectors to begin emulation in.

```
uint16 EEPROM_ACTIVE_UNIT = 0;
```

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.

5.2 EEPROM Functions

To implement this functionality, 12 functions are required to configure, program, read, and erase in Page programming. Two additional functions are needed for 64-bit programming. All functions are included in the F28P65x_EEPROM.c or F28P65x_EEPROM_PingPong.c file.

- EEPROM_Config_Check()
- Configure_Protection_Masks(Uint16* Sector_Numbers, Uint16 Num_EEPROM_Sectors)
- EEPROM_Write(Uint16* Write_Buffer)
- EEPROM_Read(Uint16* Read_Buffer)
- EEPROM_Erase( )
- Erase_Bank( )
- EEPROM_GetValidBank(Uint16 Read_Flag)
- EEPROM_UpdateBankStatus( )
- EEPROM_UpdatePageStatus( )
- EEPROM_UpdatePageData(Uint16* Write_Buffer)
- EEPROM_Get_64_Bit_Data_Address()
- EEPROM_Program_64_Bits(Uint16 Num_Words)
- EEPROM_CheckStatus(Fapi_StatusType* oReturnCheck);
- ClearFSMStatus()

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

5.2.1 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. A valid Flash Bank selection must not select Bank 0 for emulation and must be supported by the specific device variant. For example, only some F28p65x variants have Flash Banks 2-4, while others do not. To verify this information, see the device-specific data sheet.

```
if (FLASH_BANK_SELECT == FlashBank0StartAddress)
{
    return 0xFFFF;
}

if (FLASH_BANK_SELECT == FlashBank2StartAddress)
{
#if !defined(F28P65xDKx) && !defined(F28P65xSKx) && !defined(F28P65xSHx)
        return 0xFFFF;
#endif
} else if (FLASH_BANK_SELECT == FlashBank3StartAddress) // If using Bank 3
{
#if !defined(F28P65xDKx) && !defined(F28P65xSKx)
        return 0xFFFF;
#endif
} else if (FLASH_BANK_SELECT == FlashBank4StartAddress)
{
#if !defined(F28P65xDKx) && !defined(F28P65xSKx) && !defined(F28P65xSHx)
        return 0xFFFF;
#endif
}
```

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

```
uint16 NUM_EEPROM_SECTORS_1 = FIRST_AND_LAST_SECTOR[0][1] - FIRST_AND_LAST_SECTOR[0][0] + 1;
uint16 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)
{
    // Check if FIRST_AND_LAST_SECTOR is sorted in increasing order
    // and doesn't have duplicates
    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;
    }

    // Check if FIRST_AND_LAST_SECTOR contains invalid sector
    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 only using 1 sector
{

    // Verify that only sector is valid
    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, the following will also be checked for

- Check if total size of EEPROM Banks + Pages will fit in the Flash Sectors selected.

```
// Calculate size of each EEPROM Bank (16 bit words)
Bank_Size = 8 + ((EEPROM_PAGE_DATA_SIZE + 8) * NUM_EEPROM_PAGES);

// Calculate amount of available space (16 bit words)
uint32 Available_Words = NUM_EEPROM_SECTORS * FLASH_SECTOR_SIZE;

// Check if size of EEPROM Banks and Pages will fit in EEPROM sectors
if (Bank_Size * NUM_EEPROM_BANKS > Available_Words)
{
    return 0xCCCC;
}
```

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

```
// Verify that the two EEPROM units do not have overlapping protection
// masks
// First, get sectors for both units
uint64 WE_Protection_AB_Sectors_Unit_0 = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR[0],
NUM_EEPROM_SECTORS);
uint64 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;
}
```

It also warns you with the appropriate return code if one of the following conditions is detected:

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

```
// Notify for extra space (more than one EEPROM bank leftover)
if (Available_Words - (Bank_Size * NUM_EEPROM_BANKS ) >= Bank_Size)
{
    Warning_Flags += 1;
}
```

- If each page consists of less than 5 16-bit words (this wastes space as the 64-Bit Mode could be used without the need for Status Codes)

```
if (EEPROM_PAGE_DATA_SIZE < 5)
{
    Warning_Flags += 2;
}
```

If using sectors in the 32-127 range (for F28P65x 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 *TMS320F28P65x Flash API Version 3.02.00.00 Reference Guide*.

```
uint16 i;
for (i = 0; i < 2; i++)
{
    // If using any sectors from 32-127
    if (FIRST_AND_LAST_SECTOR[i][1] > 31) {

        // If all sectors use protection mask B
        if (FIRST_AND_LAST_SECTOR[i][0] > 31)
        {
            // If using less than 8 sectors
            if (NUM_EEPROM_SECTORS < 8)
            {

                Warning_Flags += 4;
                break;

            } else {
                // If sectors are multiples of 8
                if ((FIRST_AND_LAST_SECTOR[i][0] % 8) != 0 ||
                       ((FIRST_AND_LAST_SECTOR[i][1] + 1) % 8 != 0))
                {
                    Warning_Flags += 4;
                    break;
            }
            }
        } else { // If only last sector is using protection mask B

            // If not a multiple of 8
            if ((FIRST_AND_LAST_SECTOR[i][1] + 1) % 8 != 0) {

                Warning_Flags += 4;
                break;
            }
        }
    }
}
```

This function also prepares Flash for Emulation by erasing the Sectors to be used for programming.

```
    // Combine sectors from both units and separate them by which
    // protection register they use (A or B)
uint32 Combined_WE_Protection_A_Sectors =
                               (uint32)WE_Protection_AB_Sectors_Unit_0 |
                                 (uint32)WE_Protection_AB_Sectors_Unit_1;
uint32 Combined_WE_Protection_B_Sectors =
                                 WE_Protection_AB_Sectors_Unit_0 >> 32 |
                                   WE_Protection_AB_Sectors_Unit_1 >> 32;

// Create protection masks accordingly
WE_Protection_A_Mask = 0xFFFFFFFF ^ Combined_WE_Protection_A_Sectors;
WE_Protection_B_Mask = 0x00000FFF ^ Combined_WE_Protection_B_Sectors;

Erase_Bank();
```

Finally, Write/Erase Protection masks are configured for the Active EEPROM Unit.

```
// Configure Write/Erase Protection Masks used by the Flash API
uint64 WE_Protection_AB_Mask =
Configure_Protection_Masks(FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT], NUM_EEPROM_SECTORS);


WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32)WE_Protection_AB_Mask;
WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;
```

5.2.2 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 details on the implementation of the Fapi_setupBankSectorEnable function, see the *TMS320F28P65x Flash API Version 3.02.00.00 Reference Guide*.

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

```
// Initialize a variable to store the bits indicating which sectors
// need to have write/erase protection disabled.
// The first lower 32 bits will represent CMDWEPROTA and the upper 32
// bits will represent CMDWEPROTB.
uint64 Protection_Mask_Sectors = 0;

// If we have more than one Flash Sector
if (Num_EEPROM_Sectors > 1)
{

    uint64 Unshifted_Sectors;
    uint16 Shift_Amount;

    // If all sectors use Mask A
    if (Sector_Numbers[0] < 32 && Sector_Numbers[1] < 32)
    {

        // Configure Mask A
        Unshifted_Sectors = (uint64) 1 << Num_EEPROM_Sectors;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);

    }// If all sectors use Mask B
    else if (Sector_Numbers[0] > 31 && Sector_Numbers[1] > 31)
    {

        // Configure Mask B
        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;

    } else // If both Masks A and B need to be configured
    {

        // 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;
        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]);

    }


} else { // If only using 1 Flash Sector

    if(Sector_Numbers[0] < 32)
    {
        Protection_Mask_Sectors |= ((uint64) 1 << Sector_Numbers[0]);
    } else
    {
        Protection_Mask_Sectors |= ((uint64) 1 << ((Sector_Numbers[0] - 32)/8));
        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;
    }

}
```

```
return Protection_Mask_Sectors;
```

For comparison, the F28003x EEPROM Ping Pong example project's Configure_Protection_Masks functionality differs from that of the F28P65x EEPROM PingPong project example with the amount of sectors available for protection. Each bit in the Write/Erase protection mask represents it's own sector.

```
// Initialize a variable to store the bits indicating which sectors need to have write/erase
    // protection disabled.
    uint16 Protection_Mask_Sectors = 0;
    uint16 Unshifted_Sectors;

    // If we have more than one Flash Sector
    if (Num_EEPROM_Sectors > 1)
    {
        // Configure mask
        Unshifted_Sectors = (uint16) 1 << Num_EEPROM_Sectors;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);

    } else { // If only using 1 Flash Sector

        if(Sector_Numbers[0] < 16)
        {
            Unshifted_Sectors = (uint16) 1 << Sector_Numbers[0];
            Protection_Mask_Sectors |= Unshifted_Sectors;
        }

    }

    return Protection_Mask_Sectors;
```

5.2.3 EEPROM_Write

The EEPROM_Write() 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();     // Find Current Bank and Current Page

EEPROM_UpdatePageStatus(); // Update Page Status of previous page
EEPROM_UpdateBankStatus(); // Update Bank Status of current and previous bank
EEPROM_UpdatePageData();   // Update Page Data of current page
```

5.2.4 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.

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

```
uint16 i;

// Check for empty EEPROM
if (Empty_EEPROM)
{
    Sample_Error(); // Attempting to read data that hasn't been written
} else
{
    // Find Current Bank and Current Page
    EEPROM_GetValidBank(1);

    // Increment page pointer to point at first data word
    Page_Pointer += 8;

    // Transfer contents of Current Page to Read Buffer
    for(i=0;i<DATA_SIZE;i++)
    {
        Read_Buffer[i] = *(Page_Pointer++);
    }
    }
```

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

```
uint16 i;

// Check for empty EEPROM
if (Empty_EEPROM)
{
    Sample_Error(); // Attempting to read data that hasn't been written
} else
{
    // Move the bank pointer backwards to read data
    Bank_Pointer -= 4;

    // Transfer contents of Current Page to Read Buffer
    for(i=0;i<4;i++)
    {
        Read_Buffer[i] = *(Bank_Pointer++);
    }
}
```

5.2.5 EEPROM_Erase

The EEPROM_Erase() 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.

```
// Re-Configure Write/Erase Protection Masks used by the Flash API
uint64 WE_Protection_AB_Mask =
Configure_Protection_Masks(FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT ^ 1], NUM_EEPROM_SECTORS);

// Assign individual protection masks accordingly
WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32)WE_Protection_AB_Mask;
WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;

Erase_Bank();
```

For comparison, the F28003x Ping Pong example project's EEPROM_Erase functionality differs by only calling the Erase_Bank function. Write/erase protection masks are configured outside of the EEPROM_Erase call.

5.2.5.1 Erase_Bank

The Erase_Bank function leverages the Flash API to erase the inactive (full) EEPROM Unit. This function is separate from EEPROM_Erase to minimize the CPU cycles required to erase both units in the EEPROM_Config function. It begins by configuring the Write/Erase Protection masks for the appropriate Flash Sectors and then calls Fapi_issueBankEraseCommand. Finally, it waits for completion and checks for any errors.

```
Fapi_StatusType  oReturnCheck;

// Clears status of previous Flash operation
ClearFSMStatus();

// Enable program/erase protection for select sectors
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Erase the inactive EEPROM Bank
oReturnCheck = Fapi_issueBankEraseCommand((uint32*) FLASH_BANK_SELECT);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);
```

For comparison, the F28003x Ping Pong example project's EEPROM_Bank function issues

the Flash API erase command and then waits for completion, checking for any programming errors that occur. Write/Erase protection masks are provided outside of the scope of the function.

5.2.6 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() and EEPROM_Read() functions. GetValidBank Flow shows the overall flow required to search for current EEPROM bank and page.

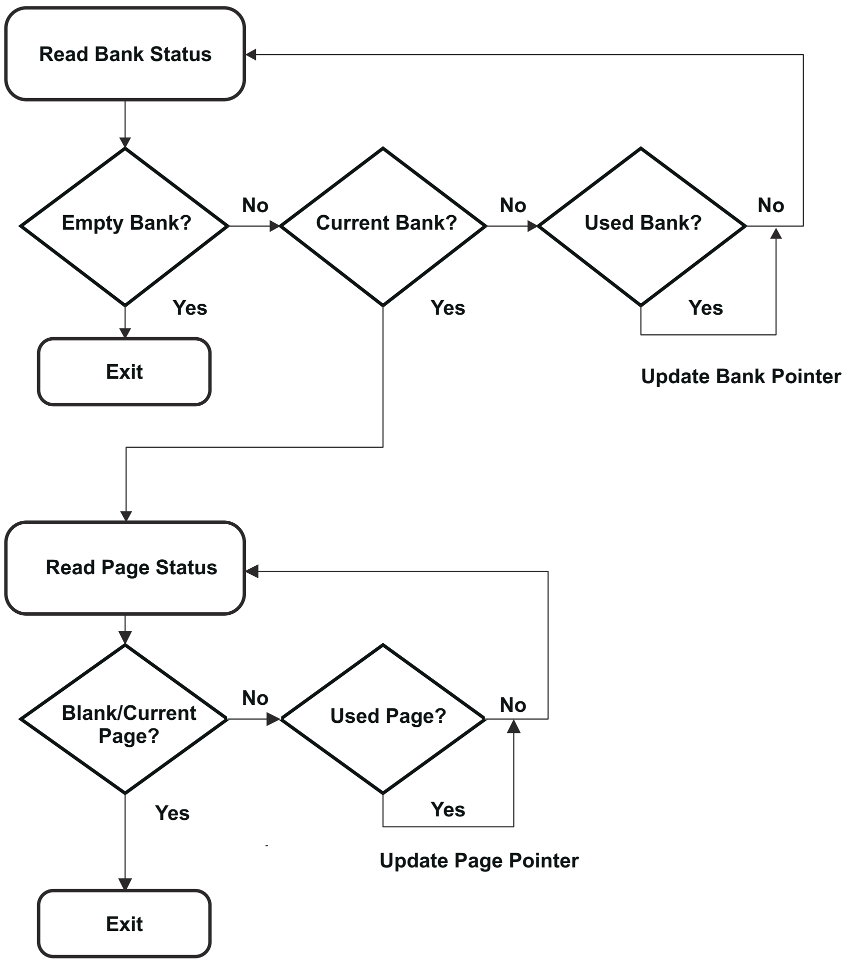


Figure 5-1. GetValidBank Flow

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 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.

An Empty EEPROM Bank is tested first. If this status is encountered, the EEPROM bank has not been used and no further searching is needed.

```
if(Bank_Status[0] == EMPTY_BANK)        // Check for Unused EEPROM Bank
{
    Bank_Counter = i;  // Set EEPROM Bank Counter to number of current page
    return;            // If EEPROM Bank is Unused, return as EEPROM is empty
}
```

If an Empty EEPROM Bank is not encountered, Current EEPROM Bank is tested next. If the EEPROM bank is the current EEPROM bank, the EEPROM bank counter is updated 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 EEPROM bank searching is needed.

```
if(Bank_Status[0] == CURRENT_BANK && Bank_Status[4] != CURRENT_BANK)      // Check for Current Bank
{
    Bank_Counter = i;      // Set Bank Counter to number of current bank
    // Set Page Pointer to first page in current bank
    Page_Pointer = Bank_Pointer + 8;
    break;      // Break from loop as current bank has been found
}
```

Lastly, Used EEPROM Bank is tested. 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.

```
// Check for Used Bank
if(Bank_Status[0] == CURRENT_BANK && Bank_Status[4] == CURRENT_BANK)
// If Bank has been used, set pointer to next 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 0x5F5F5F5F5F5F5F5F, with the remaining 64 bits set to 1 (0x5F5F5F5F5F5F5F5FFFFFFFFFFFFFFFFF). A Used Page is signified by all 128 bits being set to 0x5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F. These values can be changed if desired.

The Blank and Current Pages are tested 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.

```
// Check for Blank Page or Current Page
if(Page_Status[0] == BLANK_PAGE)
{
        Page_Counter = i;  // Set Page Counter to number of current page
        break;  // Break from loop as current page has been found
}

if (Page_Status[0] == CURRENT_PAGE && Page_Status[4] != CURRENT_PAGE)
{
       Page_Counter = i + 1;//Increment Page Counter as one has been used
       break; // Break from loop as current page has been found
}
```

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.

```
// Check for Used Page
if(Page_Status[0] == CURRENT_PAGE && Page_Status[4] == CURRENT_PAGE)
{
// If page has been used, set pointer to next page
        Page_Pointer += EEPROM_PAGE_DATA_SIZE + 8;
}
```

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 active units will be switched, Write/Erase Protection masks are reconfigured, and the Erase_Inactive_Unit flag is set.

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


        uint64 WE_Protection_AB_Mask = Configure_Protection_Masks(
                                      FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT],
                                         NUM_EEPROM_SECTORS);


        WE_Protection_A_Mask = 0xFFFFFFFF ^(uint32)WE_Protection_AB_Mask;
        WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;

        Erase_Inactive_Unit = 1;
        RESET_BANK_POINTER;
        RESET_PAGE_POINTER;
    }
}
```

This check is performed by testing the EEPROM bank and page counters. The amount 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 erasing of the inactive EEPROM unit when reading from a full EEPROM unit.

5.2.7 EEPROM_UpdateBankStatus

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

```
Bank_Status[0] = *(Bank_Pointer);
Page_Status[0] = *(Page_Pointer);
```

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

```
// Set Bank Status to Current Bank
Bank_Status[0] = CURRENT_BANK;
Bank_Status[1] = CURRENT_BANK;
Bank_Status[2] = CURRENT_BANK;
Bank_Status[3] = CURRENT_BANK;

// Clears status of previous Flash operation
ClearFSMStatus();
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Program Bank Status to current bank
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer,
                                               Bank_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);

// Set Page Pointer to first page of current bank
Page_Counter = 0;
Page_Pointer = Bank_Pointer + 8;
```

If the status is not empty, the next check is for a Current EEPROM Bank with all pages used in the EEPROM bank (full EEPROM bank). In this case, the current EEPROM banks status will be updated to show the EEPROM bank is Full and the next EEPROM banks status will be updated to Current to allow programming of the next EEPROM bank. As a last step, the page pointer is updated to the first page of the new EEPROM bank.

```
// Set Bank Status to Used Bank
Bank_Status[0] = CURRENT_BANK;
Bank_Status[1] = CURRENT_BANK;
Bank_Status[2] = CURRENT_BANK;
Bank_Status[3] = CURRENT_BANK;

// Clears status of previous Flash operation
ClearFSMStatus();
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Program Bank Status to full bank
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer + 2,
                                              Bank_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);

// Increment Bank Pointer to next bank
Bank_Pointer += Bank_Size;

// Set Bank Status to Current Bank
Bank_Status[0] = CURRENT_BANK;
Bank_Status[1] = CURRENT_BANK;
Bank_Status[2] = CURRENT_BANK;
Bank_Status[3] = CURRENT_BANK;

// Clears status of previous Flash operation
ClearFSMStatus();
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Program Bank Status to current bank
```

```
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer,
                                              Bank_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);

// Set Page Pointer to first page of current bank
Page_Counter = 0;
Page_Pointer = Bank_Pointer + 8;
```

5.2.8 EEPROM_UpdatePageStatus

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

```
Bank_Status[0] = *(Bank_Pointer);   // Read Bank Status from Bank Pointer
Page_Status[0] = *(Page_Pointer);   // Read Page Status from Page Pointer
```

If this status indicates that the page is blank, the function is exited as this status is updated in the EEPROM_Write() 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:

```
// Check if Page Status is blank. If so return to EEPROM_WRITE.
if(Page_Status[0] == BLANK_PAGE)
    return;

// Program previous page's status to Used Page
else
{

    // Set Page Status to Used Page
    Page_Status[0] = CURRENT_PAGE;
    Page_Status[1] = CURRENT_PAGE;
    Page_Status[2] = CURRENT_PAGE;
    Page_Status[3] = CURRENT_PAGE;

    // Clears status of previous Flash operation
    ClearFSMStatus();

    Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
    Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

    // Program Bank Status to current bank
    oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Page_Pointer+2,
                                                 Page_Status, 4, 0, 0,
                                                  Fapi_AutoEccGeneration);

    // Wait for completion and check for any programming errors
    EEPROM_CheckStatus(&oReturnCheck);

    // Increment Page Pointer to next page
    Page_Pointer += EEPROM_PAGE_DATA_SIZE + 8;
}
```

5.2.9 EEPROM_UpdatePageData

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

The following steps need to be taken to achieve this:

1. Clear the Flash State Machine (FSM) Status.
2. Configure program/erase protection for Flash sectors.
    - Sectors not used in EEPROM Emulation will have protection enabled.
    - Sectors used in EEPROM Emulation will have protection disabled.
3. Calculate the offset from the Page Pointer to write the data.
    a. This is required because only 64-bits are written at a time. Thus, if the data size is greater than 64-bits, multiple calls to the Flash API are necessary to write an entire Page.

4. Wait for programming completion and check for any programming errors.

```
// Clears status of previous Flash operation
ClearFSMStatus();

Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Variable for page offset
// (first write position has offset of 2 (64 bits),
// second has offset of 4 (128 bits), etc.)
uint32 Page_Offset = 4 + (2 * i);

// Program data located in Write_Buffer to current page
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Page_Pointer + Page_Offset, Write_Buffer
+ (i*4), 4, 0, 0, Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);
```

The following parameters are passed to the Flash API for programming.

- Page Pointer (programming address)
- Buffer containing data to be written
- Length of data to be programmed
- Programming mode

The fourth and fifth parameters are zero when using Fapi_AutoEccGeneration mode. For more details, see the *TMS320F28P65x Flash API Version 3.02.00.00 Reference Guide*.

If the programming is successful, the Page Status of the current Page is updated and the Empty_EEPROM flag is cleared. The code is shown below:

```
if(oReturnCheck == Fapi_Status_Success)
{
    // Set Page Status to Current Page
    Page_Status[0] = CURRENT_PAGE;
    Page_Status[1] = CURRENT_PAGE;
    Page_Status[2] = CURRENT_PAGE;
    Page_Status[3] = CURRENT_PAGE;

    Fapi_setupBankSectorEnable(
                       FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA,
                          WE_Protection_A_Mask);

    Fapi_setupBankSectorEnable(
                       FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB,
                          WE_Protection_B_Mask);

    oReturnCheck = Fapi_issueProgrammingCommand((uint32*)Page_Pointer,
                                                Page_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

    // Wait for completion and check for any programming errors
    EEPROM_CheckStatus(&oReturnCheck);
    Empty_EEPROM = 0;
}
```

After a successful write, the function checks if the inactive EEPROM unit needs to be erased. If so, it calls EEPROM_Erase, clears the flag, and re-configures the W/E Protection Masks. The flag to set the blank check is raised before erase is called. The flag to erase the inactive unit is set in EEPROM_GetValid_Bank.

```
if (Erase_Inactive_Unit)
{
    // Erase the inactive (full) EEPROM Bank
    Erase_Blank_Check = 1;
    EEPROM_Erase();
    Erase_Inactive_Unit = 0;

    // Re-configure Write/Erase Protection Masks for active EEPROM Bank
    uint64 WE_Protection_AB_Mask = Configure_Protection_Masks(
          FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT], NUM_EEPROM_SECTORS);
    WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32)WE_Protection_AB_Mask;
    WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;
}
```

5.2.10 EEPROM_Get_64_Bit_Data_Address

The EEPROM_Get_64_Bit_Data_Address() provides functionality for determining if the EEPROM unit is full and assigning the proper address, if required. If a full EEPROM unit is detected, EEPROM is erased using the EEPROM_Erase() function and the active EEPROM unit is switched.

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.

```
End_Address = (uint16 *)END_OF_SECTOR;  // Set End_Address for sector
```

Next, the EEPROM bank pointer is compared to the end address. If writing four 16-bit words beginning at the current EEPROM bank pointer would go beyond the end address, this indicates the sector is full. At this point, 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-3)         // Test if EEPROM is full
{
    EEPROM_ACTIVE_UNIT ^= 1;

    uint64 WE_Protection_AB_Mask = Configure_Protection_Masks(
                                           FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT],
                                                  NUM_EEPROM_SECTORS);


    WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32)WE_Protection_AB_Mask;
    WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;

    Erase_Inactive_Unit = 1;
    RESET_BANK_POINTER;
}
```

5.2.11 EEPROM_Program_64_Bits

The EEPROM_Program_64_Bits() function provides functionality for programming four 16-bit words to memory. The first parameter, Num_Words, allows the user to specify how many valid words will be written. The data words should be assigned to the first 4 locations of the Write_Buffer to be used by the Fapi_issueProgrammingCommand function. If less than four words are specified in the function call, missing words will be filled with 0xFFFF. This is done to comply with ECC requirements.

First, a full EEPROM unit is tested for.

```
EEPROM_Get_64_Bit_Data_Address();
```

Next, the Write Buffer is filled with 1s if less than 4 words are specified.

```
int i;
for(i = Num_Words; i < 4; i++)
{
    Write_Buffer[i] = 0xFFFF;
}
```

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

```
// Clears status of previous Flash operation
ClearFSMStatus();

Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer,
                                                Write_Buffer, 4, 0, 0,
                                                Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);

// Increment to next location
Bank_Pointer += 4;
```

Once programming is complete, the Erase_Inactive_Unit flag is checked. If set, the inactive unit is erased, performs a blank check and the Write/Erase Protection masks are reconfigured.

```
if (Erase_Inactive_Unit) {

    // Erase inactive unit
    Erase_Blank_Check = 1;
    EEPROM_Erase();
    Erase_Inactive_Unit = 0;

    uint64 WE_Protection_AB_Mask = Configure_Protection_Masks(
                                        FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT],
                                            NUM_EEPROM_SECTORS);


    WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32)WE_Protection_AB_Mask;
    WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;
}
```

Note

This function cannot be used until RESET_BANK_POINTER has been executed to set the pointer. In this example, it is called in EEPROM_Config_Check(). Executing before could produce unknown results.

5.2.12 EEPROM_CheckStatus

The EEPROM_CheckStatus function provides functionality to check the Flash API status and check the Flash State Machine status after each program/erase to Flash. There is also an additional check to confirm the flash is blank following an erase operation. If any unexpected statuses are detected, the program stops. Error handling is not implemented in this project.

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_FlashStatusWordType oFlashStatusWord;

uint32_t sectorAddress = FLASH_BANK_SELECT + FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT^1][0] *
FLASH_SECTOR_SIZE;
uint16_t sectorSize = (FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT^1][1] -
FIRST_AND_LAST_SECTOR[EEPROM_ACTIVE_UNIT^1][0] + 1) * (FLASH_SECTOR_SIZE / 2);

// Wait until the Flash program operation is over
while(Fapi_checkFsmForReady() == Fapi_Status_FsmBusy);

if(*oReturnCheck != Fapi_Status_Success)
{
    // Check Flash API documentation for possible errors
    Sample_Error();
}


// Read FMSTAT register contents to know the status of FSM after
// program command to see if there are any program operation related
// errors
oFlashStatus = Fapi_getFsmStatus();

    if (Erase_Inactive_Unit && Erase_Blank_Check){
            *oReturnCheck = Fapi_doBlankCheck((uint32_t *) sectorAddress,
                                          sectorSize, &oFlashStatusWord);
            Erase_Blank_Check = 0;
        }

if(*oReturnCheck != Fapi_Status_Success || oFlashStatus != 3)
{
    //Check FMSTAT and debug accordingly
    Sample_Error();
}
```

5.2.13 ClearFSMStatus

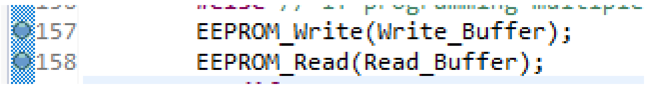
The ClearFSMStatus() function is responsible for clearing the status of the previous flash operation. This function is applicable for F280013x, F280015x, F28P65x and F28P55x devices. This function must be used as-is.

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_StatusType  oReturnCheck;

// Wait until FSM is done with the previous flash operation
while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

oFlashStatus = Fapi_getFsmStatus();

if(oFlashStatus != 0)
{

    /* Clear the Status register */
    oReturnCheck = Fapi_issueAsyncCommand(Fapi_ClearStatus);

    // Wait until status is cleared
    while (Fapi_getFsmStatus() != 0) {}

    if(oReturnCheck != Fapi_Status_Success)
    {
        // Check Flash API documentation for possible errors
        Sample_Error();
    }
}
```

5.3 Testing Example

The examples provided were tested with F28P650DK9. 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.

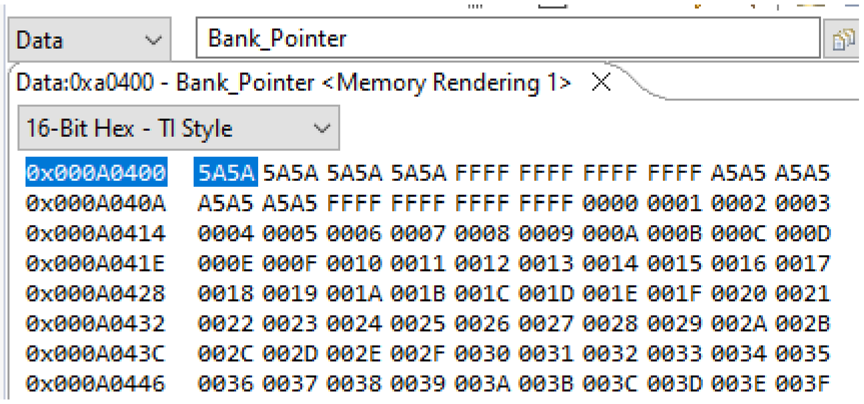
1. Connect the F28P650DK9 to the PC via USB and an XDS110 Debug Probe with JTAG connection.
2. Connect a 5V DC power supply to the board.
3. Start Code Composer Studio and open the F28P65x_EEPROM_PingPong_Example.pjt.
4. Build the project by selecting Project -> Build Project.
5. Launch the Target Configurations by going to View -> Target Configurations -> F28P65x_EEPROM_PingPong_Example -> targetConfigs -> right-click TMS320F28P650DK9.ccxml -> Launch Selected Configuration.
6. Connect to CPU1 by going to the Debug window, right clicking Texas Instruments XDS110 USB Debug Probe_0/C28xx_CPU1, and selecting Connect Target.
7. Load symbols by clicking Load Symbols and selecting the F28P65x_EEPROM_PingPong.out from the project.
8. Set breakpoints to properly view data written to and read from the memory window as shown in Break Points.



```
        EEPROM_Write(Write_Buffer);
        EEPROM_Read(Read_Buffer);
```

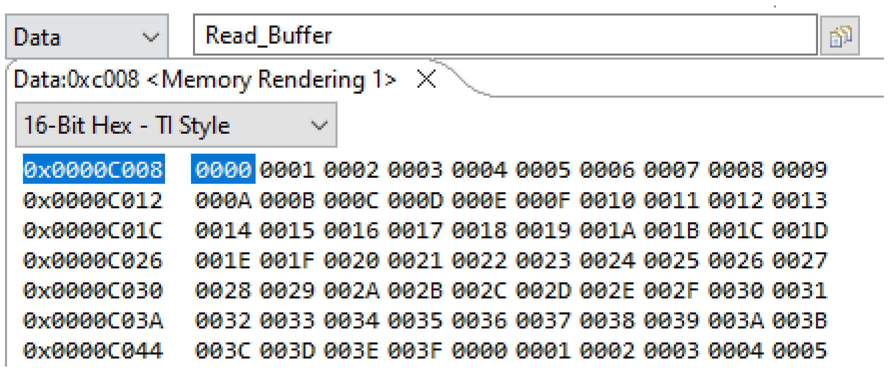
Figure 5-2. Break Points

9. 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.



```
Data    Bank_Pointer
Data:0xa0400 - Bank_Pointer <Memory Rendering 1>
16-Bit Hex - TI Style

0x000A0400   5A5A 5A5A 5A5A 5A5A FFFF FFFF FFFF FFFF A5A5 A5A5
0x000A040A   A5A5 A5A5 FFFF FFFF FFFF FFFF 0000 0001 0002 0003
0x000A0414   0004 0005 0006 0007 0008 0009 000A 000B 000C 000D
0x000A041E   000E 000F 0010 0011 0012 0013 0014 0015 0016 0017
0x000A0428   0018 0019 001A 001B 001C 001D 001E 001F 0020 0021
0x000A0432   0022 0023 0024 0025 0026 0027 0028 0029 002A 002B
0x000A043C   002C 002D 002E 002F 0030 0031 0032 0033 0034 0035
0x000A0446   0036 0037 0038 0039 003A 003B 003C 003D 003E 003F
```

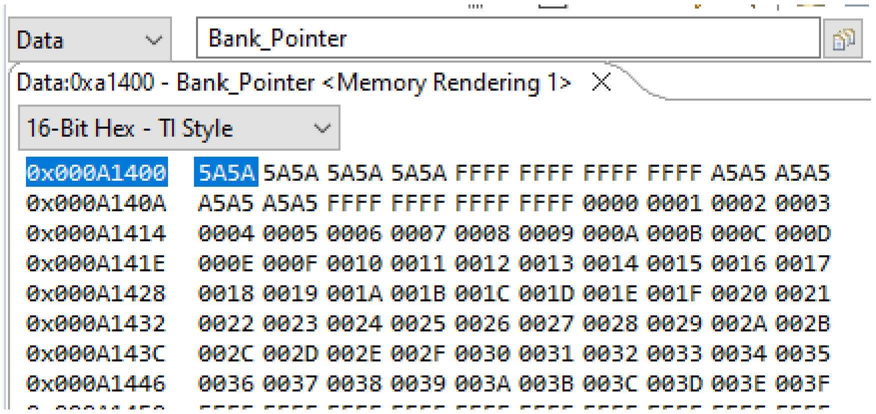
Figure 5-3. Write to EEPROM Unit



```
Data    Read_Buffer
Data:0xc008 <Memory Rendering 1>
16-Bit Hex - TI Style

0x0000C008   0000 0001 0002 0003 0004 0005 0006 0007 0008 0009
0x0000C012   000A 000B 000C 000D 000E 000F 0010 0011 0012 0013
0x0000C01C   0014 0015 0016 0017 0018 0019 001A 001B 001C 001D
0x0000C026   001E 001F 0020 0021 0022 0023 0024 0025 0026 0027
0x0000C030   0028 0029 002A 002B 002C 002D 002E 002F 0030 0031
0x0000C03A   0032 0033 0034 0035 0036 0037 0038 0039 003A 003B
0x0000C044   003C 003D 003E 003F 0000 0001 0002 0003 0004 0005
```

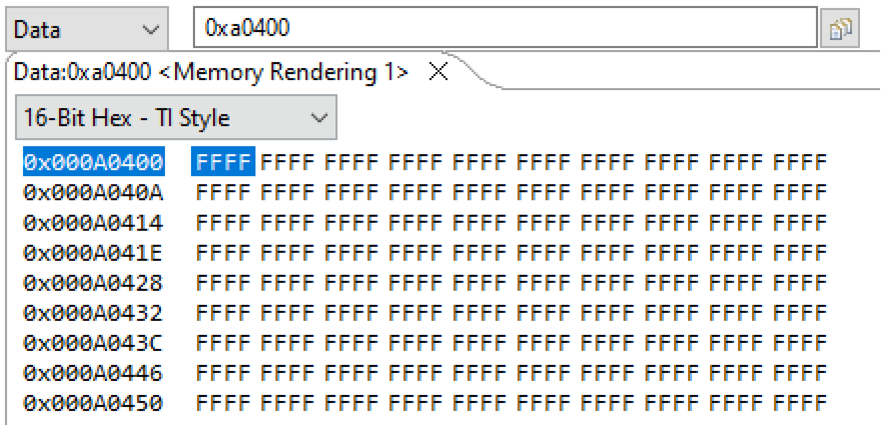
Figure 5-4. Read Data

10. Continue running from break-point to break-point until the program has finished or EEPROM is full.
11. 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.



```
Data    Bank_Pointer
Data:0xa1400 - Bank_Pointer <Memory Rendering 1>
16-Bit Hex - TI Style

0x000A1400   5A5A 5A5A 5A5A 5A5A FFFF FFFF FFFF FFFF A5A5 A5A5
0x000A140A   A5A5 A5A5 FFFF FFFF FFFF FFFF 0000 0001 0002 0003
0x000A1414   0004 0005 0006 0007 0008 0009 000A 000B 000C 000D
0x000A141E   000E 000F 0010 0011 0012 0013 0014 0015 0016 0017
0x000A1428   0018 0019 001A 001B 001C 001D 001E 001F 0020 0021
0x000A1432   0022 0023 0024 0025 0026 0027 0028 0029 002A 002B
0x000A143C   002C 002D 002E 002F 0030 0031 0032 0033 0034 0035
0x000A1446   0036 0037 0038 0039 003A 003B 003C 003D 003E 003F
```

Figure 5-5. Write to new EEPROM Unit



Data 0xa0400

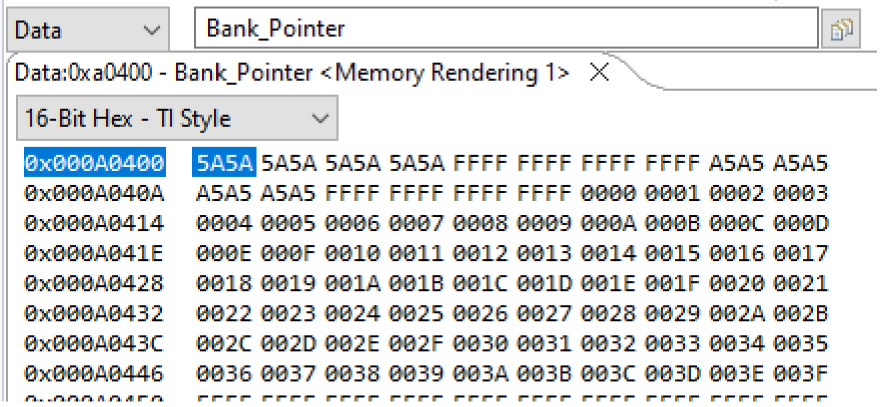
Data:0xa0400 <Memory Rendering 1>

16-Bit Hex - TI Style

```
0x000A0400   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A040A   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0414   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A041E   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0428   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0432   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A043C   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0446   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0450   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
```

Figure 5-6. Erase full EEPROM Unit

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



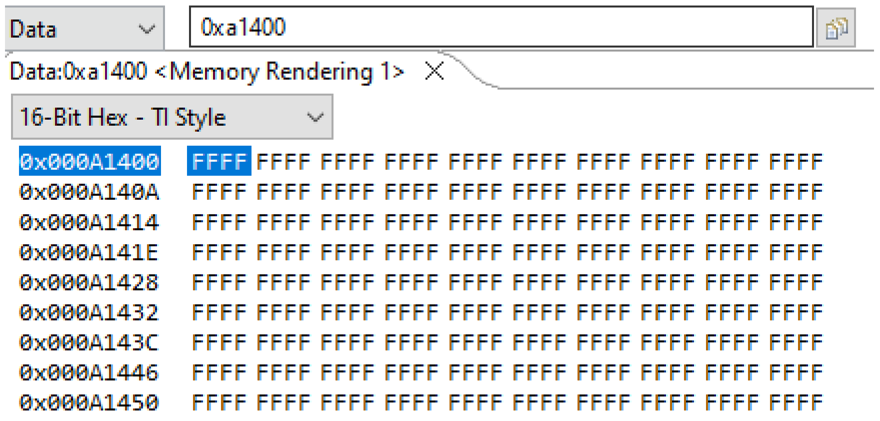
Data Bank_Pointer

Data:0xa0400 - Bank_Pointer <Memory Rendering 1>

16-Bit Hex - TI Style

```
0x000A0400   5A5A 5A5A 5A5A 5A5A FFFF FFFF FFFF FFFF A5A5 A5A5
0x000A040A   A5A5 A5A5 FFFF FFFF FFFF FFFF 0000 0001 0002 0003
0x000A0414   0004 0005 0006 0007 0008 0009 000A 000B 000C 000D
0x000A041E   000E 000F 0010 0011 0012 0013 0014 0015 0016 0017
0x000A0428   0018 0019 001A 001B 001C 001D 001E 001F 0020 0021
0x000A0432   0022 0023 0024 0025 0026 0027 0028 0029 002A 002B
0x000A043C   002C 002D 002E 002F 0030 0031 0032 0033 0034 0035
0x000A0446   0036 0037 0038 0039 003A 003B 003C 003D 003E 003F
```

Figure 5-7. Write to original EEPROM unit



Data 0xa1400

Data:0xa1400 <Memory Rendering 1>

16-Bit Hex - TI Style

```
0x000A1400   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A140A   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A1414   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A141E   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A1428   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A1432   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A143C   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A1446   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A1450   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
```

Figure 5-8. Erase full EEPROM Unit

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.

6 Single-Unit Emulation

Single-Unit Emulation is similar to the Ping Pong Emulation, but only uses one set of Flash Sectors. Thus, the Ping Pong method cannot be used and when a full EEPROM is detected. The implemented behavior is shown in this section. Many of the functions remain the same between the two modes, and the primary differences are seen in the Erase behavior.

6.1 User-Configuration

The implementation detailed in this document allows you to configure several variables for EEPROM Emulation. These variables are mainly found within EEPROM_Config.h, but one is contained in F28P65x_EEPROM.c.

6.1.1 EEPROM_Config.h

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

- Define which device variant is being used. This allows EEPROM emulation in Flash Banks not common to all devices

```
// Un-comment appropriate definition if one of the following variants is being used
#define F28P65xDKx 1
//#define F28P65xSKx 1
//#define F28P65xSHx 1
```

- Choose between Page Mode and 64-Bit Mode

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

- Choose which Flash Bank to use for emulation. **The Flash API and program are stored/run from Flash Bank 0 by default, so it cannot be used for EEPROM Emulation. In general, the Flash API and program should be stored/run from a different bank than the ones used for EEPROM emulation.**

```
#define FLASH_BANK_SELECT FlashBank1StartAddress
```

- Define the Flash Sector size (unit is 16-bit words). This will vary based on the device being used, reference the appropriate data sheet for details.

```
#define FLASH_SECTOR_SIZE F28P65x_FLASH_SECTOR_SIZE
```

- Define how many Flash sectors are in a Flash Bank. This will vary based on the device being used, reference the appropriate data sheet for details.

```
#define NUM_FLASH_SECTORS F28P65x_NUM_FLASH_SECTORS
```

- 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 3
```

- Choose the size of data contained within each EEPROM Page (unit is 16-bit words). Although any size can be specified, the size will be adjusted to the closest multiple of four that is greater than or equal to the size specified. For example, a specified size of 6 16-bit words per page will be programmed as 8 16-bit words per page, with the last two being treated as 0xFFFF. This is to comply with Flash requirements (8-bit ECC is programmed for every 64-bit aligned Flash memory address).

```
#define DATA_SIZE 64
```

6.1.2 F28P65x_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 only use sector 1, insert {1,1}. A valid configuration will have the following properties.

- Imply the same amount of sectors for emulation as specified in EEPROM_Config.h
- 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 F28P65x 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 F28P65x Flash API Reference Guide.

More details about invalid or dangerous configurations can be found in Section 6.2.1.

```
uint16 FIRST_AND_LAST_SECTOR[2][2] = {1,1};
```

6.2 EEPROM Functions

To implement this functionality, 11 functions are required to configure, program, read, and erase in Page programming. Two additional functions are needed for 64-bit programming. All functions are included in the F28P65x_EEPROM.c or F28P65x_EEPROM.c file.

- EEPROM_Config_Check()
- Configure_Protection_Masks(Uint16* Sector_Numbers, Uint16 Num_EEPROM_Sectors)
- EEPROM_Write(Uint16* Write_Buffer)
- EEPROM_Read(Uint16* Read_Buffer)
- EEPROM_Erase( )
- EEPROM_GetValidBank(Uint16 Read_Flag)
- EEPROM_UpdateBankStatus( )
- EEPROM_UpdatePageStatus( )
- EEPROM_UpdatePageData(Uint16* Write_Buffer)
- EEPROM_Get_64_Bit_Data_Address()
- EEPROM_Program_64_Bits(Uint16 Num_Words)
- EEPROM_CheckStatus(Fapi_StatusType* oReturnCheck);
- ClearFSMStatus()

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

6.2.1 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 are valid. A valid Flash Bank selection must not select Bank 0 for emulation and must be supported by a specific device variant. For example, only specific F28p65x variants have Flash Banks 2-4. To verify this information, see the device-specific data-sheet.

```
if (FLASH_BANK_SELECT == FlashBank0StartAddress)
{
    return 0xFFFF;
}

if (FLASH_BANK_SELECT == FlashBank2StartAddress)
{
#if !defined(F28P65xDKx) && !defined(F28P65xSKx) && !defined(F28P65xSHx)
        return 0xFFFF;
#endif
} else if (FLASH_BANK_SELECT == FlashBank3StartAddress) // If using Bank 3
{
#if !defined(F28P65xDKx) && !defined(F28P65xSKx)
        return 0xFFFF;
#endif
} else if (FLASH_BANK_SELECT == FlashBank4StartAddress)
{
#if !defined(F28P65xDKx) && !defined(F28P65xSKx) && !defined(F28P65xSHx)
        return 0xFFFF;
#endif
}
```

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

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

```
NUM_EEPROM_SECTORS = FIRST_AND_LAST_SECTOR[1] - FIRST_AND_LAST_SECTOR[0] + 1;
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[1] <= FIRST_AND_LAST_SECTOR[0])
    {
        return 0xEEEE;
     }
     // Check if SECTOR_NUMBERS contains invalid sector
    if (FIRST_AND_LAST_SECTOR[0] > NUM_FLASH_SECTORS - 1)
    {
        return 0xEEEE;
    }
    if (FIRST_AND_LAST_SECTOR[1] > NUM_FLASH_SECTORS - 1 || FIRST_AND_LAST_SECTOR[1] < 1)
    {
        return 0xEEEE;
    }
} else // If only one sector, validate it is input properly
{

// Verify that the only sector is valid
if (FIRST_AND_LAST_SECTOR[0] > NUM_FLASH_SECTORS - 1) {
    return 0xEEEE;
}
```

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

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

```
// Calculate size of each EEPROM Bank (16 bit words)
Bank_Size = 8 + ((EEPROM_PAGE_DATA_SIZE + 8) * NUM_EEPROM_PAGES);

// Calculate amount of available space (16 bit words)
uint32 Available_Words = NUM_EEPROM_SECTORS * FLASH_SECTOR_SIZE;

// Check if size of EEPROM Banks and Pages will fit in EEPROM sectors
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:

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

```
// Notify for extra space (more than one bank leftover)
if (Available_Words - (Bank_Size * NUM_EEPROM_BANKS ) >= Bank_Size)
{
    Warning_Flags += 1;
}
```

- If each page consists of less than 5 16-bit words (this wastes space as the 64-Bit Mode could be used without the need for Status Codes)

```
if (EEPROM_PAGE_DATA_SIZE < 5)
{
    Warning_Flags += 2;
}
```

- If using sectors in the 32-127 range (for F28P65x devices) and not using all eight sectors allocated to a single bit in the Write 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 details on the Write Protection Masks, see the *TMS320F28P65x Flash API Version 3.02.00.00 Reference Guide*.

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

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

Finally, Write/Erase Protection masks are configured for the Active EEPROM Unit. This function also prepares Flash for Emulation by erasing the Sectors to be used for programming.

```
uint64 WE_Protection_AB_Mask = Configure_Protection_Masks(FIRST_AND_LAST_SECTOR, 
NUM_EEPROM_SECTORS);


WE_Protection_A_Mask = 0xFFFFFFFF ^ (uint32)WE_Protection_AB_Mask;
WE_Protection_B_Mask = 0x00000FFF ^ WE_Protection_AB_Mask >> 32;


EEPROM_Erase();
```

6.2.2 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 details on the implementation of the Fapi_setupBankSectorEnable function, see the *TMS320F28P65x Flash API Version 3.02.00.00 Reference Guide*.

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

```
// Initialize a variable to store the bits indicating which sectors
// need to have write/erase protection disabled.
// The first lower 32 bits will represent CMDWEPROTA and the upper 32
// bits will represent CMDWEPROTB.
uint64 Protection_Mask_Sectors = 0;

// If we have more than one Flash Sector
if (Num_EEPROM_Sectors > 1)
{

    uint64 Unshifted_Sectors;
    uint16 Shift_Amount;

    // If all sectors use Mask A
    if (Sector_Numbers[0] < 32 && Sector_Numbers[1] < 32)
    {

        // Configure Mask A
        Unshifted_Sectors = (uint64) 1 << Num_EEPROM_Sectors;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);

    }// If all sectors use Mask B
    else if (Sector_Numbers[0] > 31 && Sector_Numbers[1] > 31)
    {

        // Configure Mask B
        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;

    } else // If both Masks A and B need to be configured
    {

        // 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;
        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]);
    }


} else { // If only using 1 Flash Sector

    if(Sector_Numbers[0] < 32)
    {
        Protection_Mask_Sectors |= ((uint64) 1 << Sector_Numbers[0]);
    } else
    {
        Protection_Mask_Sectors |= ((uint64) 1 << ((Sector_Numbers[0] - 32)/8));
        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;
    }

}

return Protection_Mask_Sectors;
```

For comparison, the F28003x EEPROM example project's Configure_Protection_Masks functionality differs from that of the F28P65x EEPROM project example with the amount of sectors available for protection. Each bit in the Write/Erase protection mask represents it's own sector.

```
// Initialize a variable to store the bits indicating which sectors need to have write/erase
    // protection disabled.
    uint16 Protection_Mask_Sectors = 0;
    uint16 Unshifted_Sectors;

    // If we have more than one Flash Sector
    if (Num_EEPROM_Sectors > 1)
    {
        // Configure mask
        Unshifted_Sectors = (uint16) 1 << Num_EEPROM_Sectors;
        Unshifted_Sectors -= 1;
        Protection_Mask_Sectors |= (Unshifted_Sectors << Sector_Numbers[0]);

    } else { // If only using 1 Flash Sector

        if(Sector_Numbers[0] < 16)
        {
            Unshifted_Sectors = (uint16) 1 << Sector_Numbers[0];
            Protection_Mask_Sectors |= Unshifted_Sectors;
        }

    }

    return Protection_Mask_Sectors;
```

6.2.3 EEPROM_Write

The EEPROM_Write() 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();     // Find Current Bank and Current Page

EEPROM_UpdatePageStatus(); // Update Page Status of previous page
EEPROM_UpdateBankStatus(); // Update Bank Status of current and previous bank
EEPROM_UpdatePageData();   // Update Page Data of current page
```

6.2.4 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.

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

```
uint16 i;

// Check for empty EEPROM
if (Empty_EEPROM)
{
    Sample_Error(); // Attempting to read data that hasn't been written
} else
{
    // Find Current Bank and Current Page
    EEPROM_GetValidBank(1);

    // Increment page pointer to point at first data word
    Page_Pointer += 8;

    // Transfer contents of Current Page to Read Buffer
    for(i=0;i<DATA_SIZE;i++)
    {
        Read_Buffer[i] = *(Page_Pointer++);
    }
    }
```

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

```
uint16 i;

// Check for empty EEPROM
if (Empty_EEPROM)
{
    Sample_Error(); // Attempting to read data that hasn't been written
} else
{
    // Move the bank pointer backwards to read data
    Bank_Pointer -= 4;

    // Transfer contents of Current Page to Read Buffer
    for(i=0;i<4;i++)
    {
        Read_Buffer[i] = *(Bank_Pointer++);
    }
}
```

6.2.5 EEPROM_Erase

The EEPROM_Erase() function provides functionality for erasing the 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 Single Unit implementation, this function is only called when all EEPROM Banks and Pages are full.

The function begins by configuring the Write/Erase Protection masks for the EEPROM unit, and then calls the Fapi_issueBankEraseCommand function. Finally, it waits for completion and checks for any errors.

```
Fapi_StatusType  oReturnCheck;

// Clears status of previous Flash operation
ClearFSMStatus();
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);

Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB,WE_Protection_B_Mask);

// Erase the EEPROM Bank
oReturnCheck = Fapi_issueBankEraseCommand((uint32*)FLASH_BANK_SELECT);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);
```

In the Single-Unit implementation, the EEPROM_Erase function leverages the Flash API to clear the Flash Bank. It no longer has any need for the Erase_Bank function from the Ping Pong Implementation and the two have been combined in EEPROM_Erase. Erase_Bank is no longer needed because it was created to optimize the erasing of all Flash Sectors designated for EEPROM Emulation when there were two EEPROM units.

For comparison, the F28003x example project's EEPROM_Erase function issues

the Flash API erase command and then waits for completion, checking for any programming errors that occur. Write/Erase protection masks are provided outside of the scope of the function.

```
Fapi_StatusType  oReturnCheck;

// Erase the EEPROM Bank
oReturnCheck = Fapi_issueBankEraseCommand((uint32*) FLASH_BANK_SELECT, WE_Protection_Mask);
// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);
```

6.2.6 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() and EEPROM_Read() functions. GetValidBank Flow shows the overall flow required to search for current EEPROM bank and page.

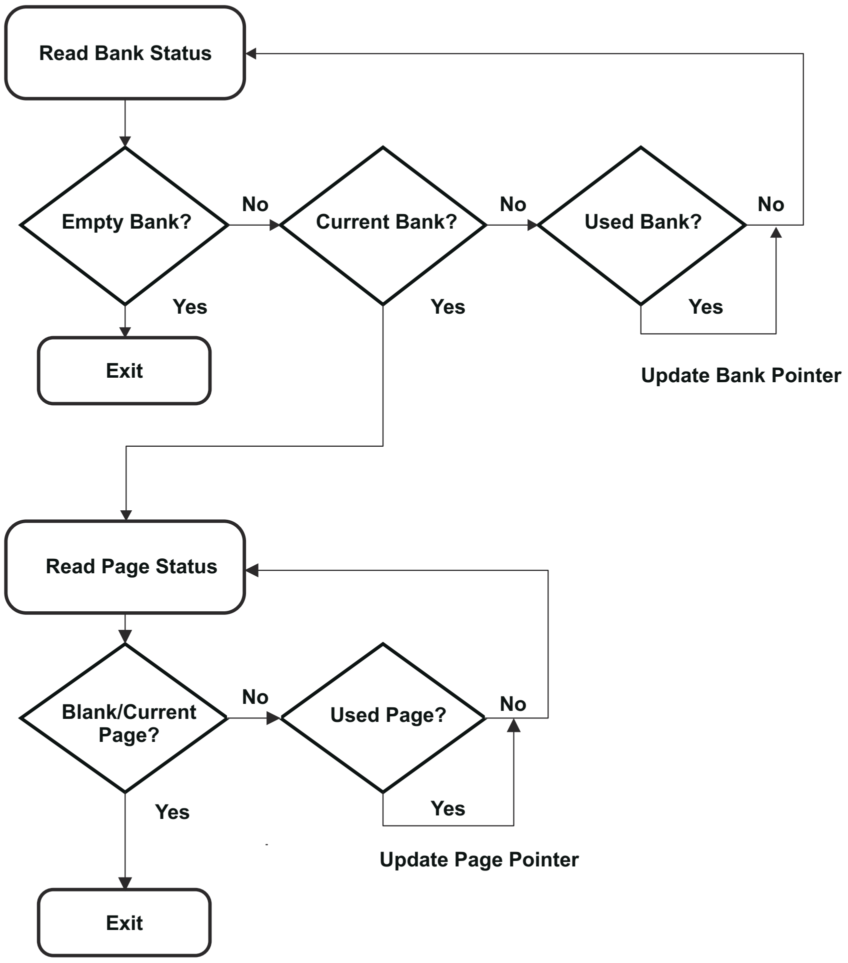


Figure 6-1. GetValidBank Flow

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 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.

An Empty EEPROM Bank is tested first. If this status is encountered, the EEPROM bank has not been used and no further searching is needed.

```
if(Bank_Status[0] == EMPTY_BANK)        // Check for Unused Bank
{
    Bank_Counter = i;     // Set Bank Counter to number of current page
    return;               // If Bank is Unused, return as EEPROM is empty
}
```

If an Empty EEPROM Bank is not encountered, Current EEPROM Bank is tested next. If the bank is the current EEPROM bank, the EEPROM bank counter is updated 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 EEPROM bank searching is needed.

```
if(Bank_Status[0] == CURRENT_BANK && Bank_Status[4] != CURRENT_BANK)      // Check for Current Bank
{
    Bank_Counter = i;      // Set Bank Counter to number of current bank
    // Set Page Pointer to first page in current bank
    Page_Pointer = Bank_Pointer + 8;
    break;      // Break from loop as current bank has been found
}
```

Lastly, Used EEPROM Bank is tested. 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.

```
// Check for Used Bank
if(Bank_Status[0] == CURRENT_BANK && Bank_Status[4] == CURRENT_BANK)
// If Bank has been used, set pointer to next 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 EEPROM Bank is signified by the first 64 bits being set to 0x5F5F5F5F5F5F5F5F, with the remaining 64 bits set to 1. A Used EEPROM Bank is signified by all 128 bits being set to 0x5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F. These values can be changed if desired.

The Blank and Current Pages are tested 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.

```
// Check for Blank Page or Current Page
if(Page_Status[0] == BLANK_PAGE)
{
        Page_Counter = i;  // Set Page Counter to number of current page
        break;  // Break from loop as current page has been found
}

if (Page_Status[0] == CURRENT_PAGE && Page_Status[4] != CURRENT_PAGE)
{
       Page_Counter = i + 1;//Increment Page Counter as one has been used
       break; // Break from loop as current page has been found
}
```

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.

```
// Check for Used Page
if(Page_Status[0] == CURRENT_PAGE && Page_Status[4] == CURRENT_PAGE)
{
    // If page has been used, set pointer to next page
    Page_Pointer += EEPROM_PAGE_DATA_SIZE + 8;
}
```

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.

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

This check is performed by testing the EEPROM bank and page counters. The amount 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 erasing of the inactive EEPROM unit when reading from a full EEPROM unit.

As show above, if the memory is full, the EEPROM_Erase() functions is called and the EEPROM bank and page pointers are reset to the first EEPROM bank and page.

6.2.7 EEPROM_Get_64_Bit_Data_Address

The EEPROM_Get_64_Bit_Data_Address is largely unchanged in the Single-Unit implementation, but the behavior upon detecting a full EEPROM unit is different.

```
if(Bank_Pointer > End_Address-3)         // Test if EEPROM is full
{
    Erase_Inactive_Unit = 1;
    Erase_Blank_Check = 1;
    EEPROM_Erase();
    Erase_Inactive_Unit = 0;
    RESET_BANK_POINTER;
}
```

As shown above, if the EEPROM unit is full, it is simply erased, performs a blank check and the pointer is reset to the beginning of the unit.

6.2.8 EEPROM_UpdateBankStatus

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

```
Bank_Status[0] = *(Bank_Pointer);
Page_Status[0] = *(Page_Pointer);
```

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

```
// Set Bank Status to Current Bank
Bank_Status[0] = CURRENT_BANK;
Bank_Status[1] = CURRENT_BANK;
Bank_Status[2] = CURRENT_BANK;
Bank_Status[3] = CURRENT_BANK;

// Clears status of previous Flash operation
ClearFSMStatus();

Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB. WE_Protection_B_Mask);
```

```
// Program Bank Status to current bank
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer,
                                               Bank_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);

// Set Page Pointer to first page of current bank
Page_Counter = 0;
Page_Pointer = Bank_Pointer + 8;
```

If the status is not empty, the next check is for a Current EEPROM Bank with all pages used in the EEPROM bank (full EEPROM bank). In this case, the current EEPROM banks status will be updated to show the EEPROM bank is Full and the next EEPROM banks status will be updated to Current to allow programming of the next EEPROM bank. As a last step, the page pointer is updated to the first page of the new EEPROM bank.

```
// Set Bank Status to Used Bank
Bank_Status[0] = CURRENT_BANK;
Bank_Status[1] = CURRENT_BANK;
Bank_Status[2] = CURRENT_BANK;
Bank_Status[3] = CURRENT_BANK;

// Clears status of previous Flash operation
ClearFSMStatus();

Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Program Bank Status to full bank
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer + 2,
                                              Bank_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);

// Increment Bank Pointer to next bank
Bank_Pointer += Bank_Size;

// Set Bank Status to Current Bank
Bank_Status[0] = CURRENT_BANK;
Bank_Status[1] = CURRENT_BANK;
Bank_Status[2] = CURRENT_BANK;
Bank_Status[3] = CURRENT_BANK;

// Clears status of previous Flash operation
ClearFSMStatus();
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Program Bank Status to current bank
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer,
                                              Bank_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);

// Set Page Pointer to first page of current bank
Page_Counter = 0;
Page_Pointer = Bank_Pointer + 8;
```

6.2.9 EEPROM_UpdatePageStatus

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

```
Bank_Status[0] = *(Bank_Pointer);   // Read Bank Status from Bank Pointer
Page_Status[0] = *(Page_Pointer);   // Read Page Status from Page Pointer
```

If this status indicates that the page is blank, the function is exited as this status is updated in the EEPROM_Write() 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:

```
// Check if Page Status is blank. If so return to EEPROM_WRITE.
if(Page_Status[0] == BLANK_PAGE)
    return;

// Program previous page's status to Used Page
else
{

    // Set Page Status to Used Page
    Page_Status[0] = CURRENT_PAGE;
    Page_Status[1] = CURRENT_PAGE;
    Page_Status[2] = CURRENT_PAGE;
    Page_Status[3] = CURRENT_PAGE;

    // Clears status of previous Flash operation
    ClearFSMStatus();

    Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
    Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

    // Program Bank Status to current bank
    oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Page_Pointer+2,
                                                Page_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

    // Wait for completion and check for any programming errors
    EEPROM_CheckStatus(&oReturnCheck);

    // Increment Page Pointer to next page
    Page_Pointer += EEPROM_PAGE_DATA_SIZE + 8;
}
```

6.2.10 EEPROM_UpdatePageData

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

The following steps need to be taken to achieve this:

1. Clear the Flash State Machine (FSM) Status.
2. Configure program/erase protection for Flash sectors.
    a. Sectors not used in EEPROM Emulation will have protection enabled.
    b. Sectors used in EEPROM Emulation will have protection disabled.
3. Calculate the offset from the Page Pointer to write the data.
    a. This is required because only 64-bits are written at a time. Thus, if the data size is greater than 64-bits, multiple calls to the Flash API are necessary to write an entire Page.
4. Wait for programming completion and check for any programming errors.

```
// Clears status of previous Flash operation
ClearFSMStatus();

Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
    Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

// Variable for page offset
//(first write position has offset of 2 (64 bits),
// second has offset of 4 (128 bits), etc.)
uint32 Page_Offset = 4 + (2 * i);

// Program data located in Write_Buffer to current page
oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Page_Pointer + Page_Offset,Write_Buffer +
(i*4), 4, 0, 0,Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);
```

The following parameters are passed to the Flash API for programming.

- Page Pointer (programming address)
- Buffer containing data to be written
- Length of data to be programmed
- Programming mode

The fourth and fifth parameters are zero when using Fapi_AutoEccGeneration mode. For more details, see the *TMS320F28P65x Flash API Version 3.02.00.00 Reference Guide*.

If the programming is successful, the page status of the current page is updated and the Empty_EEPROM flag is cleared. The code is shown below:

```
if(oReturnCheck == Fapi_Status_Success)
{
    // Set Page Status to Current Page
    Page_Status[0] = CURRENT_PAGE;
    Page_Status[1] = CURRENT_PAGE;
    Page_Status[2] = CURRENT_PAGE;
    Page_Status[3] = CURRENT_PAGE;

    Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);

    Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

    oReturnCheck = Fapi_issueProgrammingCommand((uint32*)Page_Pointer,
                                                Page_Status, 4, 0, 0,
                                                 Fapi_AutoEccGeneration);

    // Wait for completion and check for any programming errors
    EEPROM_CheckStatus(&oReturnCheck);
    Empty_EEPROM = 0;
}
if (Erase_Inactive_Unit)
{
        // Erase the inactive (full) EEPROM Bank
        Erase_Inactive_Unit = 0;
}
```

6.2.11 EEPROM_Get_64_Bit_Data_Address

The EEPROM_Get_64_Bit_Data_Address() provides functionality for determining if the EEPROM unit is full and assigning the proper address if required. If a full EEPROM unit is detected, EEPROM is erased using the EEPROM_Erase() function and the address is reset to the beginning of the first Flash Sector.

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.

```
End_Address = (uint16 *)END_OF_SECTOR;  // Set End_Address for sector
```

Next, the EEPROM bank pointer is compared to the end address. If writing four 16-bit words beginning at the current EEPROM bank pointer would go beyond the End Address, this indicates the sector 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-3)         // Test if EEPROM is full
{
   Erase_Inactive_Unit = 1;
   Erase_Blank_Check = 1;
   EEPROM_Erase();
   Erase_Inactive_Unit = 0;
   RESET_BANK_POINTER;
}
```

6.2.12 EEPROM_Program_64_Bits

The EEPROM_Program_64_Bits() function provides functionality for programming four 16-bit words to memory. The first parameter, Num_Words, allows the user to specify how many valid words are written. The data words are assigned to the first 4 indexes of the Write_Buffer to be used by the Fapi_issueProgrammingCommand function. If less than four words are specified in the function call, missing words are filled with 0xFFFF. This is done to comply with ECC requirements.

First, a full EEPROM unit is tested for.

```
EEPROM_Get_64_Bit_Data_Address();
```

Next, the Write Buffer is filled with 1s if less than 4 words are specified.

```
int i;
for(i = Num_Words; i < 4; i++)
{
    Write_Buffer[i] = 0xFFFF;
}
```

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

```
// Clears status of previous Flash operation
ClearFSMStatus();

Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA, WE_Protection_A_Mask);
Fapi_setupBankSectorEnable(FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB, WE_Protection_B_Mask);

oReturnCheck = Fapi_issueProgrammingCommand((uint32*) Bank_Pointer,
                                                Write_Buffer, 4, 0, 0,
                                                Fapi_AutoEccGeneration);

// Wait for completion and check for any programming errors
EEPROM_CheckStatus(&oReturnCheck);
Empty_EEPROM = 0;
// Increment to next location
Bank_Pointer += 4;
```

Note

This function cannot be used until RESET_BANK_POINTER has been executed to set the pointer. In this example, it is called in EEPROM_Config_Check(). Executing before can produce unknown results.

6.2.13 EEPROM_CheckStatus

The EEPROM_CheckStatus function provides functionality to check the Flash API status and check the Flash State Machine status after each program/erase to Flash. If any unexpected statuses are detected, the program stops. Error handling is not implemented in this project.

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_FlashStatusWordType oFlashStatusWord;

uint32_t sectorAddress = FLASH_BANK_SELECT + FIRST_AND_LAST_SECTOR[0] * FLASH_SECTOR_SIZE;
uint16_t sectorSize = (FIRST_AND_LAST_SECTOR[1] - FIRST_AND_LAST_SECTOR[0] + 1) *
(FLASH_SECTOR_SIZE / 2);

// Wait until the Flash program operation is over
while(Fapi_checkFsmForReady() == Fapi_Status_FsmBusy);

if(*oReturnCheck != Fapi_Status_Success)
{
    // Check Flash API documentation for possible errors
    Sample_Error();
}


// Read FMSTAT register contents to know the status of FSM after
// program command to see if there are any program operation related
// errors
oFlashStatus = Fapi_getFsmStatus();
if (Erase_Inactive_Unit && Erase_Blank_Check){
        *oReturnCheck = Fapi_doBlankCheck((uint32_t *) sectorAddress,
                                          sectorSize, &oFlashStatusWord);
        Erase_Blank_Check = 0;
}
if(*oReturnCheck != Fapi_Status_Success || oFlashStatus != 3)
{
    //Check FMSTAT and debug accordingly
    Sample_Error();
}
```

6.2.14 ClearFSMStatus

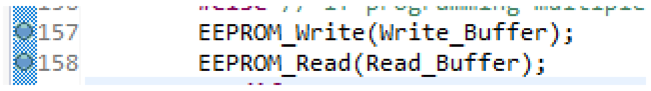
The ClearFSMStatus() function is responsible for clearing the status of the previous flash operation. This function is applicable for F280013x, F280015x, F28P65x and F28P55x devices. This function must be used as-is.

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_StatusType  oReturnCheck;

// Wait until FSM is done with the previous flash operation
while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

oFlashStatus = Fapi_getFsmStatus();

if(oFlashStatus != 0)
{

    /* Clear the Status register */
    oReturnCheck = Fapi_issueAsyncCommand(Fapi_ClearStatus);

    // Wait until status is cleared
    while (Fapi_getFsmStatus() != 0) {}

    if(oReturnCheck != Fapi_Status_Success)
    {
        // Check Flash API documentation for possible errors
        Sample_Error();
    }
}
```

6.3 Testing Example

The examples provided were tested with F28P650DK9. 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.

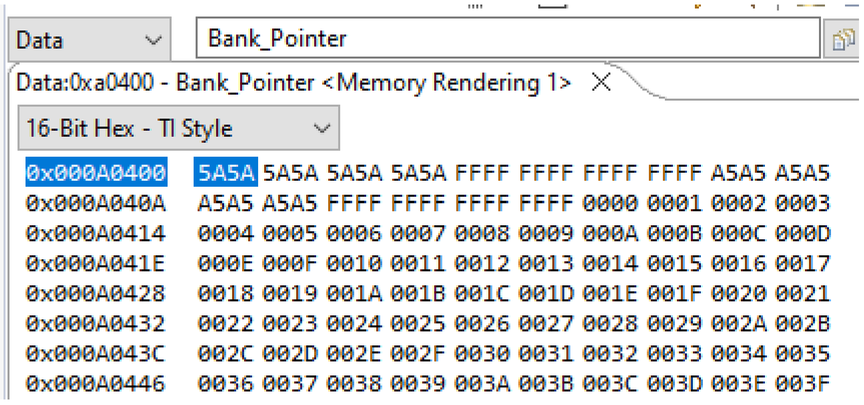
1. Connect the F28P650DK9 to the PC via USB and an XDS110 Debug Probe with JTAG connection.
2. Connect a 5V DC power supply to the board.
3. Start Code Composer Studio and open the F28P65x_EEPROM_Example.pjt.
4. Build the project by selecting Project -> Build Project.
5. Launch the Target Configurations by going to View -> Target Configurations -> F28P65x_EEPROM_Example -> targetConfigs -> right-click TMS320F28P650DK9.ccxml -> Launch Selected Configuration.
6. Connect to CPU1 by going to the Debug window, right clicking Texas Instruments XDS110 USB Debug Probe_0/C28xx_CPU1, and selecting Connect Target.
7. Load symbols by clicking Load Symbols and selecting the F28P65x_EEPROM.out from the project.
8. Set breakpoints to properly view data written to and read from the memory within the memory window as shown in Break Points.



```
157        EEPROM_Write(Write_Buffer);
158        EEPROM_Read(Read_Buffer);
```

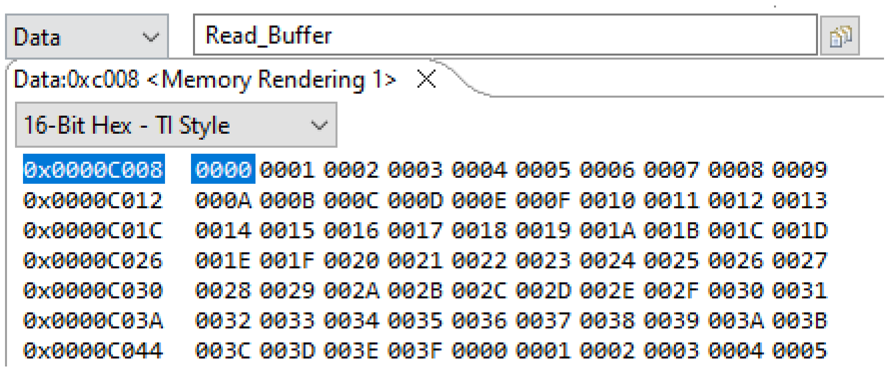
Figure 6-2. Break Points

9. 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.



```
0x000A0400   5A5A 5A5A 5A5A 5A5A FFFF FFFF FFFF FFFF A5A5 A5A5
0x000A040A   A5A5 A5A5 FFFF FFFF FFFF FFFF 0000 0001 0002 0003
0x000A0414   0004 0005 0006 0007 0008 0009 000A 000B 000C 000D
0x000A041E   000E 000F 0010 0011 0012 0013 0014 0015 0016 0017
0x000A0428   0018 0019 001A 001B 001C 001D 001E 001F 0020 0021
0x000A0432   0022 0023 0024 0025 0026 0027 0028 0029 002A 002B
0x000A043C   002C 002D 002E 002F 0030 0031 0032 0033 0034 0035
0x000A0446   0036 0037 0038 0039 003A 003B 003C 003D 003E 003F
```

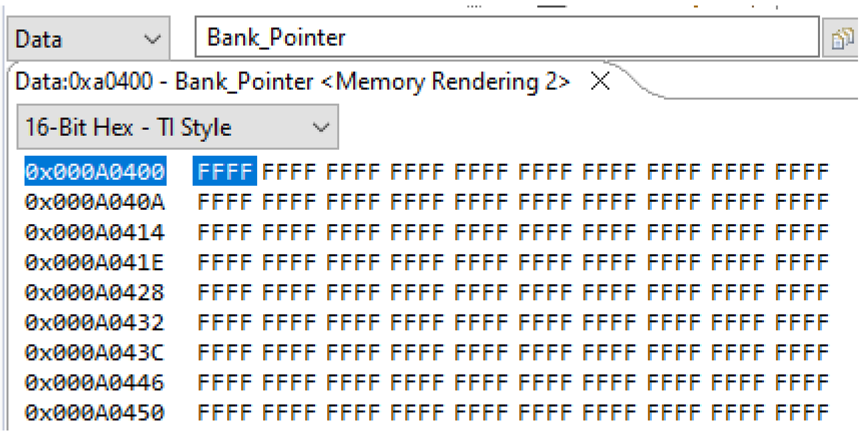
Figure 6-3. Write to EEPROM



```
0x0000C008   0000 0001 0002 0003 0004 0005 0006 0007 0008 0009
0x0000C012   000A 000B 000C 000D 000E 000F 0010 0011 0012 0013
0x0000C01C   0014 0015 0016 0017 0018 0019 001A 001B 001C 001D
0x0000C026   001E 001F 0020 0021 0022 0023 0024 0025 0026 0027
0x0000C030   0028 0029 002A 002B 002C 002D 002E 002F 0030 0031
0x0000C03A   0032 0033 0034 0035 0036 0037 0038 0039 003A 003B
0x0000C044   003C 003D 003E 003F 0000 0001 0002 0003 0004 0005
```

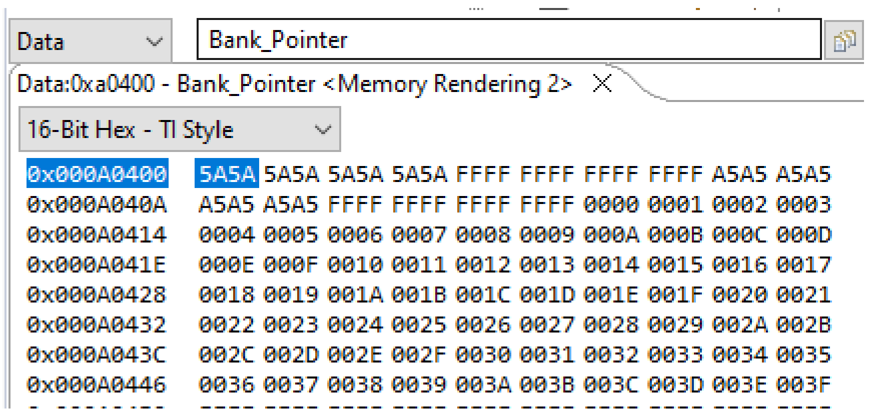
Figure 6-4. Read Data

10. Continue running from breakpoint to breakpoint until the program has finished or EEPROM is full.
11. 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.



```
0x000A0400   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A040A   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0414   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A041E   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0428   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0432   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A043C   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0446   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0x000A0450   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
```

Figure 6-5. Erase full EEPROM Unit



```
0x000A0400  5A5A 5A5A 5A5A 5A5A FFFF FFFF FFFF FFFF A5A5 A5A5
0x000A040A  A5A5 A5A5 FFFF FFFF FFFF FFFF 0000 0001 0002 0003
0x000A0414  0004 0005 0006 0007 0008 0009 000A 000B 000C 000D
0x000A041E  000E 000F 0010 0011 0012 0013 0014 0015 0016 0017
0x000A0428  0018 0019 001A 001B 001C 001D 001E 001F 0020 0021
0x000A0432  0022 0023 0024 0025 0026 0027 0028 0029 002A 002B
0x000A043C  002C 002D 002E 002F 0030 0031 0032 0033 0034 0035
0x000A0446  0036 0037 0038 0039 003A 003B 003C 003D 003E 003F
```

Figure 6-6. Write to EEPROM

12. 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.

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 the F28P650DK9, the following files are needed:

- F28P65x_EEPROM.c
- EEPROM_Config.h
- Device.c and device.h
- flash_programming_f28p65x.h
- FAPI_F28P65x_EABI_v3.02.00.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 Adapting to Other Gen 3 C2000 MCUs

As discussed earlier in the document, this guide uses the F28P65x to demonstrate the EEPROM Emulation functionality. However, this project can be adapted to other Gen 3 C2000 MCUs by making small changes to macros and function definitions. To show this, this section discusses the changes required to use the F280013x.

First and foremost, it should be noted that F280013x only has one Flash Bank as opposed to the five within certain F28P65x devices. Thus, the CPU1_RAM Build Configuration should be used instead of the CPU1_FLASH Build Configuration. This is necessary because the Flash API cannot be executed on the same Flash Bank in which it is contained.

Additionally, the default configuration contained in the EEPROM_Config.h file uses device specific values to create definitions and macros. These should be updated to the values found in the *TMS320F280013x Real-Time Microcontrollers Data Sheet*. In the case of the F280013x, these values happen to be the same as the default configuration for the F28P65x, but these values should always be verified with the device-specific data sheet.

```
#define FLASH_BANK_SELECT 0x80000

#define FLASH_SECTOR_SIZE 0x400

#define NUM_FLASH_SECTORS 128
```

These values are important for error-checking within EEPROM_Config_Check as well as defining the beginning/end address of EEPROM Emulation.

Finally, the EEPROM_Config_Check() function needs to be modified when using the F280013x. By default, Flash Bank 0 is reserved for storing the Flash API, and the function will throw an error if this Flash Bank is selected for EEPROM Emulation. However, since the CPU_1_RAM Build Configuration is selected, Flash Bank 0 is now available for EEPROM Emulation. Thus, these lines should be removed or commented out in the function.

```
if (FLASH_BANK_SELECT == FlashBank0StartAddress)
{
    return 0xFFFF;
}
```

While the changes required for using the F280013x are relatively simple, using other Gen 3 C2000 MCUs can require more changes. For a list of available projects, see the Troubleshooting section.

9 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 16-bit word at a time, but 64-bits must be written every time to align with ECC requirements.

9.1 Flash API Checklist

This following section is taken from the 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.

9.1.1 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. Note that the flash API library does not support flash erase/program operations when the system frequency is less than or equal to 20 MHz.
- Configure BANKMUXSEL and FLASHCTLSEM registers as needed
- 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 *TMS320F28P65x Flash API Version 3.02.00.00 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 or OTP on which the Flash erase/program operation is in progress
- ECC should not be programmed for link-pointer locations. The API skips programming the ECC when the start address provided for the program operation is any of the three link-pointer addresses. Care should be taken to maintain a separate structure/section for link-pointer locations in the application. Do not mix these fields with other DCSM OTP settings. If other fields are mixed with link-pointers, API skips programming ECC for the non-link-pointer locations as well. This causes ECC errors in the application.

10 Source File Listing

File	Function	Description

F28P65x_EEPROM_PingPong.c	EEPROM_Config_Check()
	Configure_Protection_Masks()
	EEPROM_Write()
	EEPROM_Read()
	EEPROM_Erase()
	Erase_Bank()
	EEPROM_GetValidBank()
	EEPROM_UpdateBankStatus()
	EEPROM_UpdatePageStatus()
	EEPROM_UpdatePageData()
	EEPROM_Get_64_Bit_Data_Address()
	EEPROM_Program_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

F28P65x_EEPROM.c	EEPROM_Config_Check()
	Configure_Protection_Masks()
	EEPROM_Write()
	EEPROM_Read()
	EEPROM_Erase()
	EEPROM_GetValidBank()
	EEPROM_UpdateBankStatus()
	EEPROM_UpdatePageStatus()
	EEPROM_UpdatePageData()
	EEPROM_Get_64_Bit_Data_Address()
	EEPROM_Program_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

11 Troubleshooting

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

11.1 General

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

Device	Build Configurations	Location
F28003x	RAM, FLASH	C2000Ware_5_02_xx_xx > driverlib > f28003x > examples > flash
F28P65x	RAM, FLASH	C2000Ware_5_02_xx_xx > driverlib > f28p65x > c28x > examples > 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: Device variation, programming mode (64 bit vs. Page), Flash Bank selection, Flash sector size, number of Flash sectors, number of EEPROM banks, number of EEPROM pages, 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/disabled for the appropriate sector(s) selected for EEPROM Emulation for your device, consulting the device's Flash API reference guide for more information.
- 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.

12 Conclusion

This application report has proven that the F28P65x Generation 3 C2000 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.

13 References

- Texas Instruments, *EEPROM Emulation for Gen 2 C2000 Real-Time MCUs*
- Texas Instruments, *TMS320F28P65x Flash API Version 3.02.00.00 Reference Guide*
- Texas Instruments, *TMS320F28P65x Real-Time Microcontrollers*
- Texas Instruments, *TMS320F280013x Real-Time Microcontrollers*

14 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (November 2023) to Revision A (April 2024)	Page
• Added project path for F28003x examples	1
• Updated Note to include TMS320F28003x device	2
• Added project path for F28003x examples	6
• Updated F28P65x code blocks and added F28003x code block comparison	8
• Updated F28P65x code blocks and added F28003x code block comparison	29
• Updated F28P65x project file name for Single-Unit Emulation and updated related function and descriptions	49
• Added Troubleshooting section with F28P65x and F28003x example project paths and general user questions	50