参考文献

Application Note

第 3 代 C2000 实时控制器的 EEPROM 仿真

Vamsikrishna Gudivada, Skyler Baumer, Charles Roberson

摘要

许多应用都需要在非易失性存储器中存储少量系统相关数据（校准值、器件配置），这样即使在系统下电上电后，也可以使用或修改并重复使用这些数据。电可擦除可编程只读存储器 (EEPROM) 主要用于此目的。EEPROM 能多次反复擦除和写入存储器的各个字节，即使在系统断电后，编程位置也能长时间保存数据。本应用报告和相关代码有助于将片上闪存存储器的一个或多个扇区定义为仿真 EEPROM，并由应用程序用于透明地写入、读取和修改数据。

本应用报告中讨论的工程配套资料和源代码可以在 C2000Ware v5.02.00.00（或更高版本）中找到，路径如下：C2000Ware_5_02_00_00/driverlib/f28p65x/examples/c28x/flash/，C2000Ware_5_02_00_00/driverlib/f28003x/examples/flash/

内容

插图清单

商标

1 简介

第 3 代 C2000 MCU 具有以多个扇区形式排列的不同闪存存储器配置。遗憾的是，片上闪存存储器所使用的技术不允许在芯片上添加传统的 EEPROM。因此，一些设计人员使用外部 EEPROM 器件来提供此类非易失性存储。好消息是闪存存储器是一种特定类型的 EEPROM，所有第 3 代 C2000 MCU 都具有闪存存储器的在线编程功能。本应用报告将采用此功能，通过在闪存存储器内仿真 EEPROM 功能，将片上闪存的扇区用作 EEPROM。请注意，至少一个完整的闪存扇区会作为仿真型 EEPROM；因此，该扇区不可用于存储应用程序代码。

备注

第 3 代 C2000 MCU 包括：TMS320F2837x、TMS320F2838x、TMS320F28004x、TMS320F28002x、TMS320F28003x、TMS320F280013x、TMS320F280015x 和 TMS320F28P65x。

2 EEPROM 与片上闪存的区别

EEPROM 具有各种不同的容量，并通过串行接口（有时为并行接口）与主机微控制器连接。由于引脚/布线数量超少，串行内部集成电路 (I2C) 和串行外设接口 (SPI) 颇受欢迎。EEPROM 可以进行电编程和擦除，并且大多数串行 EEPROM 支持逐字节编程或擦除操作。

EEPROM 与闪存操作的主要区别就在于擦除操作。EEPROM 不需要扇区擦除操作。用户可以擦除需要指定时间的特定字节。然而，闪存中擦除操作的最小单位是一个扇区。

闪存擦除/写入周期是通过对各个存储单元施加时控电压来执行的。在擦除情况下，每个存储单元（位）读取逻辑 1。因此，被擦除时，C2000 实时控制器的每个闪存位置读取 0xFFFF。通过编程，存储单元可以更改为逻辑 0。任何字都可以被覆盖，将一个位从逻辑 1 更改为 0（假设相应的 ECC 尚未编程）；但无法执行相反的操作。第 3 代 C2000 MCU 器件的片上闪存需要使用 TI 提供的特定算法（闪存 API）来执行擦除和写入操作。

备注

如需了解闪存擦除/编程/读取时间，请参阅特定器件数据手册“电气特性”中的“闪存参数”部分。

3 概述

本文档中所述的实现支持单存储单元和乒乓 EEPROM 仿真。单存储单元和乒乓实现均支持两种模式：页编程模式和 64 位编程模式。首先介绍乒乓仿真，然后介绍单存储单元仿真。

该实现支持多个用户可配置的 EEPROM 变量。节 5.1 详细介绍了这些变量。

3.1 基本概念

在该实现中，仿真 EEPROM 至少包含一个闪存扇区。由于闪存的块擦除要求，必须为 EEPROM 仿真保留一个完整的闪存扇区。根据 C2000 器件型号，闪存扇区的大小会有所不同。闪存扇区的区域被分为许多较小的部分，称为页面。例如，一个 2K x 16 的闪存扇区可以分成 32 个页面，每个页面大小为 64 x 16。

要保存的数据会首先写入 RAM 中的缓冲区。然后，借助第 3 代 C2000 MCU 的电路内编程功能，数据会被写入选择用于 EEPROM 仿真的扇区的第一个页面。下次将数据写入闪存时，数据将被写入下一个页面。该过程会一直持续，直到所选扇区的最后一个页面被写入数据为止。到达最后一个页面后，有两种方法可以继续。如果使用单存储单元 EEPROM 仿真，请参阅单存储单元仿真行为以了解如何处理。如果使用乒乓 EEPROM 仿真，请参阅乒乓方法以了解如何处理。

除上述描述的页面编程概念之外，还支持 64 位编程。在此模式下，扇区不会被分成 EEPROM 组和页面。节 5.2.10 和节 5.2.11 将进一步讨论 64 位编程。

3.2 单存储单元方法

如果使用单存储单元 EEPROM 仿真，图 3-1 展示了实现的行为。

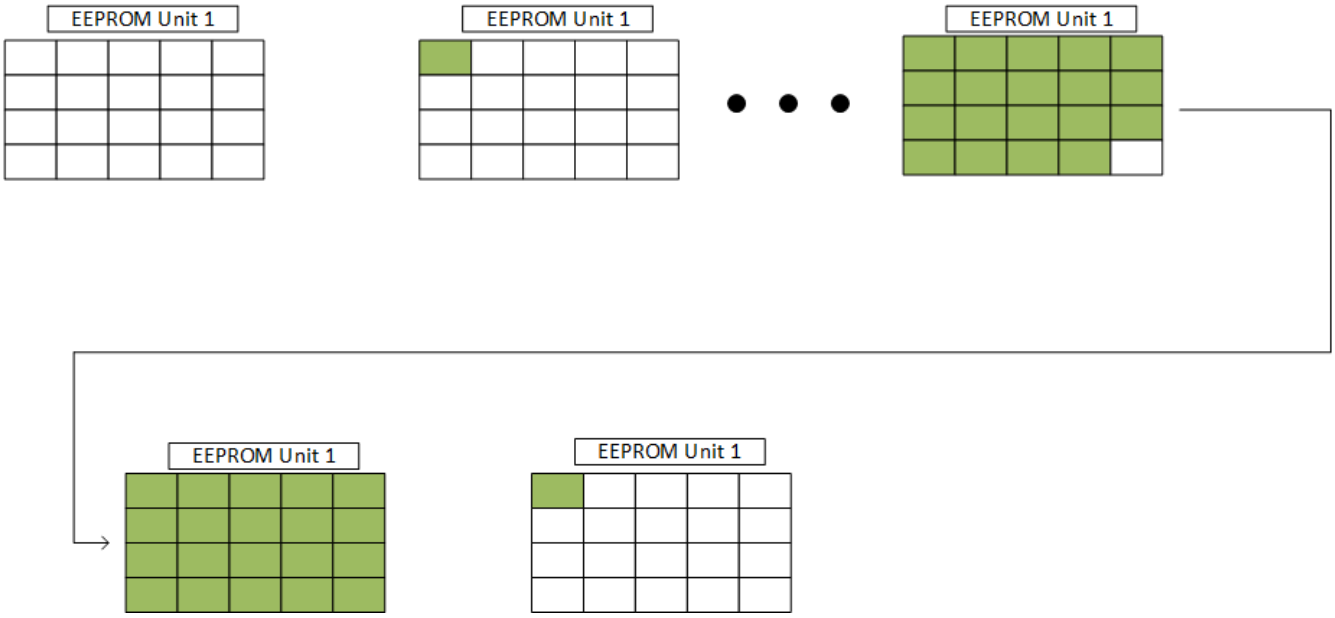

图 3-1. 单存储单元行为

如果 EEPROM 单元已满并且还有更多数据要写入，则擦除 EEPROM 单元并将新数据编程到闪存中。可以根据需要重复该过程。

3.3 乒乓方法

如果使用乒乓方法，下图展示了实现的行为。如图 3-2 所示，有两个由选定的闪存扇区组成的 EEPROM 单元。一个被标记为活动单元，另一个被标记为非活动单元。首先，数据被写入活动单元。

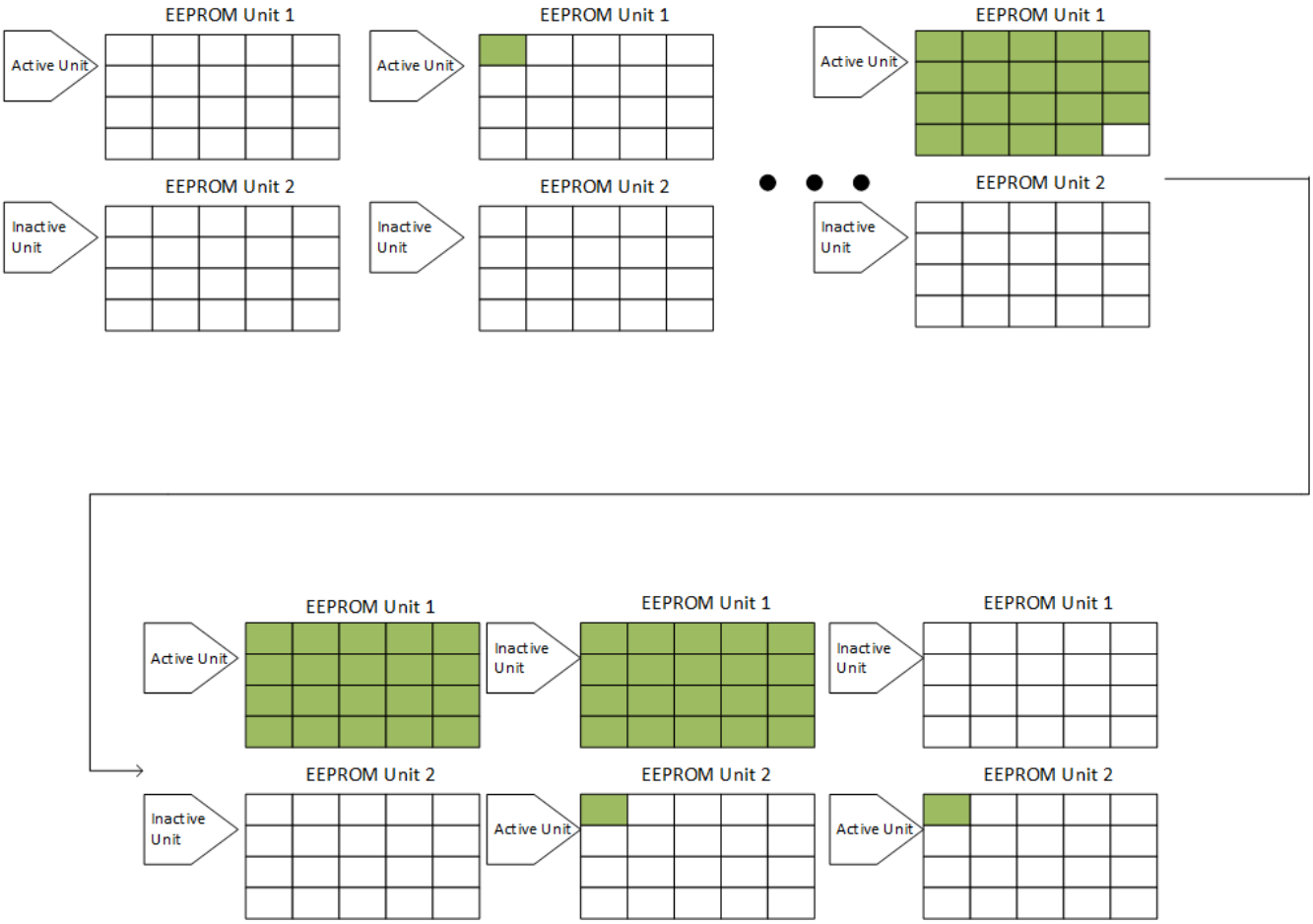

图 3-2. 乒乓行为

如果活动单元已满并且有更多数据要写入，则活动和非活动 EEPROM 单元将切换。因此，之前的活动单元（已满单元）将被标记为非活动，之前的非活动单元（空单元）将被标记为活动。随后，数据将被写入新的活动 EEPROM 单元。数据成功编程到活动 EEPROM 单元后，非活动 EEPROM 单元被擦除。该方法可确保当当前活动的 EEPROM 单元已满时，在擦除或编程操作期间发生任何失败时，最后成功写入的数据有一个回退选项。

可以根据需要重复该过程。

3.4 创建 EEPROM 节（页）和页标识

为了支持具有不同数据大小（64 位除外）的 EEPROM 仿真，选择用于仿真的闪存扇区被分为称为 EEPROM 组（不要与闪存组相混淆）和页面的格式。首先，闪存扇区（或选定的闪存扇区集）被分为 EEPROM 组。每个 EEPROM 组又被进一步分成页面。组分区展示了该分区。EEPROM 组和页面的分区对于单存储单元和乒乓仿真是相同的。

使用该格式，应用能够：

- 从在之前保存期间写入的页面中读回数据
- 将最新数据写入新页面
- 在应用需要时，从之前存储的任何数据读取

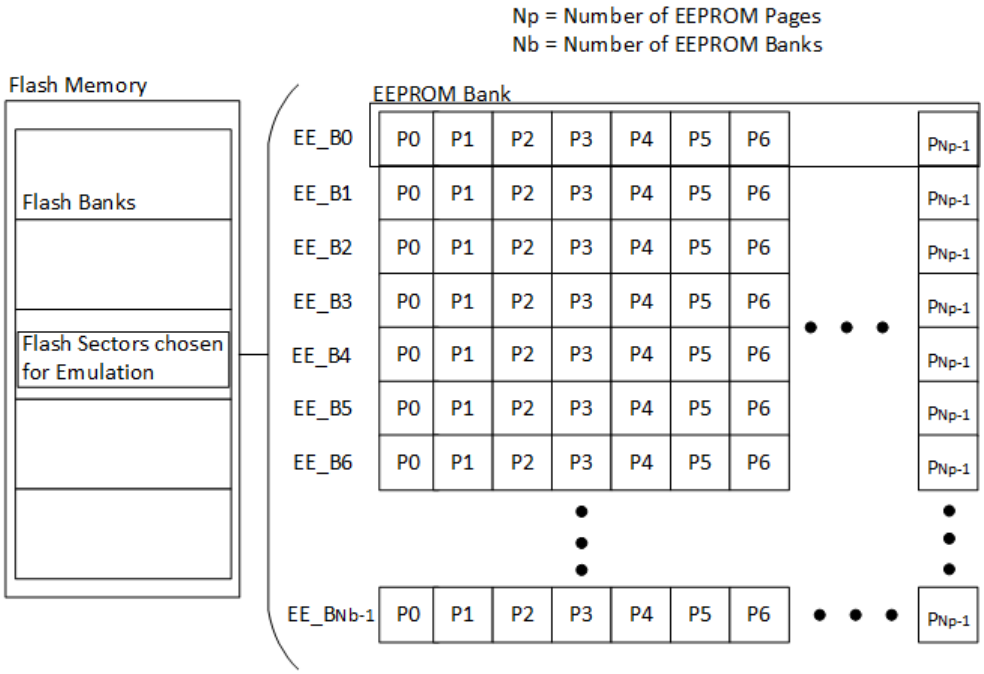

图 3-3. 组分区

每个 EEPROM 组的前 8 个 16 位字保留用于存储 EEPROM 组状态信息，每个页面的前 8 个字保留用于存储页面状态信息。每次向页写入一组新的数据时，都会修改最后一页和下一页的状态位置。使用新 EEPROM 组时，最后一个和当前 EEPROM 组的组状态将被更新。EEPROM 组和页面状态字以相同的方式区分当前 EEPROM 组/页面和已使用的 EEPROM 组/页面。要将 EEPROM 组或页面标记为当前组或页面，可以向前 64 位写入适当的状态代码。要将 EEPROM 组或页面标记为已满组或页面，可以向后 64 位写入适当的状态代码。有关状态代码的更多详细信息，请参阅 EEPROM_GetValidBank。

如页面布局中所示，所有页面都包含八字页面状态和可配置的数据空间量。第 0 页略有不同，因为该页面还包含 EEPROM 组状态。仅显示了第 0 页和第 1 页，但应注意第 2 页至第 (N-1) 页与第 1 页相同。

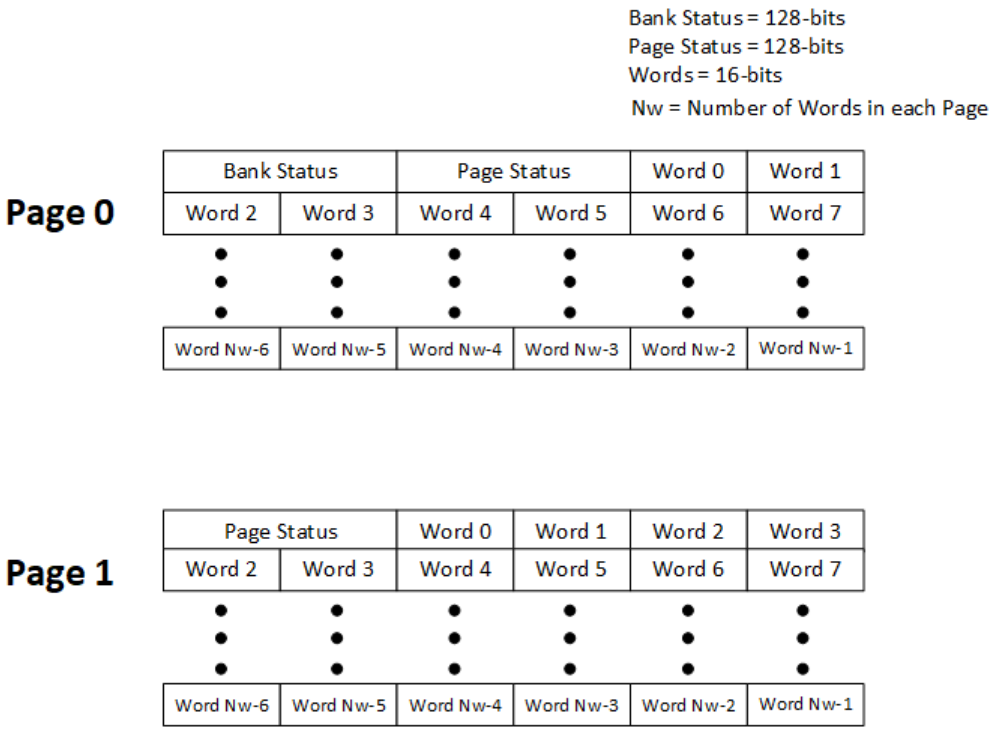

图 3-4. 页布局

4 软件说明

本应用报告随附的软件包含用于 F28P65x 第 3 代 C2000 实时控制器的 EEPROM 仿真源代码，以及一个演示如何利用源代码的示例工程。代码的某些方面可能因所使用的器件而异。有关示例，请参阅 适应其他第 3 代 C2000 MCU。本指南的其余部分是为 F28P650DK9 器件设计的，并在适用的情况下重点突出与 F280039C 器件的比较。

此软件提供了基本的 EEPROM 功能：写入、读取和擦除。闪存存储器的至少一个扇区用于仿真 EEPROM。如上所述，该（这些）扇区被分成多个 EEPROM 组和页面，每个组和页面均包含状态字，用于确定数据的有效性。

该代码使用为 F28P65x 提供的头文件和闪存 API 库。可以在 C2000Ware 目录中找到示例代码。完整路径为：C2000Ware_5_02_00_00/driverlib/f28p65x/examples/c28x/flash

为了与 F28003x 器件进行比较，可以在 C2000Ware 目录中找到示例代码。完整路径为：C2000Ware_5_02_00_00/driverlib/f28003x/examples/flash

4.1 软件功能和流程

器件必须首先执行其初始化代码来初始化时钟、外设等。使用的初始化函数是工程中包含的头库文件提供的函数。有关此序列的更多信息，请参阅头文件随附的文档。

一旦操作完成，闪存 API 初始化和参数便已设置完毕，可随时进行闪存编程。闪存 API 库需要几个文件和一些初始化/设置，才能正常工作。可以在 F28P65x 闪存 API 参考指南中找到所需步骤的完整列表。

接下来，将检查用户在 EEPROM_Config.h 中指定的 EEPROM 配置的有效性，并配置闪存 API 使用的某些变量。有关更多详细信息，请参阅用户配置和节 5.2.1。

此时，可以开始编程了。首先，需要捕捉数据才能进行编程。在完成此数据编程后，读取功能会读取编程到闪存中的最后一组数据。大多数应用都应该遵循此软件流程，尤其是初始化部分，因为需要先将某些闪存 API 函数复制到内部 RAM 中才能开始编程。

提供的示例工程遵循软件流程中所示的软件流程。要了解有关图中所示函数的更多信息，请导航至文档中的相应部分。

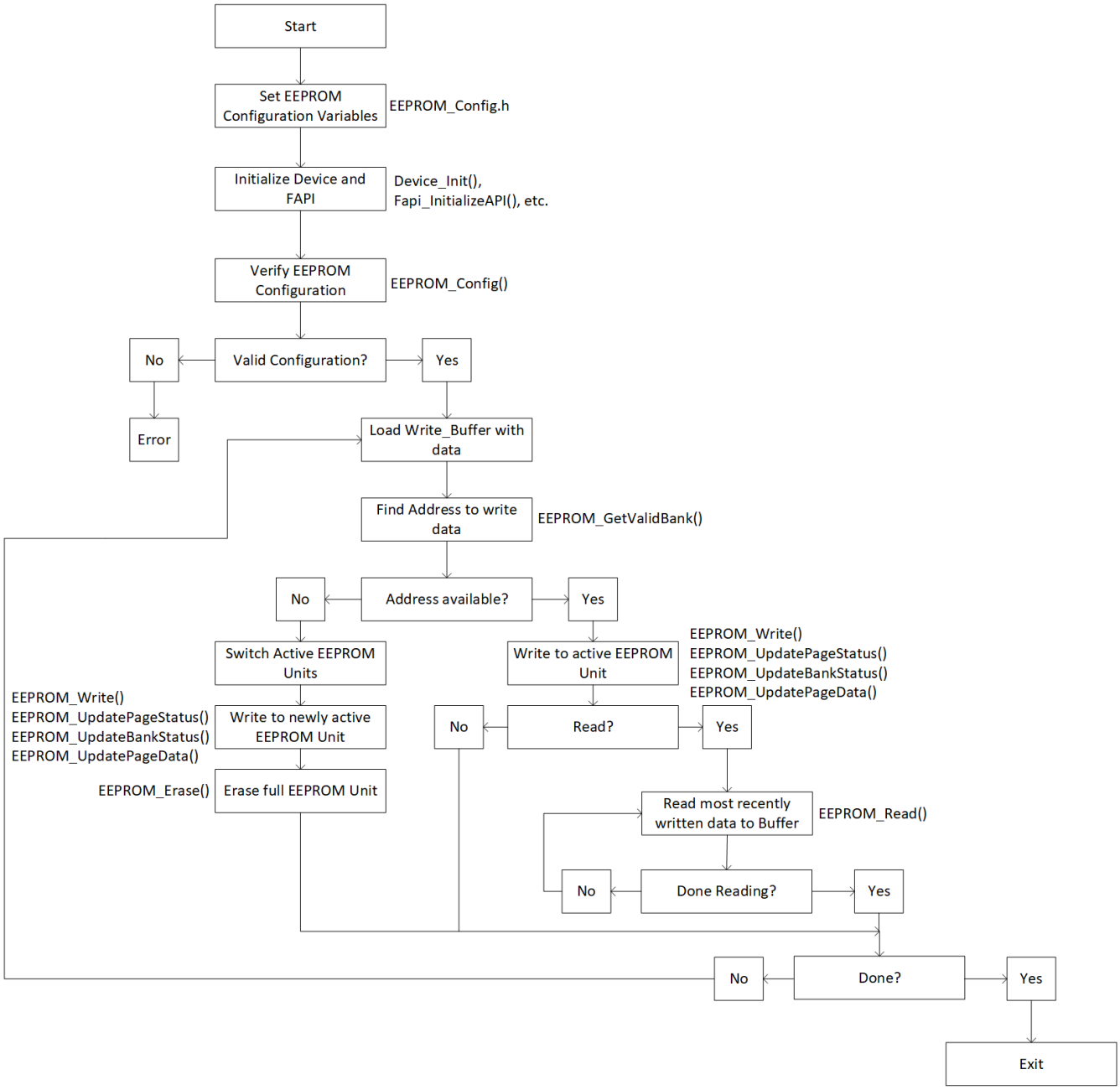

图 4-1. 软件流程

5 乒乓仿真

本节讨论乒乓实现。要查看该实现的行为，请参阅乒乓行为。

5.1 用户配置

本文档中详细介绍的实现允许用户为 EEPROM 仿真配置多个变量。这些变量主要位于 EEPROM_PingPong_Config.h 中，但有两个变量包含在 F28P65x_EEPROM_PingPong.c 中。将在下面专门讨论相应文件的部分中讨论可配置变量。

5.1.1 EEPROM_PingPong_Config.h

EEPROM_PingPong_Config.h 包含允许用户更改 EEPROM 配置各个方面的定义。这些方面包括：

- 定义正在使用的器件型号。这允许在并非所有器件都通用的闪存组中进行 EEPROM 仿真。

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

- 在页面模式和 64 位模式之间进行选择。

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

- 选择要用于仿真的闪存组。**默认情况下，闪存 API 和程序从闪存组 0 中存储/运行，因此该闪存组无法用于 EEPROM 仿真。通常，闪存 API 和程序应该从与用于 EEPROM 仿真的闪存组不同的闪存组中存储/运行。**

```
#define FLASH_BANK_SELECT FlashBank1StartAddress
```

- 定义闪存扇区大小（单位为 16 位字）。这会因使用的器件而异，请参阅相应的数据表了解详细信息。

```
#define FLASH_SECTOR_SIZE F28P65x_FLASH_SECTOR_SIZE
```

- 定义一个闪存组中有多少个闪存扇区。这会因使用的器件而异，请参阅相应的数据表了解详细信息。

```
#define NUM_FLASH_SECTORS F28P65x_NUM_FLASH_SECTORS
```

- 选择要仿真的 EEPROM 组的数量。

```
#define NUM_EEPROM_BANKS 4
```

- 选择每个 EEPROM 组中有多少个 EEPROM 页面

```
#define NUM_EEPROM_PAGES 3
```

- 选择每个 EEPROM 页面中包含的数据空间大小（单位为 16 位字）。尽管可以指定任何大小，但该大小将调整为大于或等于指定大小的最接近的四的倍数。例如，每页 6 个 16 位字的指定大小将被编程为每页 8 个 16 位字，最后两个字被视为 0xFFFF。这是为了符合闪存要求（为每个 64 位对齐的闪存存储器地址进行 8 位 ECC 编程）。

```
#define DATA_SIZE 64
```

5.1.2 F28P65x_EEPROM_PingPong.c

在 F28P65x_EEPROM_PingPong.c 中，用户可以选择用于 EEPROM 仿真的闪存扇区。选择的扇区（如果有多个）应该是连续的并且按从小到大的顺序排列。仅插入要用于 EEPROM 的第一个和最后一个扇区。例如，要使用扇区 1-10，请插入 {1,10}。要仅使用扇区 1，请插入 {1,1}。有效的配置具有以下属性。

- 意味着两个 EEPROM 单元之间的扇区数量有效且一致

- 仅包含器件上存在的扇区
- 不会在两个单元之间的写入/擦除保护掩码中产生重叠
  - F28P65x 闪存 API 要求在对闪存存储器进行编程之前配置写入/擦除保护掩码。有关这些掩码正确配置的详细信息，请参阅 *F28P65x 闪存 API 参考指南*。

有关无效或危险配置的更多详细信息，请参阅节 5.2.1。

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

此外，您还可以选择在哪组闪存扇区中开始模拟。

```
uint16 EEPROM_ACTIVE_UNIT = 0;
```

如果设置为 0，则 FIRST_AND_LAST_SECTOR 中的第一组闪存扇区将首先是活动 EEPROM 单元，第二组将首先是非活动 EEPROM 单元。如果设置为 1，则情况相反。

5.2 EEPROM 函数

为了实现该功能，需要使用 12 个函数在页面编程中执行配置、编程、读取和擦除。64 位编程需要两个附加函数。所有函数均包含在 F28P65x_EEPROM.c 或 F28P65x_EEPROM_PingPong.c 文件中。

- 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()

后续小节中会详细介绍上述每个函数。

5.2.1 EEPROM_Config_Check

EEPROM_Config_Check() 函数提供一般错误检查并配置闪存 API 所需的写入/擦除保护掩码。应在对仿真 EEPROM 单元进行编程或读取之前调用此函数。

第一，该函数验证选择用于 EEPROM 仿真的闪存组是否有效。有效的闪存组选择不得选择组 0 进行仿真，并且必须受特定器件型号的支持。例如，只有某些 F28p65x 型号具有闪存组 2-4，而其他型号没有。要验证该信息，请参阅特定于器件的数据表。

```
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
}
```

第二，检查选择用于仿真的闪存扇区的有效性。该函数检查：

- FIRST_AND_LAST_SECTOR 是否指示两个单元具有两个不同数量的闪存扇区

```
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;
}
```

- 选择用于仿真的闪存扇区数量是否多于闪存组中可用的扇区数量

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

- 选择用于仿真的第一个和最后一个扇区的组合是否无效

```
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;
    }
}
```

- 两个单元之间是否存在重叠扇区
- 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; }

如果使用页面模式，还将检查以下各项

- 检查 EEPROM 组 + 页面的总大小是否适合所选的闪存扇区。

```
// 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; }
```

- 验证两个 EEPROM 单元是否没有重叠的保护掩码

```
// 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;
}
```

如果检测到以下情况之一，该函数还会通过相应的返回代码发出警告：

- 配置 EEPROM 组和页面大小后，闪存中会保留一个或多个 EEPROM 组的空间

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

- 如果每个页面包含少于 5 个 16 位字（这会浪费空间，因为无需状态代码即可使用 64 位模式）

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

如果使用 32-127 范围内的扇区（对于 F28P65x 器件）并且未使用分配给写入/擦除保护掩码中单个位的全部八个扇区，则会发出警告。由 single-bit 设计的八个扇区中任何未使用的扇区都无法受到适当的擦除保护。有关写入/擦除保护掩码如何与扇区对应的更多信息，请参阅 *TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南*。

```
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;
            }
        }
    }
}
```

该函数还通过擦除要用于编程的扇区来为仿真准备好闪存。

```
    // 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();
```

最后，为活动 EEPROM 单元配置写入/擦除保护掩码。

```
// 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

Configure_Protection_Masks 提供了为选择用于 EEPROM 仿真的任何扇区禁用写入/擦除保护的功能。这是通过计算要传递到 Fapi_setupBankSectorEnable 函数的适当掩码来完成的。该函数需要两个参数，即指向所选闪存扇

区编号的指针和要仿真的闪存扇区的数量。有关 Fapi_setupBankSectorEnable 函数实现的更多详细信息，请参阅 *TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南*。

该函数的返回值将用于禁用为 EEPROM 仿真选择的闪存扇区中的写入/擦除保护。

```
// 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;
```

为了进行比较，F28003x EEPROM Ping Pong 示例工程 Configure_Protection_Masks 的功能与 F28P65x EEPROM PingPong 示例工程的功能不同，后者有一定量的扇区可用于保护。写入/擦除保护掩码中的每个位表示它自己的扇区。

```
// 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

EEPROM_Write() 函数的功能是将数据编程写入闪存。该函数直接利用闪存 API 并在其中进行多个函数调用以准备数据编程。下面列出了调用的函数：

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

相应的各节会详细介绍上述每个函数。首先，找到当前的 EEPROM 组和页面。找到当前 EEPROM 组和页面后，便会更新上一个页面的页面状态，如果要使用新 EEPROM 组，则会更新 EEPROM 组状态。接下来，在 EEPROM 页数据更新期间进行实际编程。

```
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

EEPROM_Read() 函数的功能是读取最近写入的数据并将其存储到临时缓冲区。此函数可用于调试目的，或者在运行时读取存储的数据。页面模式与 64 位模式的行为有所不同。通常，最近写入的数据（页面或 64 位）存储在 Read_Buffer 中。

页面模式：首先，该函数通过检查 Empty_EEPROM 标志来验证数据是否已写入 EEPROM。如果在写入任何数据之前尝试读取数据，则读入缓冲区的值无效并抛出错误。如果数据已写入，则找到当前的 EEPROM 组和页面，然后填充读缓冲区。

```
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 位模式：首先，该函数通过检查 Empty_EEPROM 标志来验证数据是否已写入 EEPROM。如果在写入任何数据之前尝试读取数据，则读入缓冲区的值无效并抛出错误。如果数据已写入，则指针向后移动四个地址（共 64 位），读取缓冲区被数据填满。

```
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

EEPROM_Erase() 函数的功能是擦除用于仿真的非活动扇区。必须至少擦除一个完整的扇区，因为不支持部分擦除。擦除之前，必须确保存储的数据不再需要/不再有效。在乒乓实现中，仅当使用一个 EEPROM 单元中的所有 EEPROM 组和页面并且数据成功写入另一个 EEPROM 单元时，才会调用该函数。该函数首先重新计算非活动（已满）EEPROM 单元的写入/擦除保护掩码，然后调用 Erase_Bank 函数。

```
// 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();
```

为便于比较，F28003x Ping Pong 示例工程的 EEPROM_Erase 功能因仅调用 Erase_Bank 函数而有所不同。写入/擦除保护掩码在 EEPROM_Erase 调用之外进行配置。

5.2.5.1 Erase_Bank

Erase_Bank 函数利用闪存 API 来擦除非活动（已满）EEPROM 单元。该函数与 EEPROM_Erase 分开，以最大限度地减少在 EEPROM_Config 函数中擦除两个单元所需的 CPU 周期。该函数首先为适当的闪存扇区配置写入/擦除保护掩码，然后调用 Fapi_issueBankEraseCommand。最后，该函数等待完成并检查是否存在任何错误。

```
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);
```

为进行比较，F28003x Ping Pong 示例工程的 EEPROM_BANK 函数问题

闪存 API 发出擦除命令，然后等待命令完成执行，并检查是否发生任何编程错误。在函数范围之外提供了写入/擦除保护掩码。

5.2.6 EEPROM_GetValidBank

EEPROM_GetValidBank() 函数的功能是查找当前 EEPROM 组和页面。EEPROM_Write() 和 EEPROM_Read() 函数都会调用此函数。GetValidBank 流程展示了搜索当前 EEPROM 组和页面所需的总体流程。

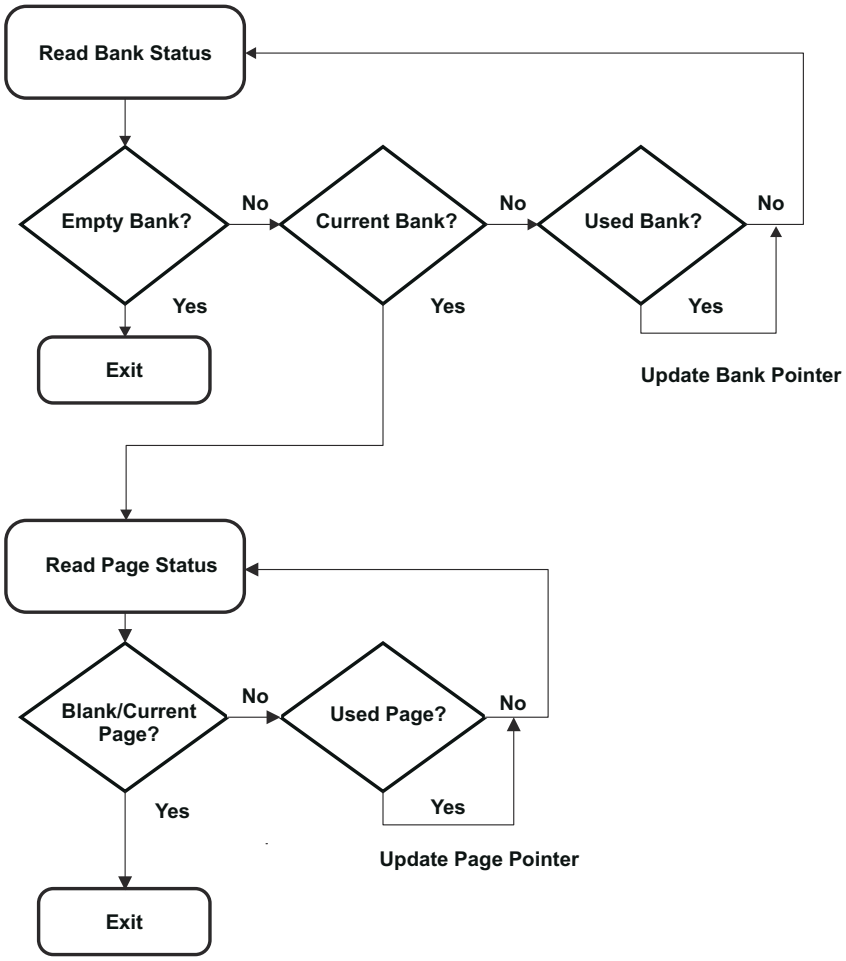

图 5-1. GetValidBank 流程

进入此函数时，EEPROM 组指针和页面指针被设置为 FIRST_AND_LAST_SECTOR 中指定的第一个扇区的开头：

```
RESET_BANK_POINTER;
RESET_PAGE_POINTER;
```

这些指针的地址在 EEPROM_Config.h 文件中针对所使用的特定器件和 EEPROM 配置进行定义。

接下来，会找到当前 EEPROM 组。如 GetValidBank 流程所示，EEPROM 组可以具有三种不同的状态：空、当前和已使用。

空 EEPROM 组由 128 个状态位全部为 1 (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF) 表示。当前 EEPROM 组由最高有效 64 位被设置为 0x5A5A5A5A5A5A5A5A、其余 64 位被设置为 1 (0x5A5A5A5A5A5A5A5AFFFFFFFFFFFFFFFF) 表示。已使用的 EERPOM 组由全部 128 位被设置为 0x5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A 表示。可以根据需要更改这些值。

首先测试空 EEPROM 组。如果遇到此状态，则表示该 EEPROM 组尚未使用，无需进一步搜索。

```
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
}
```

如果未遇到空 EEPROM 组，则接下来测试当前 EEPROM 组。如果 EEPROM 组是当前 EEPROM 组，则会更新 EEPROM 组计数器，并且页面指针被设置为 EEPROM 组的第一页，以启用对当前页面的测试。然后退出该循环，因为不需要进一步的 EEPROM 组搜索。

```
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
}
```

最后，对已使用的 EEPROM 组进行测试。在这种情况下，EEPROM 组已使用，EEPROM 组指针更新到下一个 EEPROM 组，以测试其状态。

```
// 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;
```

找到当前 EEPROM 组后，需要找到当前页面。一个页面可以具有三种不同的状态：空、当前和已使用。

空页面由 128 个状态位全为 1 (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF) 表示。当前页面由最高有效 64 位被设置为 0x5F5F5F5F5F5F5F5F、其余 64 位被设置为 1 (0x5F5F5F5F5F5F5F5FFFFFFFFFFFFFFFFF) 表示。已使用的页面由全部 128 位被设置为 0x5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F 表示。可以根据需要更改这些值。

首先会测试组和当前页面。如果页面的当前状态为这两种状态之一，则表示找到了正确的页面，并退出该循环，因为不需要进一步的搜索。

```
// 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
}
```

如果页面状态不是这两种状态中的任何一种，则唯一的其他可能性是“已使用的页面”。在这种情况下，页面指针会更新到下一个页面，以测试其状态。

```
// 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;
}
```

此时，当前 EEPROM 组和页面已找到，调用函数可以继续运行。最后，此函数将会检查是否所有 EEPROM 组和页面均已使用。这种情况下，需要擦除该扇区。活动单元将被切换，写入/擦除保护掩码将被重新配置，并且 Erase_Inactive_Unit 标志将被设置。

```
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;
    }
}
```

可以通过测试 EEPROM 组和页面计数器来执行该检查。表示已满 EEPROM 的 EEPROM 组和页面的数量取决于应用。如上述代码片段中所示针对当前 EEPROM 组和页面执行测试时，会设置这些计数器。然而，当设置了 Read_Flag 时，不会进行此检查。这是为了防止在从已满 EEPROM 单元读取时过早擦除非活动 EEPROM 单元。

5.2.7 EEPROM_UpdateBankStatus

EEPROM_UpdateBankStatus() 函数的功能是更新 EEPROM 组状态。此函数由 EEPROM_Write() 函数调用。首先读取 EEPROM 组状态，以确定如何继续。

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

如果此状态表示 EEPROM 组为空，则状态会更改为当前并进行编程。

```
// 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;
```

如果状态不为空，则接下来检查是否存在其中的所有页面都被使用的当前 EEPROM 组（已满 EEPROM 组）。在这种情况下，当前 EEPROM 组状态将更新以显示 EEPROM 组已满，并且下一个 EEPROM 组状态将更新为当前以允许对下一个 EEPROM 组进行编程。最后，页面指针会更新为新 EEPROM 组的第一页。

```
// 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

EEPROM_UpdatePageStatus() 函数的功能是更新上一页的状态。此函数由 EEPROM_Write() 函数调用。首先读取页状态，以确定如何继续。

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

如果此状态表示该页面为空，该函数便会退出，因为此状态会在 EEPROM_Write() 函数中更新。否则，页面状态会更新，以显示页面已满，同时页面指针会递增，为对下一个页面进行编程做好准备：

```
// 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

EEPROM_UpdatePageData() 函数的功能是更新 EEPROM 页数据。此函数由 EEPROM_Write() 函数调用。

为了实现这一目标，需要采取以下步骤：

1. 清除闪存状态机 (FSM) 状态。
2. 配置闪存扇区的编程/擦除保护。
   - EEPROM 仿真中未使用的扇区将启用保护。
   - EEPROM 仿真中已使用的扇区将禁用保护。
3. 计算相对于页面指针的偏移量以写入数据。
   a. 这是必需的，因为一次仅写入 64 位。因此，如果数据大小大于 64 位，则需要多次调用闪存 API 才能写入整个页面。
4. 等待编程完成并检查是否存在任何编程错误。

```
// 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);
```

以下参数被传递至闪存 API 以进行编程。

- 页面指针 (编程地址)
- 包含待写入数据的缓冲区
- 要编程的数据长度
- 编程模式

使用 Fapi_AutoEccGeneration 模式时，第四个和第五个参数为零。有关更多详细信息，请参阅 *TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南*。

如果编程成功，则会更新当前页面的页面状态并清除 Empty_EEPROM 标志。代码如下所示：

```
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; }
```

成功写入后，该函数会检查是否需要擦除非活动 EEPROM 单元。如果需要，则该函数调用 EEPROM_Erase，清除标志并重新配置 W/E 保护掩码。在调用擦除之前发出用于设置空白检查的标志。擦除非活动单元的标志在 EEPROM_GetValid_Bank 中设置。

```
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

EEPROM_Get_64_Bit_Data_Address() 提供确定 EEPROM 单元是否已满并分配正确地址 (如果需要) 的功能。如果检测到已满的 EEPROM 单元，则使用 EEPROM_Erase() 函数擦除 EEPROM，并切换活动 EEPROM 单元。

首先，根据所使用的器件和配置设置 EEPROM 的结束地址。END_OF_SECTOR 指令在 EEPROM_Config.h 文件中进行设置。

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

接下来，将 EEPROM 组指针与结束地址进行比较。如果从当前 EEPROM 组指针开始写入 4 个 16 位字会超出结束地址，则表明该扇区已满。此时，将切换活动 EEPROM 单元，配置新的写入/保护掩码，设置 Erase_Inactive_Unit 标志，并将 EEPROM 组指针重置为新的活动 EEPROM 单元的开头。

```
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

EEPROM_Program_64_Bits() 函数提供将四个 16 位字编程到存储器中的功能。第一个参数 Num_Words 允许用户指定将写入多少个有效字。数据字应分配给 Write_Buffer 的前 4 个位置，以供 Fapi_issueProgrammingCommand 函数使用。如果函数调用中指定的字少于四个，则缺少的字将用 0xFFFF 填充。这样做是为了符合 ECC 要求。

首先，测试是否存在已满的 EEPROM 单元。

```
EEPROM_Get_64_Bit_Data_Address();
```

接下来，如果指定的字数少于 4，则写入缓冲区中将填充 1。

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

接下来，对数据进行编程，并且指针递增到对数据进行编程的下一个位置。

```
// 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;
```

编程完成后，将检查 Erase_Inactive_Unit 标志。如果设置，则非活动单元将被擦除，执行空白检查，并且写入/擦除保护掩码将被重新配置。

```
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;
}
```

备注

在执行 RESET_BANK_POINTER 设置指针之前，无法使用该函数。在本例中，该函数在 EEPROM_Config_Check() 中调用。如果在此之前执行，则会生成未知结果。

5.2.12 EEPROM_CheckStatus

EEPROM_CheckStatus 函数提供检查闪存 API 状态以及在每次对闪存进行编程/擦除后检查闪存状态机状态的功能。在执行擦除操作之后，还需执行额外的检查以确认闪存为空白。如果检测到任何意外状态，程序会停止。该工程中未实现错误处理。

```
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

ClearFSMStatus() 函数负责清除之前闪存操作的状态。该函数适用于 F280013x、F280015x、F28P65x 和 F28P55x 器件。该函数必须按原样使用。

```
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 测试示例

提供的示例使用 F28P650DK9 进行了测试。为了正常测试示例，需要在 Code Composer Studio 中使用存储器窗口和断点。在对工程进行编程和测试时，执行了以下步骤。

1. 通过 USB 和带有 JTAG 接头的 XDS110 调试探针将 F28P650DK9 连接到 PC。
2. 将一个 5V 直流电源连接到电路板。
3. 启动 Code Composer Studio 并打开 F28P65x_EEPROM_PingPong_Example.pjt。
4. 通过选择“Project”-> “Build Project”构建工程。
5. 通过依次转到“View”-> “Target Configurations”-> “F28P65x_EEPROM_PingPong_Example”-> “targetConfigs”-> 右键点击 TMS320F28P650DK9.ccxml -> “Launch Selected Configuration”来启动目标配置。
6. 通过转到“Debug”窗口，右键点击“Texas Instruments XDS110 USB Debug Probe_0/C28xx_CPU1”，然后选择“Connect Target”来连接到 CPU1。
7. 通过点击“Load Symbols”并从工程中选择 F28P65x_EEPROM_PingPong.out 来加载符号。
8. 设置断点，以正确地查看写入存储器窗口中的内存的数据和从存储器读取的数据，如断点中所示。

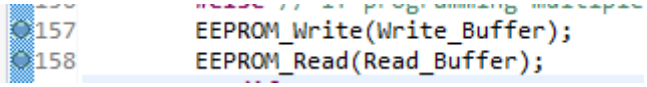

图 5-2. 断点

9. 运行至第一个断点，然后打开 Memory Browser（“View”->“Memory Browser”）来查看数据。Bank_Pointer 可用来观察写入的数据，而 Read_Buffer 可用来观察从存储器读回的数据。对 EEPROM 单元进行写入和读取数据展示了该情况。

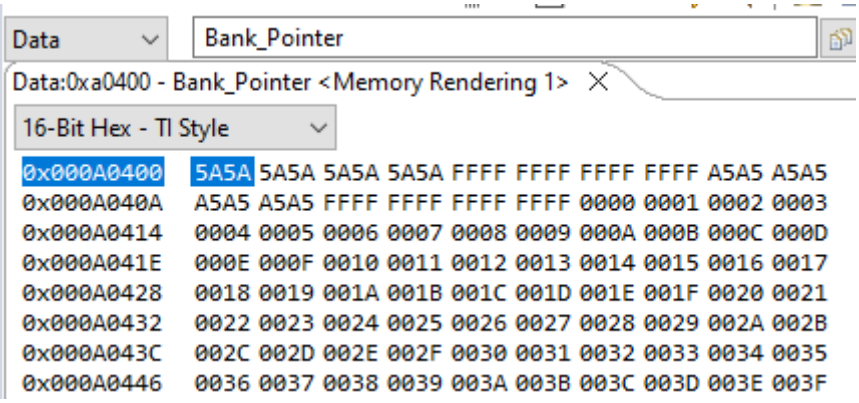

图 5-3. 对 EEPROM 单元进行写入

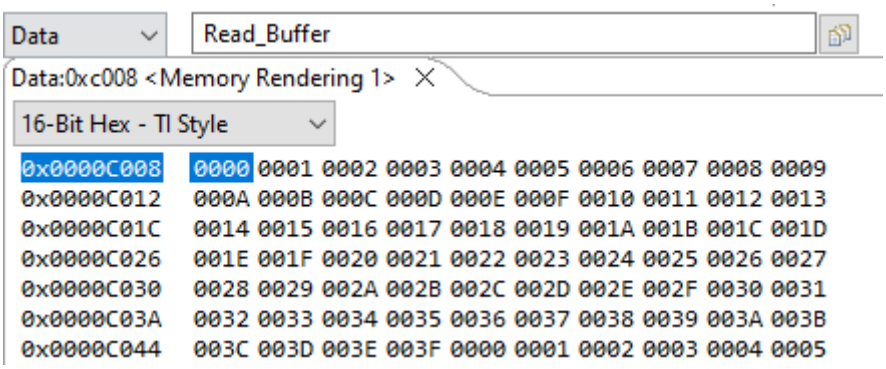

图 5-4. 读数据

10. 继续从断点运行到断点，直到程序运行完成或 EEPROM 已满。
11. EEPROM 已满后，您将看到新数据写入先前不活动的单元，并且已满 EEEPROM 将被擦除。对新 EEPROM 单元进行写入和擦除已满 EEPROM 单元展示了该情况。

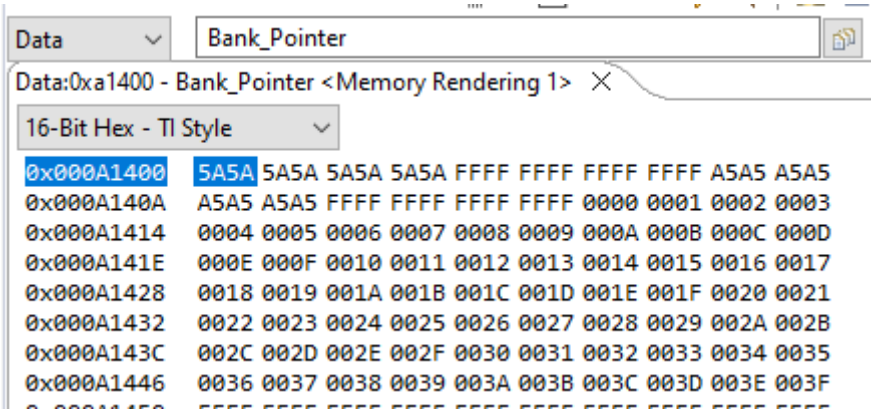

图 5-5. 对新 EEPROM 单元进行写入

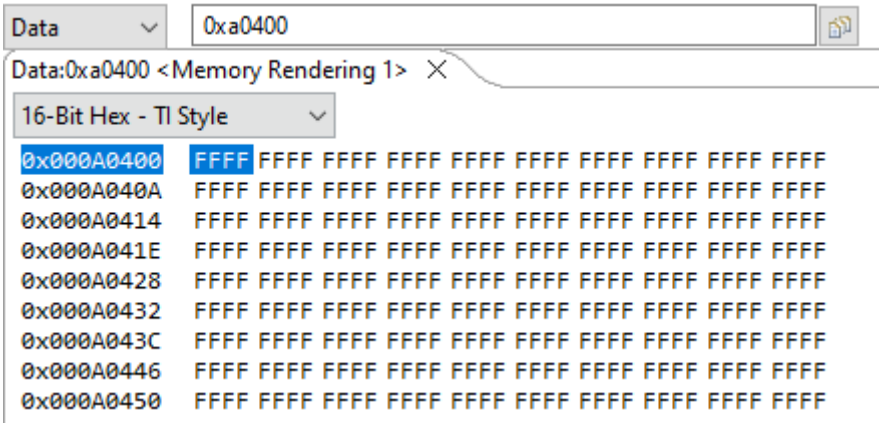

图 5-6. 擦除已满 EEPROM 单元

12. 可以根据需要在两个 EEPROM 单元之间重复该过程。

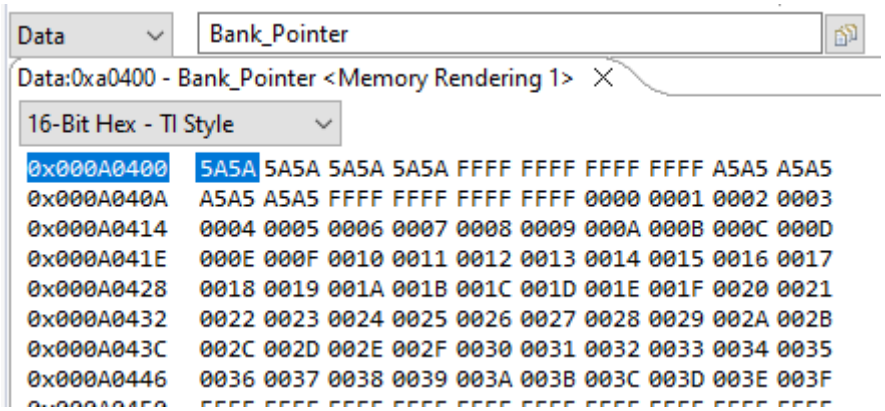

图 5-7. 对原始 EEPROM 单元进行写入

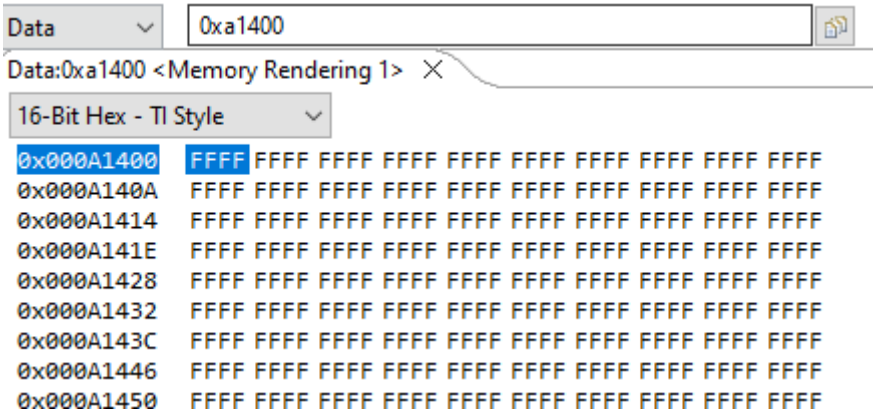

图 5-8. 擦除已满 EEPROM 单元

上述步骤用于测试页面模式配置。64 位模式配置也可以使用相同的步骤进行测试。要启用 64 位模式，请通过取消注释 _64_BIT_MODE 指令并注释掉 PAGE_MODE 指令来更改 EEPROM_PingPong_Config.h 文件中的定义。

6 单存储单元仿真

单存储单元仿真与乒乓仿真类似，但前者仅使用一组闪存扇区。因此，当检测到已满 EEPROM 时，无法使用乒乓方法。本节介绍已实现的行为。许多功能在两种模式之间保持相同，主要差异体现在擦除行为上。

6.1 用户配置

本文档中详细介绍的实现允许您为 EEPROM 仿真配置多个变量。这些变量主要位于 EEPROM_Config.h 中，但有一个变量包含在 F28P65x_EEPROM.c 中。

6.1.1 EEPROM_Config.h

该头文件包含允许用户更改 EEPROM 配置各个方面的定义。这些方面包括：

- 定义正在使用的器件型号。这允许在并非所有器件都通用的闪存组中进行 EEPROM 仿真

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

- 在页面模式和 64 位模式之间选择

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

- 选择要用于仿真的闪存组。**默认情况下，闪存 API 和程序从闪存组 0 中存储/运行，因此该闪存组无法用于 EEPROM 仿真。通常，闪存 API 和程序应该从与用于 EEPROM 仿真的闪存组不同的闪存组中存储/运行。**

```
#define FLASH_BANK_SELECT FlashBank1StartAddress
```

- 定义闪存扇区大小（单位为 16 位字）。这会因使用的器件而异，请参阅相应的数据表了解详细信息。

```
#define FLASH_SECTOR_SIZE F28P65x_FLASH_SECTOR_SIZE
```

- 定义一个闪存组中有多少个闪存扇区。这会因使用的器件而异，请参阅相应的数据表了解详细信息。

```
#define NUM_FLASH_SECTORS F28P65x_NUM_FLASH_SECTORS
```

- 选择要仿真的 EEPROM 组的数量。

```
#define NUM_EEPROM_BANKS 4
```

- 选择每个 EEPROM 组中有多少个 EEPROM 页面

```
#define NUM_EEPROM_PAGES 3
```

- 选择每个 EEPROM 页面中包含的数据大小（单位为 16 位字）。尽管可以指定任何大小，但该大小将调整为大于或等于指定大小的最接近的四的倍数。例如，每页 6 个 16 位字的指定大小将被编程为每页 8 个 16 位字，最后两个字被视为 0xFFFF。这是为了符合闪存要求（为每个 64 位对齐的闪存存储器地址进行 8 位 ECC 编程）。

```
#define DATA_SIZE 64
```

6.1.2 F28P65x_EEPROM.c

选择用于 EEPROM 仿真的闪存扇区。选择的扇区（如果有多个）应该是连续的并且按从小到大的顺序排列。仅插入要用于 EEPROM 的第一个和最后一个扇区。例如，要使用扇区 1-10，请插入 {1,10}。要仅使用扇区 1，请插入 {1,1}。有效的配置将具有以下属性。

- 意味着与 EEPROM_Config.h 中指定的用于仿真的扇区数量相同
- 仅包含器件上存在的扇区
- 不会在两个单元之间的写入/擦除保护掩码中产生重叠
  - F28P65x 闪存 API 要求在对闪存存储器进行编程之前配置写入/擦除保护掩码。有关这些掩码正确配置的详细信息，请参阅 F28P65x 闪存 API 参考指南。

有关无效或危险配置的更多详细信息，请参阅节 6.2.1。

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

6.2 EEPROM 函数

为了实现该功能，需要使用 11 个函数在页面编程中执行配置、编程、读取和擦除。64 位编程需要两个附加函数。所有函数均包含在 F28P65x_EEPROM.c 或 F28P65x_EEPROM.c 文件中。

- 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()

后续小节中会详细介绍上述每个函数。

6.2.1 EEPROM_Config_Check

EEPROM_Config_Check() 函数提供一般错误检查并配置闪存 API 所需的写入/擦除保护掩码。应在对仿真 EEPROM 单元进行编程或读取之前调用此函数。

第一，该函数验证选择用于 EEPROM 仿真的闪存组是否有效。有效的闪存组选择不得选择组 0 进行仿真，并且必须受特定器件型号的支持。例如，只有特定的 F28p65x 型号具有闪存组 2-4。要验证该信息，请参阅特定于器件的数据表。

```
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
}
```

第二，检查选择用于仿真的闪存扇区的有效性。该函数检查：

- 选择用于仿真的闪存扇区数量是否多于闪存组中可用的扇区数量

```
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;
}
```

- 选择用于仿真的第一个和最后一个扇区的组合是否无效

```
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;
}
```

如果使用页面模式，还将检查以下各项

- 检查 EEPROM 组 + 页面的总大小是否适合所选的闪存扇区

```
// 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;
}
```

如果检测到以下情况之一，该函数还会通过相应的代码发出警告：

- 配置 EEPROM 组和页面大小后，闪存中会保留一个或多个 EEPROM 组的空间

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

- 如果每个页面包含少于 5 个 16 位字（这会浪费空间，因为无需状态代码即可使用 64 位模式）

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

- 如果使用 32-127 范围内的扇区（对于 F28P65x 器件）并且未使用分配给写入保护掩码中 single-bit 的全部八个扇区，则会发出警告。由 single-bit 设计的八个扇区中任何未使用的扇区都无法受到适当的擦除保护。有关写保护掩码的更多详细信息，请参阅 *TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南*。

```
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;
        }
    }
}
```

最后，为活动 EEPROM 单元配置写入/擦除保护掩码。该函数还通过擦除要用于编程的扇区来为仿真准备好闪存。

```
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

Configure_Protection_Masks 提供了为选择用于 EEPROM 仿真的任何扇区禁用写入/擦除保护的功能。这是通过计算要传递到 Fapi_setupBankSectorEnable 函数的适当掩码来完成的。该函数需要两个参数，即指向所选闪存扇区编号的指针和要仿真的闪存扇区的数量。有关 Fapi_setupBankSectorEnable 函数实现的更多详细信息，请参阅 *TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南*。

该函数的返回值用于禁用为 EEPROM 仿真选择的闪存扇区中的写入/擦除保护。

```
// 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;
```

为了进行比较，F28003x EEPROM 示例工程 Configure_Protection_Masks 的功能与 F28P65x EEPROM 示例工程的功能不同，后者有一定量的扇区可用于保护。写入/擦除保护掩码中的每个位表示它自己的扇区。

```
// 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

EEPROM_Write() 函数的功能是将数据编程写入闪存。该函数直接利用闪存 API 并在其中进行多个函数调用以准备数据编程。下面列出了调用的函数：

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

相应的各节会详细介绍上述每个函数。首先，找到当前的 EEPROM 组和页面。找到当前 EEPROM 组和页面后，便会更新上一个页面的页面状态，如果要使用新 EEPROM 组，则会更新 EEPROM 组状态。接下来，在 EEPROM 页数据更新期间进行实际编程。

```
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

EEPROM_Read() 函数的功能是读取最近写入的数据并将该数据存储到临时缓冲区。此函数可用于调试目的，或者在运行时读取存储的数据。页面模式与 64 位模式的行为有所不同。通常，最近写入的数据（页面或 64 位）存储在 Read_Buffer 中。

页面模式：首先，该函数通过检查 Empty_EEPROM 标志来验证数据是否已写入 EEPROM。如果在写入任何数据之前尝试读取数据，则读入缓冲区的值无效并抛出错误。如果数据已写入，则找到当前的 EEPROM 组和页面，然后填充读缓冲区。

```
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 位模式：首先，该函数通过检查 Empty_EEPROM 标志来验证数据是否已写入 EEPROM。如果在写入任何数据之前尝试读取数据，则读入缓冲区的值无效并抛出错误。如果数据已写入，则指针向后移动四个地址（共 64 位），读取缓冲区被数据填满。

```
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

EEPROM_Erase() 函数的功能是擦除用于仿真的扇区。必须至少擦除一个完整的扇区，因为不支持部分擦除。擦除之前，必须确保存储的数据不再需要/不再有效。在单存储单元实现中，仅当所有 EEPROM 组和页面已满时才调用此函数。

该函数首先配置 EEPROM 单元的写入/擦除保护掩码，然后调用 Fapi_issueBankEraseCommand 函数。最后，该函数等待完成并检查是否存在任何错误。

```
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);
```

在单存储单元实现中，EEPROM_Erase 函数利用闪存 API 来清除闪存组。该函数不再需要乒乓实现中的 Erase_Bank 函数，这两者已合并在 EEPROM_Erase 中。不再需要 Erase_Bank，因为创建该函数是为了优化在有两个 EEPROM 单元时指定用于 EEPROM 仿真的所有闪存扇区的擦除。

为进行比较，F28003x 示例工程目的 EEPROM_Erase 函数问题

闪存 API 发出擦除命令，然后等待命令完成执行，并检查是否发生任何编程错误。在函数范围之外提供了写入/擦除保护掩码。

```
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

EEPROM_GetValidBank() 函数的功能是查找当前 EEPROM 组和页面。EEPROM_Write() 和 EEPROM_Read() 函数都会调用此函数。GetValidBank 流程展示了搜索当前 EEPROM 组和页面所需的总体流程。

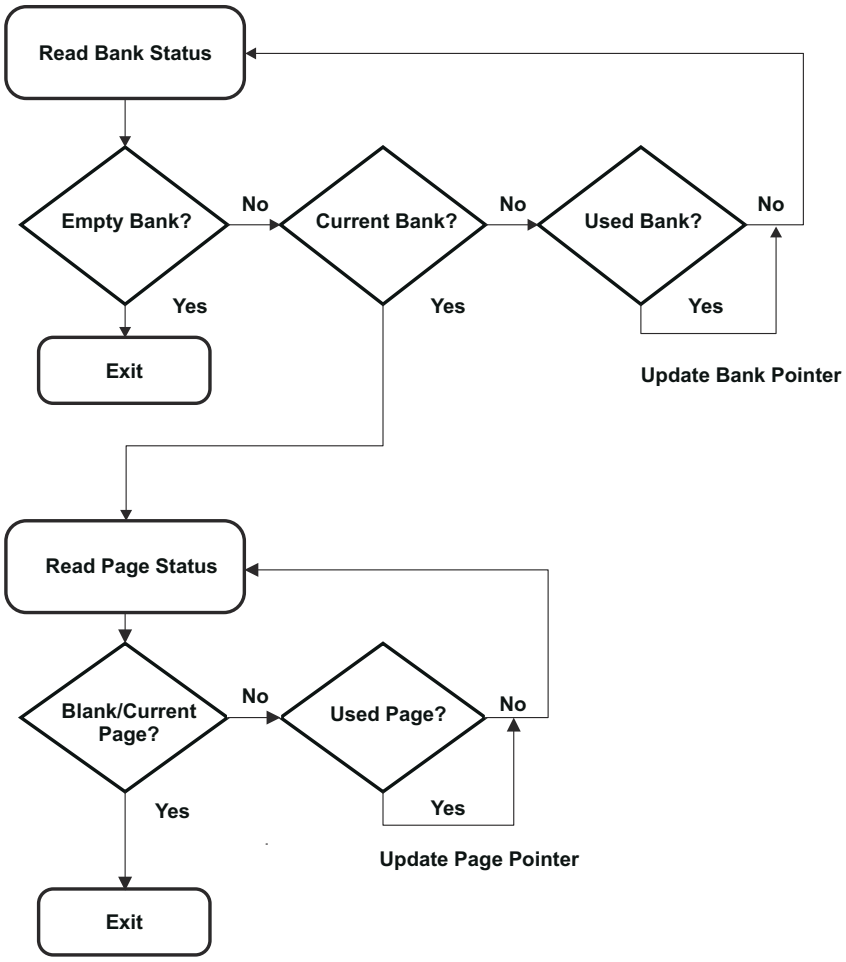

图 6-1. GetValidBank 流程

进入此函数时，EEPROM 组指针和页面指针被设置为 FIRST_AND_LAST_SECTOR 中指定的第一个扇区的开头：

```
RESET_BANK_POINTER;
RESET_PAGE_POINTER;
```

这些指针的地址在 EEPROM_Config.h 文件中针对所使用的特定器件和 EEPROM 配置进行定义。

接下来，会找到当前 EEPROM 组。如 GetValidBank 流程所示，EEPROM 组可以具有三种不同的状态：空、当前和已使用。

空 EEPROM 组由 128 个状态位全部为 1 (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF) 表示。当前 EEPROM 组由前 64 位被设置为 0x5A5A5A5A5A5A5A5A、其余 64 位被设置为 1 表示。已使用的 EEPROM 组由全部 128 位被设置为 0x5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A 表示。可以根据需要更改这些值。

首先测试空 EEPROM 组。如果遇到此状态，则表示该 EEPROM 组尚未使用，无需进一步搜索。

```
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
}
```

如果未遇到空 EEPROM 组，则接下来测试当前 EEPROM 组。如果组是当前 EEPROM 组，则会更新 EEPROM 组计数器，并且页面指针被设置为 EEPROM 组的第一页，以启用对当前页面的测试。然后退出该循环，因为不需要进一步的 EEPROM 组搜索。

```
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
}
```

最后，对已使用的 EEPROM 组进行测试。在这种情况下，EEPROM 组已使用，EEPROM 组指针更新到下一个 EEPROM 组，以测试其状态。

```
// 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;
```

找到当前 EEPROM 组后，需要找到当前页面。一个页面可以具有三种不同的状态：空、当前和已使用。

空页面由 128 个状态位全为 1 (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF) 表示。当前 EEPROM 组由前 64 位被设置为 0x5F5F5F5F5F5F5F5F、其余 64 位被设置为 1 表示。已使用的 EEPROM 组由全部 128 位被设置为 0x5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F 表示。可以根据需要更改这些值。

首先会测试组和当前页面。如果页面的当前状态为这两种状态之一，则表示找到了正确的页面，并退出该循环，因为不需要进一步的搜索。

```
// 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
}
```

如果页面状态不是这两种状态中的任何一种，则唯一的其他可能性是“已使用的页面”。在这种情况下，页面指针会更新到下一个页面，以测试其状态。

```
// 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;
}
```

此时，当前 EEPROM 组和页面已找到，调用函数可以继续运行。最后，此函数将会检查是否所有 EEPROM 组和页面均已使用。这种情况下，需要擦除该扇区。

```
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;
    }
}
```

可以通过测试 EEPROM 组和页面计数器来执行该检查。表示已满 EEPROM 的 EEPROM 组和页面的数量取决于应用。如上述代码片段中所示针对当前 EEPROM 组和页面执行测试时，会设置这些计数器。然而，当设置了 Read_Flag 时，不会进行此检查。这是为了防止在从已满 EEPROM 单元读取时过早擦除非活动 EEPROM 单元。

如上方所示，如果存储器已满，则会调用 EEPROM_Erase() 函数，同时 EEPROM 组和页面指针会复位至第一个 EEPROM 组和第一个页面。

6.2.7 EEPROM_Get_64_Bit_Data_Address

EEPROM_Get_64_Bit_Data_Address 在单存储单元实现中基本没有变化，但检测到已满 EEPROM 单元时的行为有所不同。

```
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;
}
```

如上所示，如果 EEPROM 单元已满，则只需将其擦除，检查它是否为空白，然后将指针重置为单元的开头。

6.2.8 EEPROM_UpdateBankStatus

EEPROM_UpdateBankStatus() 函数的功能是更新 EEPROM 组状态。此函数由 EEPROM_Write() 函数调用。首先读取 EEPROM 组状态，以确定如何继续。

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

如果此状态表示 EEPROM 组为空，则状态会更改为当前并进行编程。

```
// 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;
```

如果状态不为空，则接下来检查是否存在其中的所有页面都被使用的当前 EEPROM 组（已满 EEPROM 组）。在这种情况下，当前 EEPROM 组状态将更新以显示 EEPROM 组已满，并且下一个 EEPROM 组状态将更新为当前以允许对下一个 EEPROM 组进行编程。最后，页面指针会更新为新 EEPROM 组的第一页。

```
// 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

EEPROM_UpdatePageStatus() 函数的功能是更新上一页的状态。此函数由 EEPROM_Write() 函数调用。首先读取页状态，以确定如何继续。

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

如果此状态表示该页面为空，该函数便会退出，因为此状态会在 EEPROM_Write() 函数中更新。否则，页面状态会更新，以显示页面已满，同时页面指针会递增，为对下一个页面进行编程做好准备：

```
// 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

EEPROM_UpdatePageData() 函数的功能是更新 EEPROM 页数据。此函数由 EEPROM_Write() 函数调用。

为了实现这一目标，需要采取以下步骤：

1. 清除闪存状态机 (FSM) 状态。
2. 配置闪存扇区的编程/擦除保护。
   a. EEPROM 仿真中未使用的扇区将启用保护。
   b. EEPROM 仿真中已使用的扇区将禁用保护。
3. 计算相对于页面指针的偏移量以写入数据。
   a. 这是必需的，因为一次仅写入 64 位。因此，如果数据大小大于 64 位，则需要多次调用闪存 API 才能写入整个页面。
4. 等待编程完成并检查是否存在任何编程错误。

```
// 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);
```

以下参数被传递至闪存 API 以进行编程。

- 页面指针 (编程地址)
- 包含待写入数据的缓冲区
- 要编程的数据长度
- 编程模式

使用 Fapi_AutoEccGeneration 模式时，第四个和第五个参数为零。有关更多详细信息，请参阅 *TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南*。

如果编程成功，则会更新当前页面的页面状态并清除 Empty_EEPROM 标志。代码如下所示：

```
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

EEPROM_Get_64_Bit_Data_Address() 提供确定 EEPROM 单元是否已满并分配正确地址 (如果需要) 的功能。如果检测到已满的 EEPROM 单元，则使用 EEPROM_Erase() 函数擦除 EEPROM，并将地址重置为第一个闪存扇区的开头。

首先，根据所使用的器件和配置设置 EEPROM 的结束地址。END_OF_SECTOR 指令在 EEPROM_Config.h 文件中进行设置。

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

接下来，将 EEPROM 组指针与结束地址进行比较。如果从当前 EEPROM 组指针开始写入 4 个 16 位字会超出结束地址，则表明该扇区已满。此时，EEPROM 单元被擦除，执行空白检查，并且 EEPROM 组指针被重置为 EEPROM 单元的开头。

```
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

EEPROM_Program_64_Bits() 函数提供将四个 16 位字编程到存储器中的功能。第一个参数 Num_Words 允许用户指定将写入多少个有效字。数据字分配给 Write_Buffer 的前 4 个索引，以供 Fapi_issueProgrammingCommand 函数使用。如果函数调用中指定的字少于四个，则缺少的字将用 0xFFFF 填充。这样做是为了符合 ECC 要求。

首先，测试是否存在已满的 EEPROM 单元。

```
EEPROM_Get_64_Bit_Data_Address();
```

接下来，如果指定的字数少于 4，则写入缓冲区中将填充 1。

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

接下来，对数据进行编程，并且指针递增到对数据进行编程的下一个位置。

```
// 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;
```

备注

在执行 RESET_BANK_POINTER 设置指针之前，无法使用该函数。在本例中，该函数在 EEPROM_Config_Check() 中调用。如果在此之前执行，则可能会生成未知结果。

6.2.13 EEPROM_CheckStatus

EEPROM_CheckStatus 函数提供检查闪存 API 状态以及在每次对闪存进行编程/擦除后检查闪存状态机状态的功能。如果检测到任何意外状态，程序会停止。该工程中未实现错误处理。

```
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

ClearFSMStatus() 函数负责清除之前闪存操作的状态。该函数适用于 F280013x、F280015x、F28P65x 和 F28P55x 器件。该函数必须按原样使用。

```
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 测试示例

提供的示例使用 F28P650DK9 进行了测试。为了正常测试示例，需要在 Code Composer Studio 中使用存储器窗口和断点。在对工程进行编程和测试时，执行了以下步骤。

1. 通过 USB 和带有 JTAG 接头的 XDS110 调试探针将 F28P650DK9 连接到 PC。
2. 将一个 5V 直流电源连接到电路板。
3. 启动 Code Composer Studio 并打开 F28P65x_EEPROM_Example.pjt。
4. 通过选择“Project”-> “Build Project”构建工程。
5. 通过依次转到“View”-> “Target Configurations”-> “F28P65x_EEPROM_Example”-> “targetConfigs”-> 右键点击 TMS320F28P650DK9.ccxml -> “Launch Selected Configuration”来启动目标配置。
6. 通过转到“Debug”窗口，右键点击“Texas Instruments XDS110 USB Debug Probe_0/C28xx_CPU1”，然后选择“Connect Target”来连接到 CPU1。
7. 通过点击“Load Symbols”并从工程中选择 F28P65x_EEPROM.out 来加载符号。
8. 设置断点，以正确地查看写入存储器窗口中的内存的数据和从存储器读取的数据，如断点中所示。

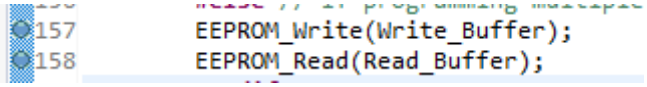

图 6-2. 断点

9. 运行至第一个断点，然后打开 Memory Browser（“View”->“Memory Browser”）来查看数据。Bank_Pointer 可用来观察写入的数据，而 Read_Buffer 可用来观察从存储器读回的数据，对 EEPROM 进行写入和读取数据展示了该情况。

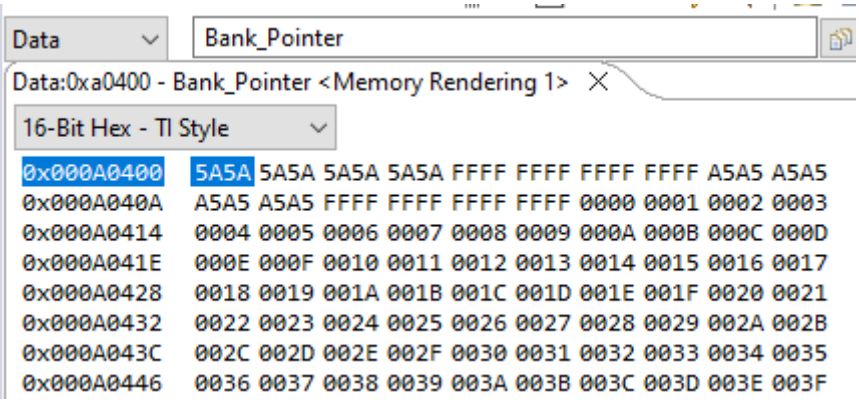

图 6-3. 对 EEPROM 进行写入

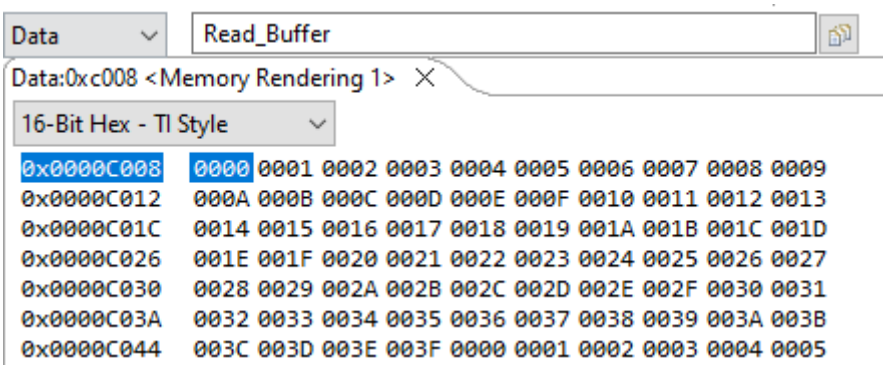

图 6-4. 读数据

10. 继续从断点运行到断点，直到程序运行完成或 EEPROM 已满。
11. EEPROM 已满后，您将看到新数据写入先前不活动的单元，并且已满 EEEPROM 将被擦除。擦除已满 EEPROM 单元和对 EEPROM 进行写入展示了该情况。

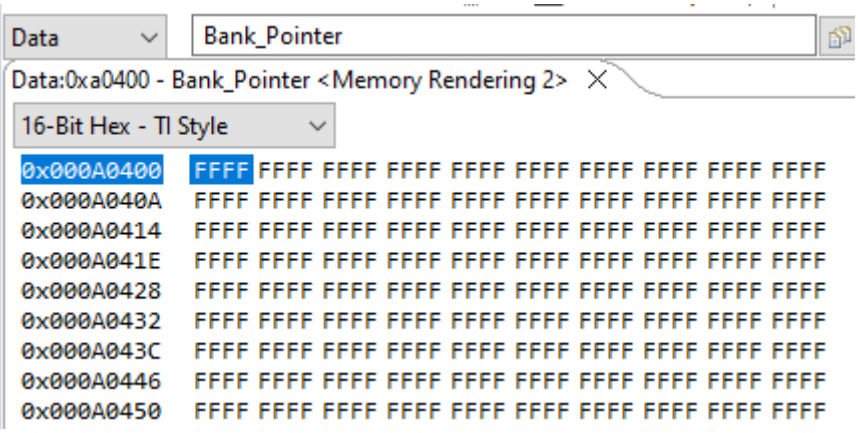

图 6-5. 擦除已满 EEPROM 单元

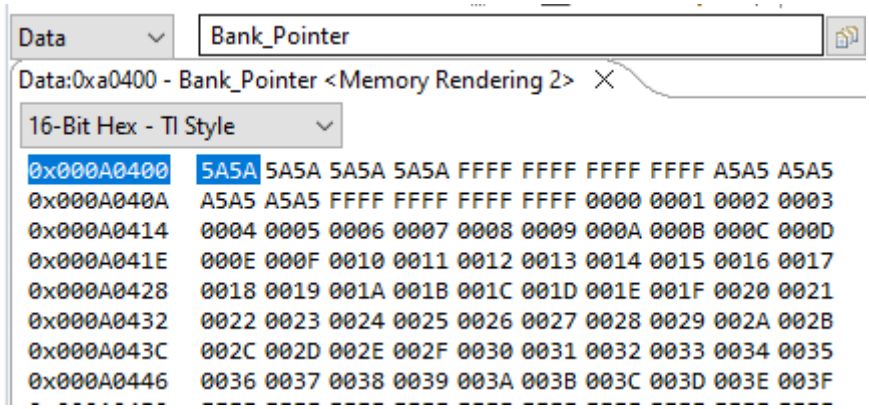

图 6-6. 对 EEPROM 进行写入

12. 可以根据需要重复该过程。

上述步骤用于测试页面模式配置。64 位模式配置也可以使用相同的步骤进行测试。要启用 64 位模式，请通过取消注释 _64_BIT_MODE 指令并注释掉 PAGE_MODE 指令来更改 EEPROM_Config.h 文件中的定义。

7 应用集成

如果应用需要此功能，则需要包含针对该器件提供的 EEPROM_Config.h 和 EEPROM.c 文件。另外，还需包含面向相应器件的闪存 API 和 driverlib。例如，对于 F28P650DK9 上的单存储单元仿真，需要以下文件：

- F28P65x_EEPROM.c
- EEPROM_Config.h
- Device.c 和 device.h
- flash_programming_f28p65x.h
- FAPI_F28P65x_EABI_v3.02.00.lib
- driverlib.lib

备注

随着硅元素新版本的不断发布，闪存 API 也将定期行更新。为了确保功能正常，应使用最新的闪存 API 库。

8 适配其他第 3 代 C2000 MCU

如本文档前面所述，本指南使用 F28P65x 来演示 EEPROM 仿真功能。不过，该工程可以通过对宏和函数定义进行少量更改来适应其他第 3 代 C2000 MCU。为了说明这一点，本节讨论使用 F280013x 所需的更改。

首先，应该指出的是，F280013x 只有一个闪存组，而某些 F28P65x 器件中有五个闪存组。因此，应使用 CPU1_RAM 构建配置而不是 CPU1_FLASH 构建配置。这是必要的，因为闪存 API 无法在包含它的同一闪存组上执行。

此外，EEPROM_Config.h 文件中包含的默认配置使用特定于器件的值来创建定义和宏。这些值应更新为 *TMS320F280013x 实时微控制器数据表* 中的值。对于 F280013x，这些值恰好与 F28P65x 的默认配置相同，但应始终使用特定于器件的数据表来验证这些值。

```
#define FLASH_BANK_SELECT 0x80000

#define FLASH_SECTOR_SIZE 0x400

#define NUM_FLASH_SECTORS 128
```

这些值对于 EEPROM_Config_Check 中的错误检查以及定义 EEPROM 仿真的起始/结束地址非常重要。

最后，使用 F280013x 时需要修改 EEPROM_Config_Check() 函数。默认情况下，闪存组 0 被保留用于存储闪存 API，如果选择此闪存组进行 EEPROM 仿真，该函数将抛出错误。然而，由于选择了 CPU_1_RAM 构建配置，闪存组 0 现在可用于 EEPROM 仿真。因此，应该在函数中删除或注释掉这些行。

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

使用 F280013x 所需的更改相对简单，而使用其他第 3 代 C2000 MCU 可能需要更多更改。有关可用工程的列表、请参阅故障排除章节。

9 闪存 API

闪存 API 常驻 CPU 中，并由 CPU 调用来完成各种闪存操作。API 库包括用于擦除、编程和校验闪存阵列的函数。一次可以擦除的最小内存量是一个扇区。编程函数只能将位从 1 更改为 0（假设相应的 ECC 位尚未写入）。编程函数不能将位从 0 改回为 1。编程功能一次对单个 16 位字进行操作，但每次都必须写入 64 位，以符合 ECC 要求。

9.1 闪存 API 检查清单

以下部分摘自闪存 API 参考指南，说明了使用各种 API 函数的流程。

- 器件首次上电后，必须先调用 Fapi_initializeAPI() 函数，然后才能调用任何其他 API 函数（Fapi_getLibraryInfo() 函数除外）。此过程根据用户指定的操作系统频率配置闪存包装程序。
- 在首次进行闪存操作之前，必须调用 Fapi_setActiveFlashBank() 函数。
- 如果在初始调用 Fapi_initializeAPI() 函数后更改了系统工作频率，则必须再次调用该函数，然后才能使用任何其他 API 函数（Fapi_getLibraryInfo() 函数除外）。该过程会更新 API 内部状态变量。

9.1.1 使用闪存 API 时的注意事项

使用 API 时的应做事项

- 从 RAM 或未选择用于 EEPROM 仿真的闪存组执行闪存 API 代码（某些功能必须从 RAM 运行）。
- 针对正确的 CPU 工作频率配置 API
- 按照闪存 API 检查清单来将 API 集成到应用中
- 根据需要配置 PLL，并将配置的 CPUCLK 值传递给 Fapi_initializeAPI() 函数。请注意，当系统频率小于或等于 20MHz 时，闪存 API 库不支持闪存擦除/编程操作。
- 根据需要配置 BANKMUXSEL 和 FLASHCTLSEM 寄存器
- 在调用闪存 API 函数之前，请根据特定于器件的数据手册配置等待状态。如果应用程序配置的等待状态不适合应用程序的工作频率，闪存 API 会发出错误。
- 请仔细查看 *TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南* 中所述的 API 限制。

使用 API 时的禁止事项

- 请勿从选择用于仿真的同一个闪存组执行闪存 API
- 请勿配置导致从正在进行擦除/编程操作的闪存组进行读取/获取访问的中断服务例程 (ISR)。闪存 API 函数、调用闪存 API 函数的用户应用程序函数以及任何 ISR 必须从 RAM 或没有正在进行的活动擦除/编程操作的闪存组中执行。
- 请勿访问正在进行闪存擦除/编程操作的闪存组或 OTP
- 不应针对链路指针位置对 ECC 进行编程。当为编程操作提供的起始地址是三个链路指针地址中的任何一个时，API 将跳过对 ECC 的编程。应注意为应用程序中的链路指针位置维护一个单独的结构/段。请勿将此类字段与其他 DCSM OTP 设置混合。如果其他字段与链路指针混合，API 也会跳过对非链路指针位置的 ECC 编程。这会导致应用程序中出现 ECC 错误。

10 源文件清单

文件	功能	说明
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()	验证 EEPROM 配置 配置 W/E 保护掩码位 执行写入操作 执行读取操作 执行擦除操作 执行擦除操作 查找有效组和页面 更新组状态 更新页面状态 更新页面数据 查找 64 位操作的指针并测试是否存在已满扇区 将 64 位编程到闪存中 验证闪存操作是否成功 清除闪存状态机状态
EEPROM_PingPong_Config.h		包含函数原型、全局变量、包括闪存 API 头、指针初始化、常量和宏定义、输入用户可配置变量
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()	验证 EEPROM 配置 配置 W/E 保护掩码位 执行写入操作 执行读取操作 执行擦除操作 查找有效组和页面 更新组状态 更新页面状态 更新页面数据 查找 64 位操作的指针并测试是否存在已满扇区 将 64 位编程到闪存中 验证闪存操作是否成功 清除闪存状态机状态
EEPROM_Config.h		包含函数原型、全局变量、包括闪存 API 头、指针初始化、常量和宏定义、输入用户可配置变量

11 故障排除

以下是针对用户在使用 EEPROM 和 EEPROM_PingPong 工程时遇到的一些常见问题的解决方案。

11.1 常见问题

问题：我找不到 EEPROM 和 EEPROM_PingPong 工程，它们在哪里？

器件	编译配置	位置
F28003x	RAM、闪存	C2000Ware_5_02_xx_xx > driverlib > f28003x > examples > flash
F28P65x	RAM、闪存	C2000Ware_5_02_xx_xx > driverlib > f28p65x > c28x > examples > flash

问题：如果 EEPROM 工程遇到错误，我首先应该检查什么？

回答：

- 查看配置文件 (EEPROM_Config.h、EEPROM_PingPong_Config.h)，然后检查为以下项目提供的选项：器件差异、编程模式（64 位与页）、闪存组选择、闪存扇区大小、闪存扇区数量、EEPROM 组数量、EEPROM 页数量和 EEPROM 页的数据大小。此外，还应检查主程序文件（EEPROM_Example.c、EEPROM_PingPong_Example.c），查看是否将正确的闪存扇区位置用于 EEPROM 仿真。如果提供了错误的第一个和最后一个扇区值，则会发生错误并在 EEPROM_Config_Check 函数中看到。EEPROM_Config_Check 函数将提供一般信息用于错误检查。
- 确保针对为器件的 EEPROM 仿真选择的相应扇区启用/禁用保护掩码；有关更多信息，请参阅器件的闪存 API 参考指南。
- 要检查的程序的一个区域是链接器命令文件 - 确保所有闪存部分都与 128 位边界对齐。在 SECTIONS 中，在将段分配给闪存的每一行之后添加一个逗号和“ALIGN(8)”。

12 结语

本应用报告证明了 F28P65x 第 3 代 C2000 实时控制器能够利用其内部的闪存来模拟 EEPROM，从而实现了系统内存储，并减少对外部元件的需求。这在很大程度上取决于代码大小以及是否有额外的闪存扇区可供使用。本文还为设计人员提供了一个现成的驱动程序，使用闪存 API 库加快设计速度并简化设计工作。

13 参考资料

- 德州仪器 (TI)，*第 2 代 C2000 实时 MCU 的 EEPROM 仿真*
- 德州仪器 (TI)，*TMS320F28P65x 闪存 API 版本 3.02.00.00 参考指南*
- 德州仪器 (TI)，*TMS320F28P65x 实时微控制器*
- 德州仪器 (TI)，*TMS320F280013x 实时微控制器*

14 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from Revision * (November 2023) to Revision A (April 2024)	Page
• 添加了 F28003x 示例的工程路径。	1
• 更新了注释以包含 TMS320F28003x 器件	2
• 添加了 F28003x 示例的工程路径	6
• 更新了 F28P65x 代码块并添加了 F28003x 代码块比较	8
• 更新了 F28P65x 代码块并添加了 F28003x 代码块比较	29
• 更新了单存储单元仿真的 F28P65x 工程文件名并更新了相关函数和说明	49
• 添加了包含 F28P65x 和 F28003x 示例工程路径和一般用户问题的故障排除章节	50

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265