Application Report
SPRAB69–September 2009

EEPROM Emulation With the TMS320F28xxx DSCs

Tim Love and Pradeep Shinde

ABSTRACT

Many applications require storing small quantities of system related data (e.g., calibration values, device configuration) in a non-volatile memory, so that it can be used or modified and reused even after power cycling the system. EEPROMs are 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 one sector of onboard Flash memory as the emulated electrically erasable programmable read-only memory (EEPROM) are 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 downloaded from the following URL: http://www.ti.com/lit/zip/SPRAB69.

1 Introduction

The F28xxx parts come with different configurations of Flash memory that is arranged in multiple sectors. Unfortunately, the CMOS process technology used for the on-chip Flash memory does not allow adding a traditional EEPROM on the chip. Some designers use the external EEPROM part for such non-volatile storage. As such, Flash memory is a specific type of EEPROM. The good news is all the F28xxx parts have in-circuit programming capability for the Flash memory. This application report makes use of this facility and allows using one sector of on-chip Flash as EEPROM by emulating the EEPROM functionality within the limitations of the Flash memory. Note that one *Flash sector* is entirely used as an emulated EEPROM; therefore, it is not available for the application code.

2 Difference Between EEPROM and On-Chip Flash

EEPROMs are available in different capacities and connect with the host microcontroller 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.

Compared to EEPROM, Flash memories have higher density, which allow larger memory arrays (sectors) implemented on-chip. 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 an F28xxx part 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; but not the other way around. The on-chip Flash memory on F28xxx parts require TI-supplied specific algorithms (Flash API) for erase and write operations.

The major difference between EEPROM and Flash operations is seen in the write and erase timings. A typical Flash write time is 50 µs/16-bit word; whereas, EEPROM typically requires 5 to 10 ms. The EEPROM does not require a page (sector) erase operation. One can erase a particular byte requiring the specified time. Flash erase time runs in seconds for a page. For F28xxx, the typical value for erase time is 10 seconds/8K sector. The Flash power supply must be steady during write/erase operations.

There are challenges involved in emulating EEPROM with the Flash, due to their different characteristics.

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

3 Implementation Scheme

While emulating EEPROM with the Flash, the most important challenge is meeting the reliability targets as far as Flash program/erase endurance and data retention are concerned. Secondly, real-time application requirements of updating and reading the data need to be met under control of the application program. Note that during the Flash erase/program period it cannot execute the application program, as it cannot be read during this time interval.

NOTE: Flash erase and write cycles draw additional current from V_{DD} and V_{DDIO} supply voltage rails. The system power supply should be designed to source this additional current. For the values of the additional current, see the Flash Timing section of the device-specific data manual.

NOTE: Flash endurance is rated for 100 min/1000 typical erase/write cycles. This is shown in the Flash Timing section in Electrical Characteristics of the device-specific data manual. Programming data to the sector, byte-by-byte, helps to prolong Flash life. This is due to the fact that the entire sector is used before erasing as opposed to only partially using the sector and erasing.

3.1 Basic Concept

A Flash sector has to be entirely reserved for the emulated EEPROM, due to the block erase requirement of Flash. This means, based on the F28xxx part, the size for this Flash sector will be anything from 4K x 16 (F2801, F28015/F28016) to 32K x 16 (F28335/F28235, F2809). The area of this Flash sector is divided into the number of smaller sections (referred to as *Page*, hereafter) each having the size of emulated EEPROM. For example a 4K x 16 size Flash sector can be divided into 32 pages, each having a size of 128 x 16. That makes each Page equivalent to a 256 x 8 Bytes EEPROM. The data to be *saved* is first written in a buffer in RAM. Then, using the *in-circuit programming* facility of the F28xxx, the data is *written* to the first page (Page 1) in the selected sector of the Flash.

After the power cycling, the application copies this *saved* data from the page to the buffer in RAM. The next time the application requires saving new data that gets written to the next page, Page 2. The process continues until the last page in the selected sector is written.

On reaching the last page, the entire Flash sector has to be erased and the process starts again from Page 1 for the consecutive EEPROM data save operation.

In addition to the multi-byte programming concept described above, there is also support for single-byte programming. In this mode, the sector is not broken into banks and pages. Single-byte programming is discussed further in Section 4.2.7 and Section 4.2.8. The remainder of this application report discusses the multi-byte programming mode.

3.2 Creating EEPROM Sections (Pages) and Page Identification

It is possible to create a large number of pages depending on the size of Flash sectors and the emulated EEPROM. It is necessary to:

- Read back the data from the page written during the previous save.
- Write the latest data to a new page.
- Know the *last* page, for erasing all pages and start from Page 1.
- Read from any previously stored data, if required by the application.

To simplify the tracking of a page, the Flash sector is first divided into *Banks*. Each bank is further divided into *Pages*. This allows a smaller number of banks and number of pages within each bank. This partitioning is shown in Figure 1.

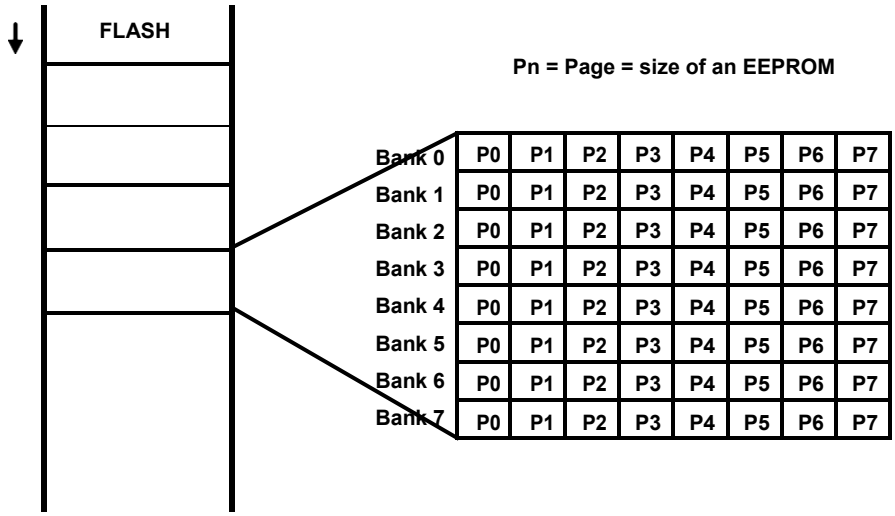

Figure 1. Bank Partitioning

The first word of each bank is reserved for bank status information and the first word of each page 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 bank is used the Bank Status of the last and current banks are updated.

All pages contain a page status and 64 bytes. Page 0 is slightly different as it contains the bank status as well (see Figure 2). Pages 0 and 1 are only shown. It should be noted that Pages 2-7 are identical to Page 1.

Page 0

Bank	Page Status	Byte 0	Byte 1	Byte 2
Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
. . .	. . .	. . .	. . .	. . .
Byte 59	Byte 60	Byte 61	Byte 62	Byte 63

Page 1

Page Status	Byte 0	Byte 1	Byte 2	Byte 3
Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
. . .	. . .	. . .	. . .	. . .
Byte 59	Byte 60	Byte 61	Byte 62	Byte 63

Figure 2. Page Layout

4 Software Description

The software provided with this application report includes source code for each TMS320F28xxx generation with an example project demonstrating how to utilize the source code for all devices within each generation.

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

The software package is self contained and will extract with F28xxx_EEPROM as the base directory. This code uses the Header Files [3] and Flash API libraries [1], [2] provided for each F28xxx generation. Figure 3 shows the directory structure of the F28xxx_EEPROM directory.

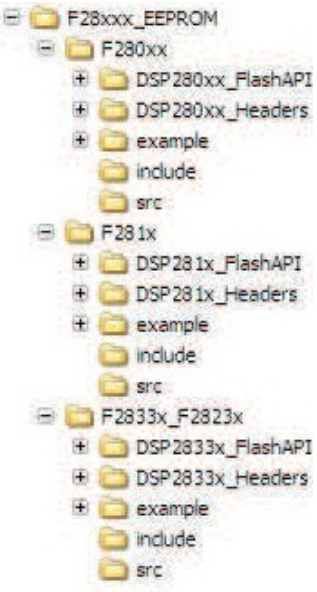

Figure 3. F28xxx_EEPROM Directory Structure

As shown, each F28xxx generation has its own folder that contains subfolders. The Flash API and header files for each generation are provided. The example folder contains the example project and source files. The include and src folders contain the header files and source files for implementing the EEPROM emulation. This code was tested in Code Composer Studio™ version 3.3 with F28xxx Code Generation Tools version 5.2.0.

It is assumed that *Running an Application From Internal Flash Memory on the TMS320F28xx DSP* (SPRA958) [5] has been viewed and its methodologies are followed for Flash implementation.

4.1 Software Functionality and Flow

The high-level software flow is shown in Figure 4. The device must first go through its initialization code to initialize clocks, peripherals, etc. The initialization functions used are the functions that are provided with the header files software package [3]. The provided examples follow the same initialization steps as the examples from the header files software package [3]. Further information regarding this sequence can be read in the documentation provided with the header files [3].

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 is included with the library download. This list is also available in Appendix A along with an overview of the Flash API. The library download for each F28xxx generation provides a library file for each chip within that generation.

At this point, programming can begin. First, data needs to be captured to program. After programming this data, the read functionality will read 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 the Flash API needs to be copied to internal RAM before programming can begin. The example projects provided follow this software flow.

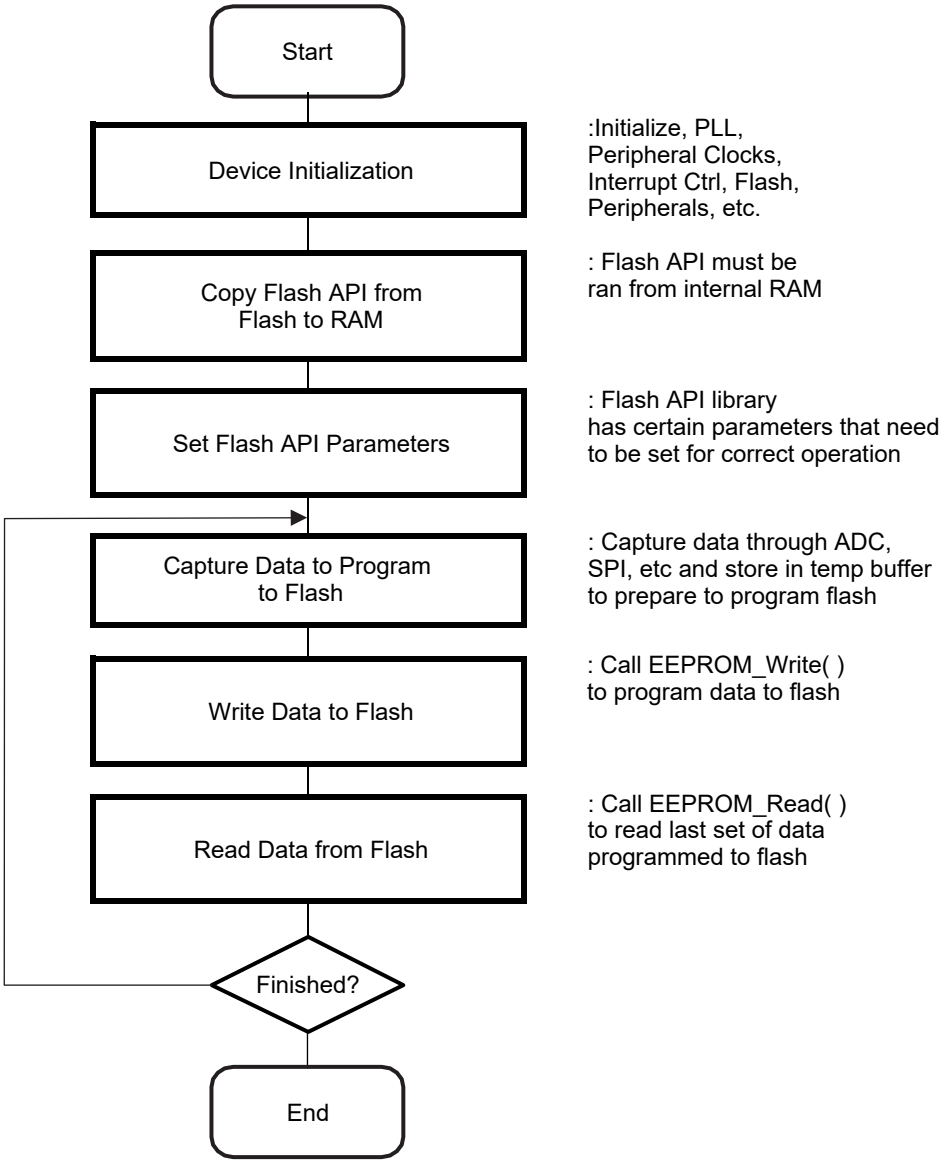

Figure 4. Software Flow

4.2 EEPROM Functions

To implement this functionality, six functions are required to program, read, and erase in multi-byte configuration. Two additional functions are needed for single-byte configuration. All functions are included in the EEPROM.C file for each F28xxx generation. These files are listed in Appendix B.

- EEPROM_Write( )
- EEPROM_Read( )
- EEPROM_Erase( )
- EEPROM_GetValidBank( )
- EEPROM_UpdateBankStatus( )
- EEPROM_UpdatePageStatus( )
- EEPROM_GetSinglePointer(Uint16 First_Call)
- EEPROM_ProgramSingleByte(Uint16 data)

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

4.2.1 EEPROM_Write

The EEPROM_Write( ) function provides functionality for programming data to the Flash. There are a few function calls within this code in order to prepare for data programming. These function calls are shown below:

```
EEPROM_GetValidBank();       // Find In Use Bank and Current Page

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

Each of the above functions are described in detail in later sections. After the current bank and page are found, the page status of the previous page is updated and the bank status is updated if a new bank is being used. Next, the actual programming occurs. The following invokes this process:

```
// Program data located in Write_Buffer to current page
Length = 64;     // Set Length to size of page length for programming
Status = Flash_Program(Page_Pointer+1,Write_Buffer,Length,&ProgStatus);
```

The length must first be set and then the Flash_Program() function is called to invoke the programming process. Four parameters need to be passed to Flash_Program():

- Flash Pointer (Programming Address)
- Buffer containing data to be written
- Length of data (How much to program)
- Status Structure defined by Flash API library

Finally, if the programming is successful, the page status of the current page is updated. The code is shown below:

```
// Modify Page Status from Blank Page to Current Page if Flash programming was successful

if (Status == STATUS_SUCCESS)
{
Page_Status[0] = CURRENT_PAGE;   // Set Page Status to Current Page
  Length = 1;                    // Set Length for programming status
  Status = Flash_Program(Page_Pointer,Page_Status,Length,&ProgStatus);
}
```

4.2.2 EEPROM_Read

The EEPROM_Read( ) function provides functionality for reading the data and storing it into a temporary buffer. This function can be used for debug purposes or to read stored data at runtime.

The current bank and page are found and the contents of the current page are stored in the Read_Buffer:

```
EEPROM_GetValidBank();          // Find In Use Bank and Current Page

// Transfer contents of Current Page to Read Buffer
for(i=0;i<64;i++)
Read Buffer[i] = *(++Page Pointer);
```

4.2.3 EEPROM_Erase

The EEPROM_Erase( ) function provides functionality for erasing the sector used for storage. The entire sector must be erased; partial erase is not supported. Before erasing, you must ensure that stored data is no longer needed/valid. As supplied, this function is only called when all banks and pages are used.

The last sector in the Flash is used for EEPROM functionality. The code below shows the configuration for the F2833x family:

```
#ifdef F28332        // If F28332/F28232 is being used erase Sector D
Status = Flash_Erase((SECTORD),&FlashStatus);
#else                // Otherwise erase Sector H for other devices
Status = Flash_Erase((SECTORH),&FlashStatus);
#endif
```

As supplied, error checking is not included and will need to be added on an application-specific basis. The algorithm currently halts at a breakpoint if erasing is not successful.

```
if(Status != STATUS_SUCCESS)        // If Erase fails halt program or handle error
{
asm(" ESTOP0");
}
```

4.2.4 EEPROM_GetValidBank

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

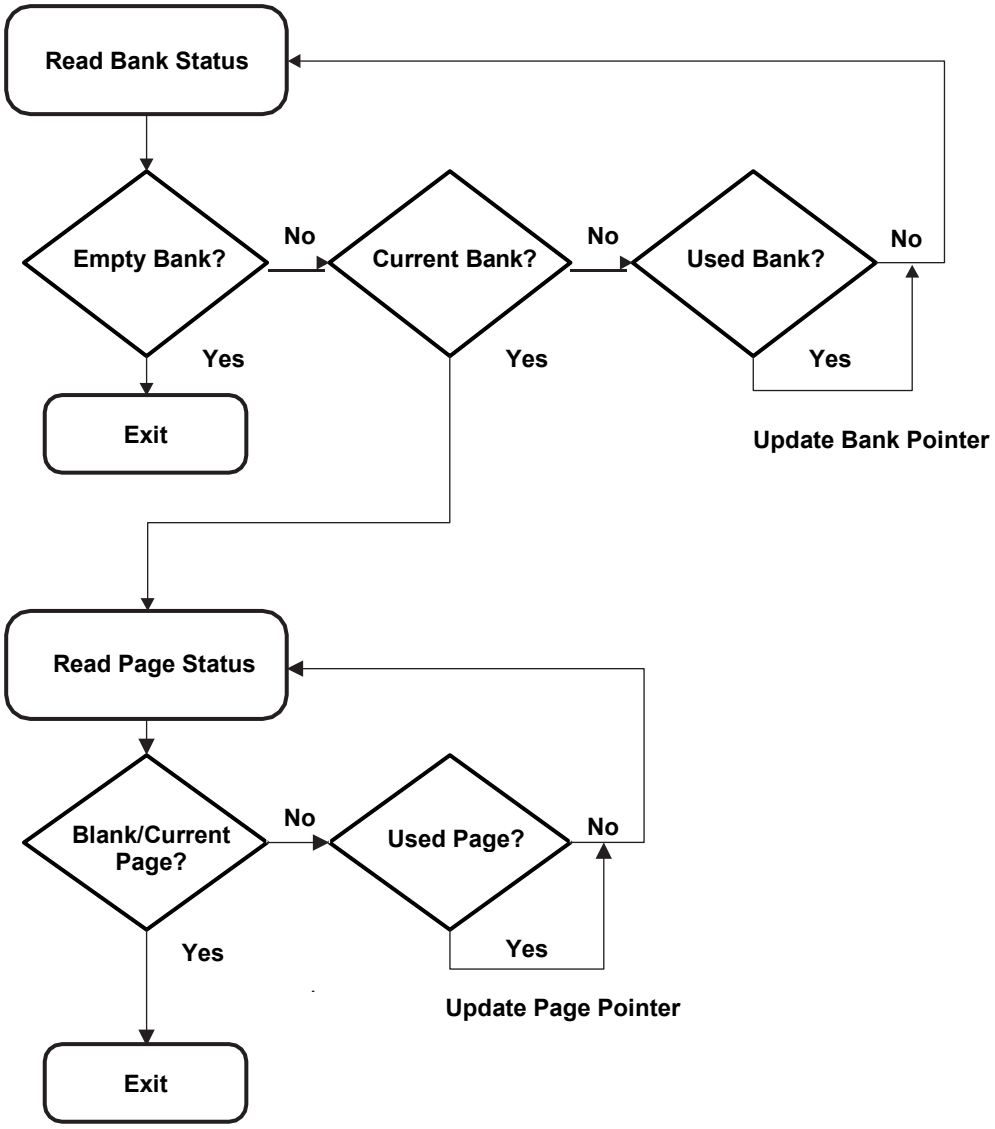

Figure 5. Current Bank and Page Flow

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

```
RESET_BANK_POINTER;   // Reset Bank Pointer to enable search for current Bank
RESET_PAGE_POINTER;   // Reset Page Pointer to enable search for current Page
```

The addresses for these pointers are defined in the EEPROM.h for the specific device being used.

Next, the current bank is found. As Figure 5 shows, there are three different statuses that each bank can have:

- EMPTY_BANK
- USED_BANK
- CURRENT_BANK

EMPTY_BANK status is tested first. If this status is encountered, the 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 EMPTY_BANK is not encountered, CURRENT_BANK status is tested next. If the bank is the current bank, the bank counter is updated and the page pointer is set to the first page of the bank to enable testing for the current page. The loop is then exited as no further bank searching is needed.

```
if(Bank_Status[0] == CURRENT_BANK) // Check for In Use 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 + 1;
        break;                      // Break from loop as current bank has been found
}
```

Lastly, the USED_BANK status is tested. In this case the bank has been used and the bank pointer is updated to the next bank to test its status.

```
if(Bank_Status[0] == USED_BANK)       // Check for Used Bank
       Bank_Pointer += 521;        // If Bank has been used, set pointer to next bank
```

After the current bank has been found, the current page needs to be found. There are three possible statuses that a page can have:

- BLANK_PAGE
- CURRENT_PAGE
- USED_PAGE

The BLANK_PAGE and CURRENT_PAGE statuses 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_Status[0] == CURRENT_PAGE)
{
        Page_Counter = i;          // Set Page Counter to number of current page
        break;                        // Break from loop as current page has been found
}
```

If the page status is neither of these, the only other status available is USED_PAGE. In this case, the page pointer is updated to the next page to test its status.

```
if(Page_Status[0] == USED_PAGE)     // Check for Used Page
        Page_Pointer += 65;         // If page has been used, set pointer to next page
```

At this point, the current bank and page is found and the calling function can continue. As a final step, this function will check if all banks and pages have been used. In this case, the sector needs to be erased. This check is performed by testing the bank and page counters. These counters are set when testing for the current banks and pages as shown in the code snippets above.
If both of these counters are 7, this means the last bank and last page are valid and the memory is full.

```
if (Bank_Counter==7 && Page_Counter==7)     // Check for full EEPROM
{
        EEPROM_Erase();             // Erase Flash sector being used as EEPROM
        RESET_BANK_POINTER;         // Reset Bank Pointer as EEPROM is empty
        RESET_PAGE_POINTER;         // Reset Pank Pointer as EEPROM is empty
        asm(" ESTOP0");
}
```

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

NOTE: The EEPROM_Erase( ) function will erase the entire Flash sector. Any data that needs to be saved should be saved prior to running this function. The EEPROM_Erase( ) call can be removed from the EEPROM_GetValidBank( ) function and can be called at a separate time if erase is not wanted at this point in the application.

As supplied, both the page and bank searches are performed within an 8 count loop as this is the default number of banks used in this implementation. This can be changed on an application-specific basis.

4.2.5 EEPROM_UpdateBankStatus

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

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

If this status indicates the bank is empty, the status is changed to CURRENT_BANK and programmed:

```
// Program Bank Status for Empty EEPROM
if (Bank_Status[0] == EMPTY_BANK)
{
        Bank_Status[0] = CURRENT_BANK;   // Set Bank Status to In Use Bank

        // Program Bank Status to current bank
        Status = Flash_Program(Bank_Pointer,Bank_Status,Length,&ProgStatus);
        Page_Counter =0;

        // Set Page Pointer to first page of current bank
        Page_Pointer = Bank_Pointer + 1;
}
```

If the status is not empty, the next check is for the CURRENT_BANK status with all pages used in the bank (full bank). In this case, the current bank's status will be changed to USED_BANK and the next bank's status will be updated to CURRENT_BANK to allow programming of the next bank. As a last step, the page pointer is updated to the first page of the new bank:

```
// Program Bank Status of full bank and following bank
if (Bank_Status[0] == CURRENT_BANK && Page_Counter == 7)
{
        Bank_Status[0] = USED_BANK;        // Set Bank Status to Used Bank

        // Program Bank Status to full bank
        Status = Flash Program(Bank Pointer,Bank Status,Length,&ProgStatus);

        Bank_Pointer +=521;                // Increment Bank Pointer to next bank

        Bank_Status[0] = CURRENT_BANK;     // Set Bank Status to In Use Bank

        // Program Bank Status to current bank
       Status = Flash_Program(Bank_Pointer,Bank_Status,Length,&ProgStatus);

       Page_Counter =0;

        // Set Page Pointer to first page of current bank
       Page_Pointer = Bank_Pointer +1;
}
```

4.2.6 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.

```
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 USED_PAGE 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
{
         Page Status[0] = USED PAGE;     // Set Page Status to Used Page
         Length = 1;                     // Set Length for programming status
         Status = Flash_Program(Page_Pointer,Page_Status,Length,&ProgStatus);
         Page_Pointer +=65;              // Increment Page Pointer to next page
}
```

4.2.7 EEPROM_GetSinglePointer

The EEPROM_GetSinglePointer( ) function provides functionality for finding the pointer to the first unused location in single byte mode. This function also provides functionality for determining if the full sector has been used. If so, the sector is erased using the EEPROM_Erase( ) function. To begin the end address of the sector is set according to the device being used. The END_OF_SECTOR directive is set in the F28xxx_EEPROM.h file.

```
End_Address = (Uint16 *)END_OF_SECTOR;     // Set End_Address for sector
```

In single byte mode, the first unused location must be found. In this case the application code should call this function before programming. Passing a 1 to this function will set First_Call enabling the code to search for the first unused location. This loop only needs to be ran once to find the valid pointer. After initial execution, 0 should be passed to the function to bypass this loop.

```
if(First_Call == 1)    // If this is first call to function, find valid pointer
{
         RESET_BANK_POINTER;              // Reset Bank Pointer to beginning of sector
         while(*(Bank_Pointer) != 0xFFFF) // Test each location for data
         Bank_Pointer++;                  // Increment to next location
}
```

Next, the bank pointer is compared to end address. If the bank pointer is greater than or equal to the end address, this indicates the sector is full. At this point, the sector is erased and bank pointer is set back to the beginning of the sector.

```
if(Bank_Pointer >= End_Address) // Test if sector is full
{
        EEPROM_Erase();         // Erase Flash sector being used as EEPROM
        RESET_BANK_POINTER;     // Reset Bank Pointer as EEPROM is empty
        asm(" ESTOP0");
}
```

4.2.8 EEPROM_ProgramSingleByte

The EEPROM_ProgramSingleByte( ) function provides functionality for programming a single byte to the memory. The single byte is passed directly from the function to a variable named data. This data is assigned to Write_Buffer to be used by the Flash_Program( ) function. Once programming is complete, EEPROM_GetSinglePointer( ) is called to determine if the sector is full and needs to be erased. Zero is passed to this function as Bank_Pointer does not need to be determined. This is discussed in Section 4.2.7.

```
Write_Buffer[0] = data;          // Prepare data to be programmed

Length = 1;                      // Set Length for programming
Status = Flash_Program(Bank_Pointer++,Write_Buffer,Length,&ProgStatus);
EEPROM_GetSinglePointer(0);      // Test for full sector
```

NOTE: This function cannot be used until EEPROM_GetSinglePointer() function has been called to find the valid pointer location. Executing before would produce unknown results.

4.3 Testing Example

The examples provided were tested with the F2812, F2808, and F28335 eZdsp development boards. To properly test the example, the memory window and breakpoints need to be utilized within the Code Composer Studio. The following steps were followed to program and test the project with the F28335 eZdsp. The same procedure can be used for the other eZdsp development boards as well.

1. Connect the F28335 eZdsp to the PC using the on-board USB connection; power the board with the supplied power connector.
2. Start Code Composer Studio with the F28335 eZdsp emulation driver selected in the Code Composer Studio setup utility.
3. Open the F2833x_EEPROM_Example.pjt by selecting Project → Open.
4. Select the device being used in the project configurations as shown in Figure 6.

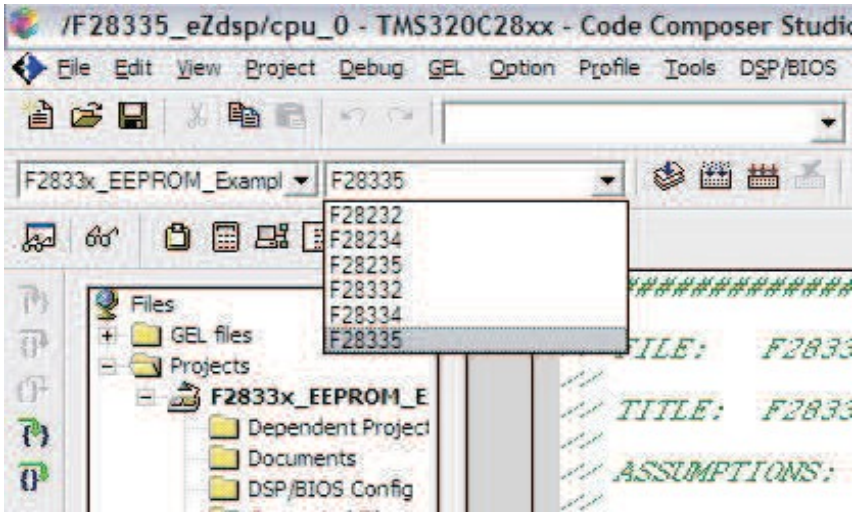

Figure 6. Device Selection

5. Build the project by selecting Project → Rebuild All.
6. Program the resulting .out file to the Flash by using the Code Composer Studio On-Chip Flash Programmer from the Tools menu. If this is currently not installed, it can be downloaded from the Update Advisor.
7. Load Symbols to debug the program by selecting File → Load Symbols → Load Symbols Only.
8. Enter the main() function by selecting Debug → Go Main once the project is loaded.

9. Set breakpoints to properly view data written to and read from the memory within the memory window as shown Figure 7.

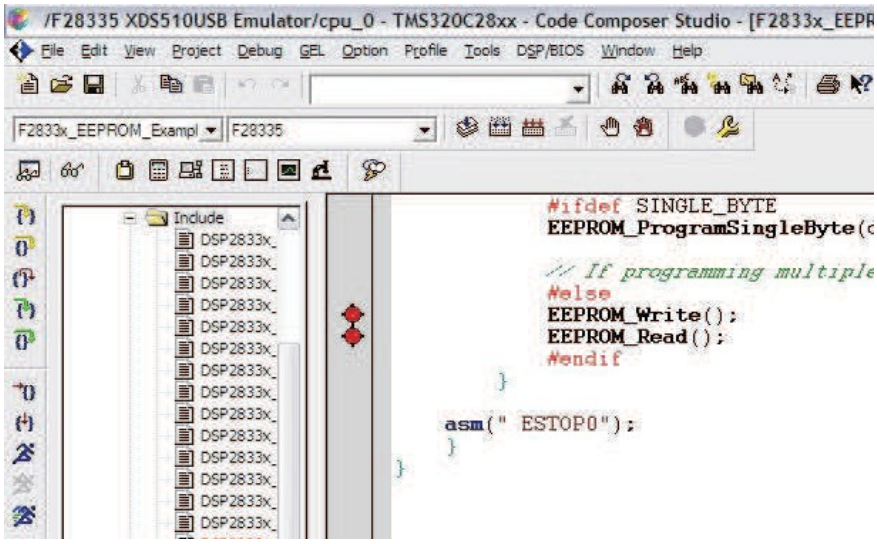

Figure 7. Breakpoints

10. Run to the first breakpoint and then open the Memory Window (View → Memory) 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 Figure 8.

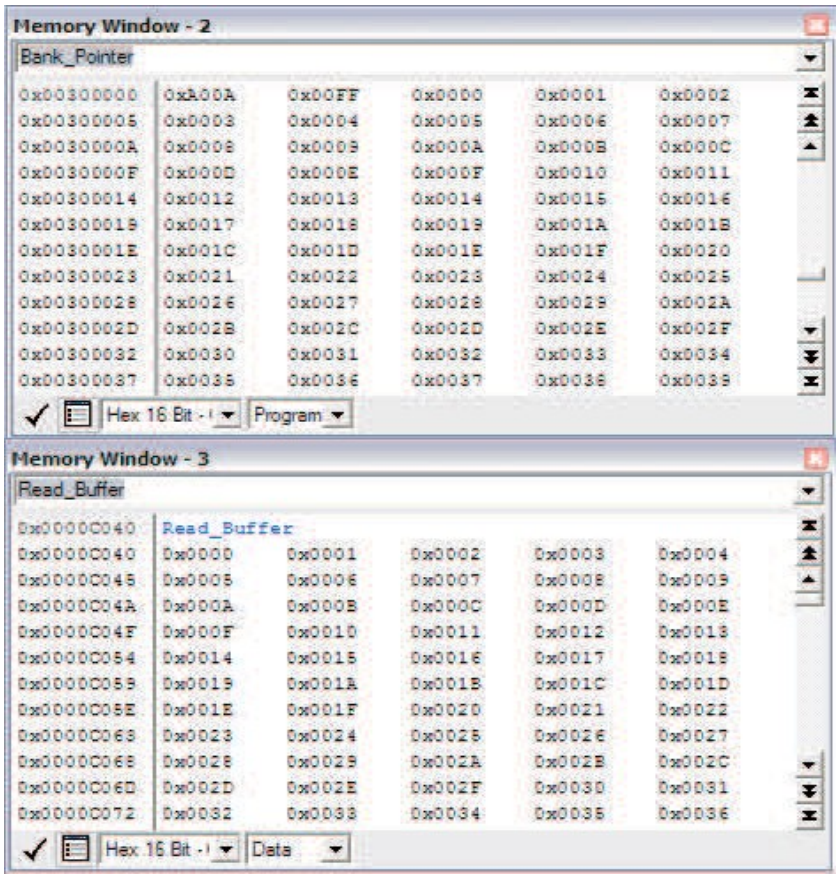

Figure 8. Memory Window

11. Continue running from breakpoint to breakpoint watching each write until the program has finished.

The preceding steps were used to test the multi-byte configuration. The single-byte configuration can also be tested with the same procedure. The only step left out would be the read procedure as this is not included in single-byte mode. To enable single-byte mode, change the definition in the F28xxx_EEPROM.h file by uncommenting the SINGLE_BYTE directive and commenting the MULTI_BYTE directive:

```
// Project specific defines
#define SINGLE_BYTE 1
//#define MULTI_BYTE  1
```

4.4 Application Integration

Applications needing this functionality need to include the EEPROM.c and EEPROM.h files provided for each F28xxx generation. The Flash API also needs to be included for the specific device. For example, for the TMS320F28335 the following files are needed:

- F2833x_EEPROM.c
- F2833x_EEPROM.h
- Flash2833x_API_Config.h
- Flash2833x_API_Library.h
- Flash28335_API_V210.lib

All Flash API files are included with the library download. See Appendix A for more details on Flash API. Once these files are included the software flow and functionality described above can be followed to ensure proper operation.

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.

NOTE: Inclusion of EEPROM.C and EEPROM.h adds 356 words of program and 192 words of data to the overall code size.

5 Conclusion

This application report has proven that the TMS320F28xxx DSC is capable of utilizing its internal Flash to emulate an 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.

6 References

1. *TMS320F28x Fixed-Point Flash APIs*:
   (a) *TMS320F2801x Flash APIs* (SPRC327)
   (b) *TMS320F2804x Flash APIs* (SPRC325)
   (c) *TMS320F280x Flash APIs* (SPRC193)
   (d) *TMS320F2810, TMS320F2811 and TMS320F2812 Flash API* (SPRC125)
2. *TMS320F28x Floating Point Flash APIs*:
   (a) *TMS320F2833x Flash APIs* (SPRC539)

3. TMS320C28x Header Files:
   (a) *F2803x (Piccolo) C/C++ Header Files and Peripheral Examples* (SPRC892)
   (b) *C2834x (Delfino) C/C++ Header Files and Peripheral Examples* (SPRC876)
   (c) *F2802x (Piccolo) C/C++ Header Files and Peripheral Examples* (SPRC832)
   (d) *F2833x / F2823x C/C++ Header Files and Peripheral Examples* (SPRC530)
   (e) *C281x C/C++ Header Files and Peripheral Examples* (SPRC097)
   (f) *C280x C/C++ Header Files and Peripheral Examples* (SPRC191)
   (g) *C2804x C/C++ Header Files and Peripheral Examples* (SPRC324)
4. *Flash Programming Solutions for the TMS320F28xxx DSCs* (SPRAAL3)
5. *Running an Application From Internal Flash Memory on the TMS320F28xx DSP* (SPRA958)

Appendix A Flash API

A.1 Description

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 Flash API erase function includes the Flash pre-conditioning and a separate *clear* step is not required. The program function operates on both the Flash array and the OTP block. The program function can only change bits from a 1 to a 0. 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.

This section gives a small overview of the Flash API. For complete details on all options and operating procedures of the Flash API library, see the API documentation included in the API zip file [1], [2]. There is also an application report available that discusses all flashing options and the flashing procedure, [4].

A.2 Flash API Checklist

This following data is taken from the API document as a reference and for completeness.

Integration of the Flash APIs into user software requires that the system designer implement operations to satisfy several key requirements. The following checklist gives an overview of the steps required to use the API. These steps are further discussed in detail in the reference section indicated. This checklist applies to all of the F2823x Flash APIs and was taken from the F2823x Flash API documentation. This general checklist applies to all F28xxx Flash API Libraries and can be found in each library's respective documentation [1], [2].

Before using the API, do the following:

1. Modify Flash2823x_API_Config.h for your target operating conditions.
2. Include Flash2823x_API_Library.h in your source code.
3. Add the proper Flash API library to your project.

   When using the Flash API, build your code with the large memory model. The API Library is built in 28x Object code (OBJMODE = 1, AMODE = 1).

In your application, before calling any Flash API functions do the following:

1. Initialize the PLL Control Register (PLLCR) and wait for the PLL to lock.
2. Make sure the PLL is not running in limp mode. If the PLL is in limp mode, do not call any of the API functions as the device will not be running at the proper frequency.
3. Optional: The API must execute from zero-wait state internal SARAM. If the API is to be copied from Flash/OTP into internal SARAM memory then follow the instructions in this section.
4. Initialize the 32-bit global variable Flash_CPUScaleFactor.
5. Initialize the global function pointer Flash_CallbackPtr to point to the application's callback function. Alternatively, set the pointer to NULL.
6. Optional: Disable global interrupts before calling an API function.
7. Understand the API restrictions detailed in this section before making any APIcalls.
8. Optional: Run the frequency toggle test to confirm proper frequency configuration of the Flash API. Note: The ToggleTest function will execute forever. You must halt the processor to stop this test.
9. Optional: Unlock the code security module (CSM).
10. Call the Flash API Functions described in the API Reference.

The called Flash API function does the following:

- The watchdog timer is disabled.
- Checks the PARTID (memory location 0x0882) register to make sure the part is the correct device
- Checks the contents of 0x3FFFB9 in the boot ROM for API version vs. silicon revision compatibility.
  - Performs the called operation and:
    - Disables and restores global interrupts (via INTM, DBGM, XNMICR) around time critical code segments.
    - Invokes the callback function if Flash_CallbackPtr is not NULL.
  - Returns success or an error code. These are defined in F2823x_API_Library.h.

The user's code should then do the following:

- Check the return status against the error codes.
- Optional: Re-enable the watchdog timer.

A.3 Flash API Do's and Dont's

A.3.1 API Do's

- Execute the Flash API code from zero-wait state internal SARAM memory. The F2823x and F2833x devices contain both zero wait state (L0-L3) and one wait state SARAM (L4-L7). The Flash APIs should be run from L0-L3 SARAM.
- Configure the API for the correct CPU frequency of operation.
- Follow the Flash API checklist in Section A.2.0 to integrate the API into an application.
- Initialize the PLL control register (PLLCR) and wait for the PLL to lock before calling an API function.
- Initialize the API callback function pointer (Flash_CallbackPtr). If the callback function is not going to be used then it is best to explicitly set the function pointer to NULL. Failure to initialize the callback function pointer can cause the code to branch to an undefined location.
- Carefully review the API restrictions for the callback function, interrupts, and watchdog described in the document.

A.3.2 API Don'ts

- Don't execute the Flash APIs from the Flash or OTP. If the APIs are stored in Flash or OTP memory, they must first be copied to internal zero-wait state SARAM before they are executed.
- Don't execute any interrupt service routines (ISRs) that can occur during an erase, program or depletion recovery API function from the Flash or OTP memory blocks. Until the API function completes and exits the Flash and OTP are not available for program execution or data storage.
- Don't execute the API callback function from Flash or OTP. When the callback function is invoked by the API during the erase, program or depletion recovery routine the Flash and OTP are not available for program execution or data storage. Only after the API function completes and exits will the Flash and OTP be available.
- Don't stop the erase, program or depletion recovery functions while they are executing (for example, don't stop the debugger within API code, don't reset the part, etc.).
- Do not execute code or fetch data from the Flash array or OTP while the Flash and/or OTP is being erased, programmed or during depletion recovery.

Appendix B Source File Listing

Table 1. Table 1. Function Listing

File	Function	Description
F280xx_EEPROM.c	EEPROM_Write()	Performs write operation
F281x_EEPROM.c	EEPROM_Read()	Performs read operation
F28335_EEPROM.c	EEPROM_Erase()	Erases Flash sector
	EEPROM_GetValidBank()	Finds valid bank and page
	EEPROM_UpdateBankStatus()	Updates bank status
	EEPROM_UpdatePageStatus()	Updates page status
	EEPROM_GetSinglePointer(Uint16 First_Call)	Finds valid pointer for single byte operation and test for full sector
	EEPROM_ProgramSingleByte(Uint16 data)	Programs single byte to sector
F280xx_EEPROM.h		Contains Function prototypes, Includes
F281x_EEPROM.h		Flash API headers, global variables
F2833x_EEPROM.h		Pointer initialization, status definitions

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265