Application Note

适用于 F29H85x 的 EEPROM 仿真驱动程序指南

Alex Wasinger

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 F29H85x SDK 1.00.00.00（或更高版本）中找到，路径如下：C:\ti\f29h85x-sdk_1_00_00_00\examples\driverlib\single_core\flash。

内容

插图清单

商标

1 简介

F29H85x 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。不过，此功能可以在 C2000 MCU 上模拟，因为它具备针对闪存存储器进行在线编程的功能。本应用报告使用片上闪存存储器的扇区演示了此功能。请注意，至少一个完整的闪存扇区会用于仿真；因此，该扇区不可用于存储应用程序代码。

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 与闪存的最大区别在于擦除操作：EEPROM 可以擦除任何特定的单独字节，而闪存必须清除至少一个整个扇区。

闪存写入和擦除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。任何字节都可以被覆盖，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。F29H85x 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* 部分中的“闪存参数”部分。

3 概述

本文档描述了两种适用于 F29H85x 器件的 EEPROM 仿真实现：单存储单元和乒乓。每种实现都支持两种编程模式：页面编程模式和 64 位编程模式。首先，我们将介绍单存储单元实现，然后介绍乒乓实现。

每种实现都支持多个用户可配置的 EEPROM 变量。这些变量在 节 6.1 中有详细介绍。

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，因此必须保留完整的闪存扇区用于 EEPROM 仿真。闪存扇区的大小因 C2000 器件型号而异。

每个 EEPROM 工程都有两种编程模式：64 位模式和页面模式：

- **64 位模式：**无页面或页面跟踪开销，最大 EEPROM 大小为 64 位
- **页面模式：**将所选闪存扇区细分为存储更新迭代后的 EEPROM 的“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。

在指定的 Flash 存储器已满时，这两种实现方式（单存储单元和乒乓）处理擦除过程的方式不同。后续章节会介绍这些过程。

3.2 单存储单元方法

如果使用单存储单元 EEPROM 仿真，在 EEPROM 单元已满并且还有更多数据要写入的情况下，则擦除 EEPROM 单元并将新数据编程到闪存中。下图中描绘了此行为：

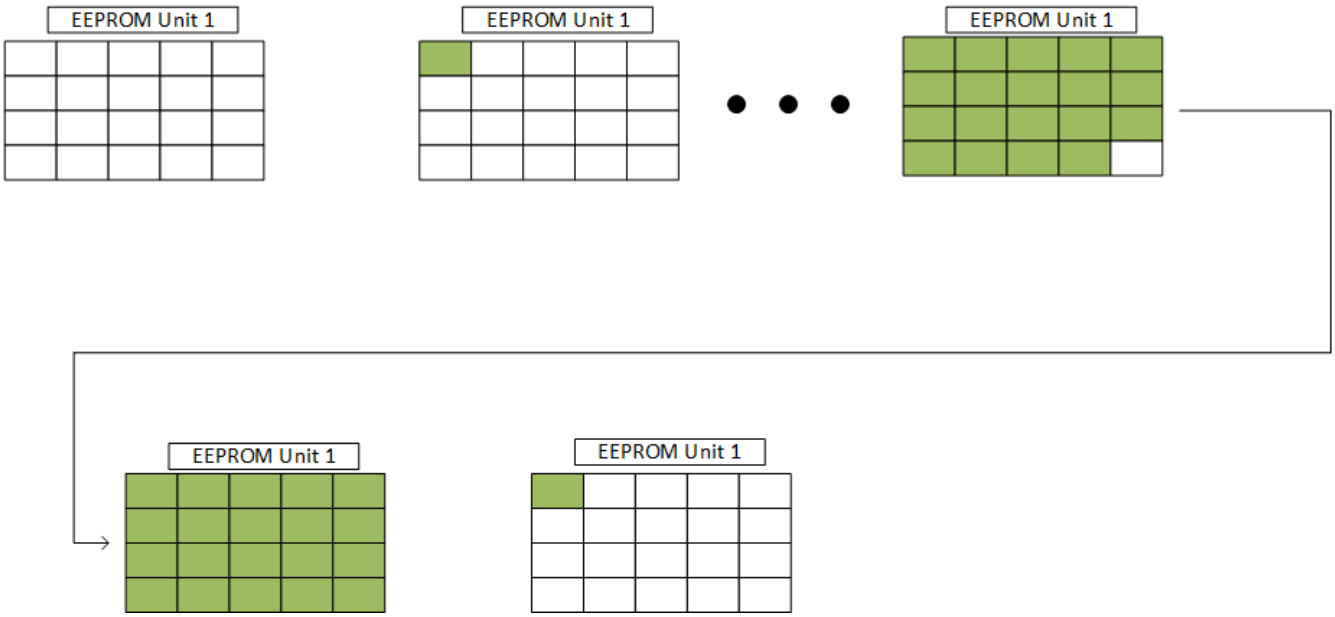

图 3-1. 单存储单元行为

3.3 乒乓方法

如果使用乒乓仿真，则指定第二个闪存存储器，并将仿真分为**活动单元**和**非活动单元**。如果活动单元已满且 EEPROM 已更新，单元名称将交换，并且数据将写入新的活动单元。对数据进行编程后，非活动单元会被安全擦除，无需担心数据丢失。

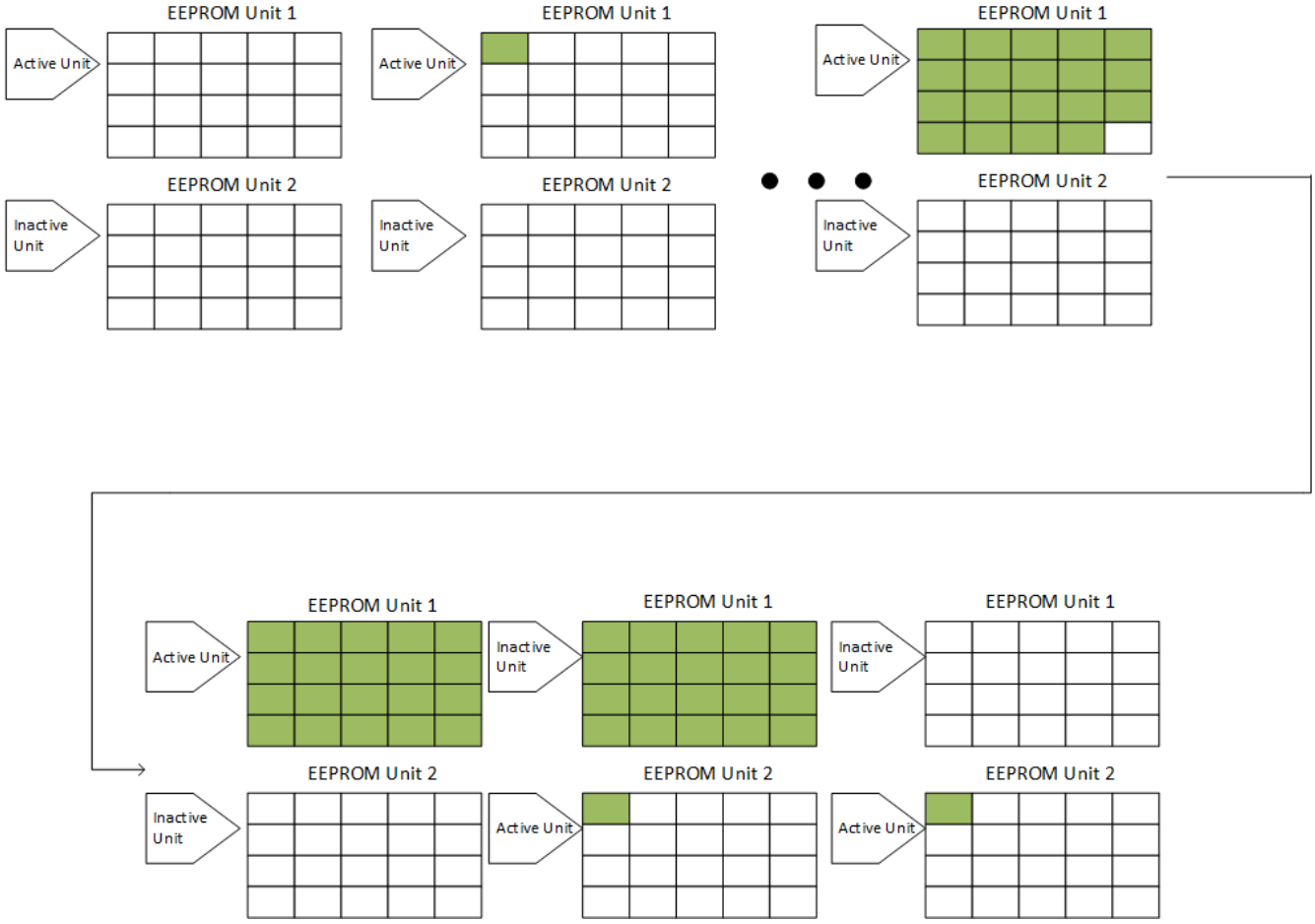

图 3-2. 乒乓行为

3.4 创建 EEPROM 节（页）和页标识

为了支持具有不同数据大小（大于 64 位）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。对于单存储单元和乒乓实现，仿真的这一方面是相同的。

为仿真所选的闪存扇区被细分为 EEPROM 组，然后分成页面。如图 3-3 所示。

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

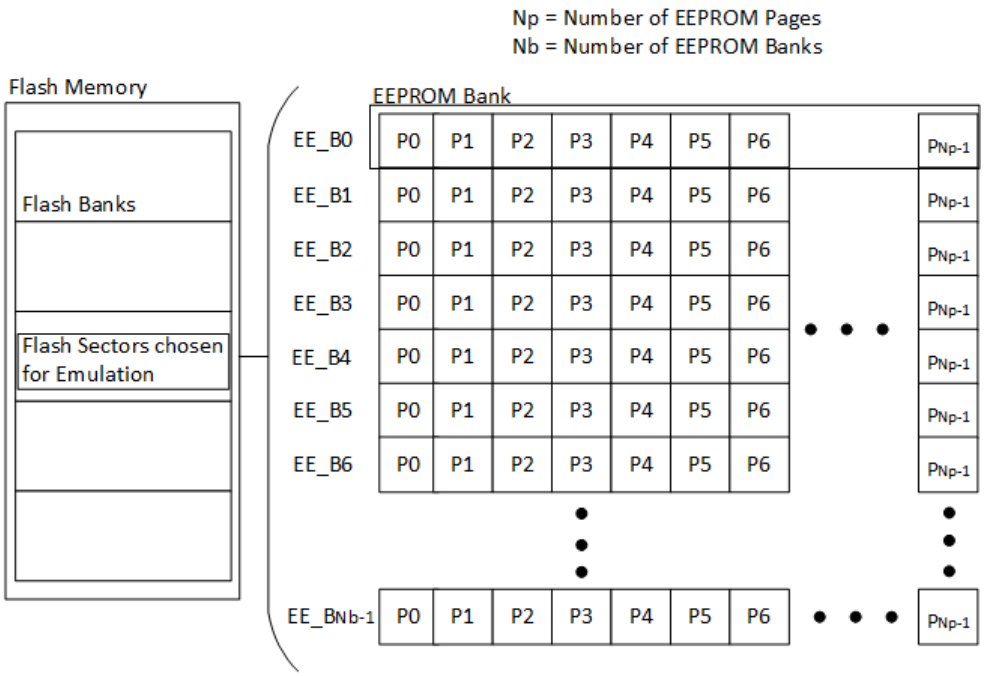

图 3-3. 组分区

应用通过状态代码跟踪正在使用的组和页面。有关更多信息，请参阅节 5.2.2.1。

为了跟踪 EEPROM 组的状态（空、当前或已满），需要保留每个组的前 16 个字节（128 位）。切换到新组时，之前和新 EEPROM 组的状态都会更新以反映新状态。

页面以类似的方式进行处理：保留每页的前 16 个字节（128 位），以确定状态是空、当前还是已使用。每当新数据写入页面时，前一页和新页的状态都会更新。

要将 EEPROM 组或页面标记为当前组或页面，可以向首个 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，可以向第二个 64 位写入适当的状态代码。

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。尽管仅显示了第 0 页和第 1 页，但应注意第 2 页至第 (N-1) 页与第 1 页相同。

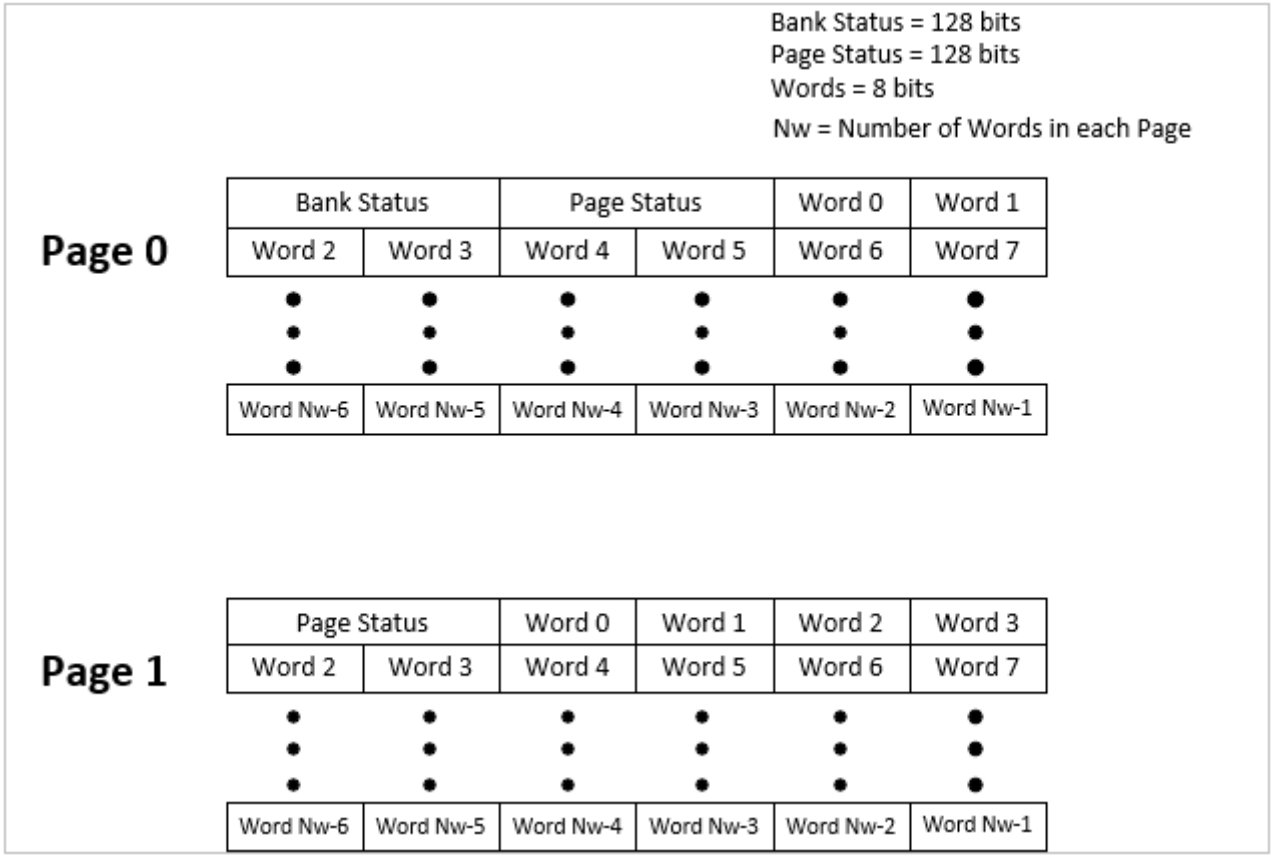

图 3-4. 页布局

4 软件说明

本应用报告随附的软件包含用于 F29H85 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。

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 F29H85x 提供的头文件和闪存 API 库。如需获取示例代码，请查看 F29H85x SDK 目录。完整路径为：f29h85x-sdk_1_00_00_00\examples\driverlib\single_core\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。有关所需步骤的完整列表，请查看 *F29H85x 闪存 API 参考指南*。

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 6.2.1.2。

此时，可以开始编程了。用户数据被写入闪存，并在随后被回读。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。

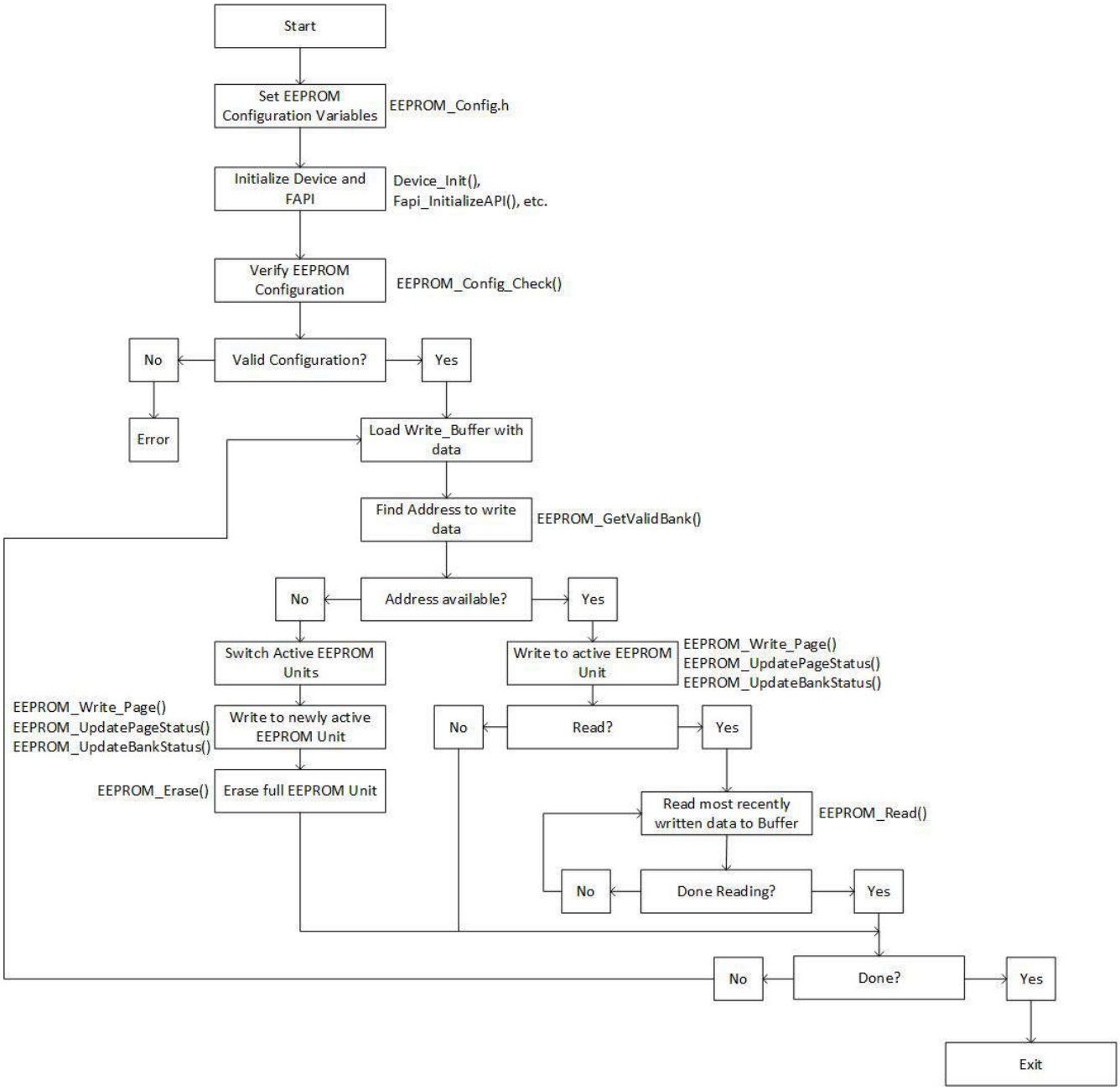

图 4-1. 软件流程

5 单存储单元仿真

单存储单元 EEPROM 仿真仅使用一组闪存扇区，因此没有非活动 EEPROM 单元。下面记录了该函数的行为，以及与 EEPROM_Erase() 函数中的乒乓的主要区别。

5.1 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。这些变量位于 EEPROM_Config.h 和 F29H85x_EEPROM.c 中。

5.1.1 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：

- 在页面模式和 64 位模式之间选择，这两种模式互斥

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

- 选择要仿真的 EEPROM 组的数量。

```
#define NUM_EEPROM_BANKS 8
```

- 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（单位为字节）。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数字的最接近的八的倍数。例如，每页六个字节的指定大小将被编程为每页八个字节，最后两个字节被视为 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 F29H85x_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}。

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

有效的配置具有以下属性：

- 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
  - F29H85x 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 *F29H85x 闪存 API 参考指南* (SPRUJE7)。

有关无效或危险配置的更多详细信息，请参阅节 5.2.1.2。

用户还可以在擦除后启用空白检查：

```
uint8_t Erase_Blank_Check = 1;
```

5.2 EEPROM 函数

下面列出了实现所需的函数。这些函数均包含在 F29H85x_EEPROM.c 或 F29H85x_EEPROM_Example.c 文件中。

初始化 + 设置

- Configure_Device()
- EEPROM_Config_Check()

页面模式

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

64 位模式

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

两种

- EEPROM_Erase()
- EEPROM_Read(uint8_t* Read_Buffer)

实用程序

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

后续章节会详细讨论上述每个函数。

5.2.1 初始化和设置函数

5.2.1.1 Configure_Device

此函数包含所有标准器件和 FlashAPI 设置。

首先，函数对器件及其外设进行初始化。

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

Device_initGPIO();

Interrupt_initModule();

Interrupt_initVectorTable();

__asm(" ENINT")
```

然后，它请求获取闪存信标并使用所选的用户配置对闪存 API 进行初始化。

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

u32UserFlashConfig = Fapi_getUserConfiguration(BankType, FOTAStatus);

Fapi_SetFlashCPUConfiguration(u32UserFlashConfig);

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

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

5.2.1.2 EEPROM_Config_Check

EEPROM_Config_Check() 函数可执行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。在 F29H85x 上，当前仅支持数据闪存。

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

第二，检查选择用于仿真的闪存扇区的有效性。这用于检查以下内容：

- 选择用于仿真的闪存扇区数量是否超过器件上的数量，以及是否至少选择了一个闪存扇区

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

NUM_EEPROM_SECTORS = NUM_EEPROM_SECTORS_1;

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

- 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;
    }

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

如果使用页面模式，还将检查以下各项：

- EEPROM 组 + 页面的总大小是否适合所选的闪存扇区

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

uint32 Available_Words = NUM_EEPROM_SECTORS * FLASH_SECTOR_SIZE;

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

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，闪存中是否会保留一个或多个 EEPROM 组的空间

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

- 如果每个页面包含少于或等于 64 位（8 个字节）（这会浪费空间，因为 64 位模式可以产生同样的效果，而且不会产生状态代码开销）

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

- 如果使用 32-127 范围内的扇区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。组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信息，请参阅 *F29H85x 闪存 API 参考指南*。

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

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


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

}


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

5.2.2 页面模式函数

5.2.2.1 EEPROM_GetValidBank

EEPROM_GetValidBank() 函数的功能是查找当前 EEPROM 组和页面。EEPROM_Write_Page() 和 EEPROM_Read() 函数都会调用此函数。图 5-1 展示了搜索当前 EEPROM 组和页面所需的总体流程。

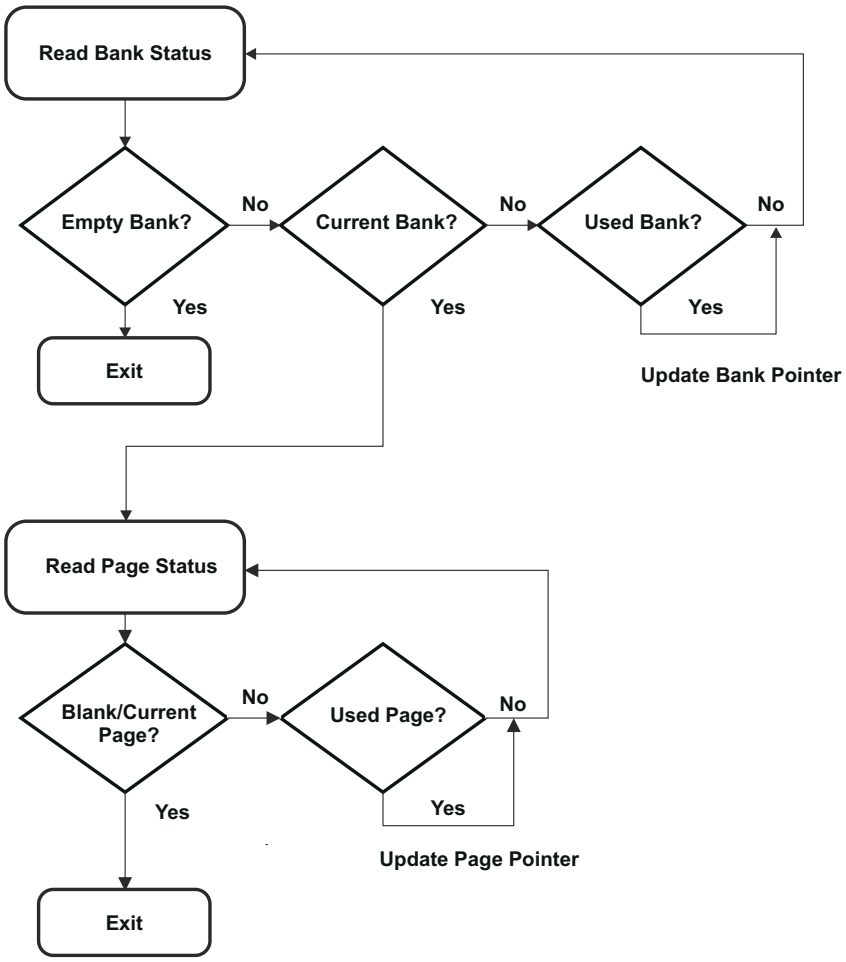

图 5-1. GetValidBank 流程

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 图 5-1 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，程序会检查当前 EEPROM 组是否为空。如果是，则表示该 EEPROM 组尚未使用，无需进一步搜索。

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

```
    return;
}
```

如果未遇到空的 EEPROM 组，程序会检查该组是否被标记为当前。如果是，则会更新 EEPROM 组计数器并将页面指针设置为 EEPROM 组的第一页，以便在稍后的循环中进一步调整。然后退出该循环，因为不需要进一步的搜索。

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

最后，程序会检查 EEPROM 组是否已使用。如果是，EEPROM 组指针会更新到下一个 EEPROM 组，然后循环继续。

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

找到当前 EEPROM 组后，即可找到当前页面。页面可能具有三种不同的状态：空白、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 0xA5A5A5A5A5A5A5A5、其余 64 位被设置为 1 表示。已使用的 EEPROM 页面由 128 位全被设置为 0xA5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5 表示。如果需要，可以在 EEPROM_Config.h 中更改这些状态代码。

首先，程序会检查该页面是空白页面还是当前页面。如果是这样，则表明已找到正确的页面并退出循环。

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

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

如果不是空白或当前页面，则必须使用该页面，并且页面指针会更新到下一个页面以测试其状态。

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

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。如果是这样，则最后一个组和页面会被标记为已满，以确保完整性，并擦除用于仿真的扇区。

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

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 组和页面的数量将取决于应用配置。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。

为防止从已满 EEPROM 单元读取时发生过早擦除，如果已设置 Read_Flag，则不执行此检查。

如上所示，如果存储器已满，在调用 EEPROM_Erase() 函数并复位组和页面指针之前，请将最后一个页面和组标记为已使用，以确保完整性。

5.2.2.2 EEPROM_UpdateBankStatus

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

```
uint8_t Current_Bank_Status = *(Bank_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。

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

    EEPROM_Write_Buffer(Bank_Pointer, Bank_Status);

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

如果状态不为空，则接下来检查是否存在已满的 EEPROM 组。在这种情况下，当前 EEPROM 组的状态将会更新为已满，并且下一个组的状态将设置为当前，以便允许对下一个 EEPROM 组进行编程。最后，页面指针会更新为新 EEPROM 组的第一页。

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

    EEPROM_Write_Buffer(Bank_Pointer + WRITE_SIZE_BYTES, Bank_Status);

    Bank_Pointer += Bank_Size;

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

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

}
```

5.2.2.3 EEPROM_UpdatePageStatus

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

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

如果此状态表示该页为空，函数则会退出。页面状态将在 EEPROM_Write_Page() 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：

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

Fill_Buffer(Page_Status, status_buffer_len, CURRENT_PAGE);

EEPROM_Write_Buffer(Page_Pointer + WRITE_SIZE_BYTES, Page_Status);

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

5.2.2.4 EEPROM_UpdatePageData

EEPROM_UpdatePageData() 函数的功能是更新 EEPROM 页数据。此函数由 EEPROM_Write_Page() 函数调用，并以 64 位的增量将页面编程到闪存中。

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

如果编程成功，则会将页面标记为当前并清除 Empty_EEPROM 标志。

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

Empty_EEPROM = 0;
```

5.2.2.5 EEPROM_Write_Page

EEPROM_Write_Page() 函数的功能是将数据编程到闪存中。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。然后，将上一页标记为已使用；如果组发生更改，则更新 EEPROM 组状态。接下来，在 EEPROM 页面数据更新期间对数据进行编程。

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

5.2.3 64 位模式函数

5.2.3.1 EEPROM_64_Bit_Mode_Check_EOS

EEPROM_64_Bit_Mode_Check_EOS() 确定 EEPROM 单元是否已满；如果已满，则将其擦除。

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。

```
uint8_t* End_Address = (uint8_t*) END_OF_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 8 个字节会超出结束地址，则表明 EEPROM 单元已满。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 - WRITE_SIZE_BYTES)
{
    EEPROM_Erase();
}
```

5.2.3.2 EEPROM_Write_64_Bits

EEPROM_Write_64_Bits() 函数的功能是面向存储器进行八字节编码。第一个参数 Num_Bytes 允许用户指定将写入多少个有效字。数据字会被分配给 Write_Buffer 的第一个索引。如果函数调用中指定的字节少于四个，则缺少的字节将用 0xFF 填充。这样做是为了符合 ECC 要求。

首先，程序会检查 EEPROM 是否已满。

```
EEPROM_64_Bit_Mode_Check_EOS();
```

接下来，如果指定的字节少于 8，则写入缓冲区中将填充 1。

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

然后，对数据进行编程，并且指针递增到下一个空位置

```
EEPROM_Write_Buffer(Bank_Pointer, Write_Buffer);

Empty_EEPROM = 0;

Bank_Pointer += WRITE_SIZE_BYTES;
```

备注

在执行 RESET_BANK_POINTER 设置指针之前，无法使用该函数。如果在此之前执行，则会生成未知结果。

5.2.4 两种模式下使用的函数

5.2.4.1 EEPROM_Erase

EEPROM_Erase() 函数的功能是擦除用于仿真的扇区并将组和页面指针复位。必须至少擦除一个完整的扇区，部分擦除不受支持。擦除之前，请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。

该函数为 EEPROM 单元配置写入/擦除保护掩码，然后调用 Erase_Bank 函数，最后将指针复位到组开头。

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

5.2.4.2 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。

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。

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

页面模式：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。

```
EEPROM_GetValidBank(1);

Page_Pointer += WRITE_SIZE_BYTES*2;

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

64 位模式：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。

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

5.2.5 实用功能

5.2.5.1 EEPROM_Write_Buffer

EEPROM_Write_Buffer() 将指向闪存地址的指针作为写入位置，并将指向 64 位写入缓冲区的指针作为输入。该函数会调用所有必要的 FlashAPI 函数，将写入缓冲区提交到位于指定地址的闪存。

首先，函数会清除 FSM 状态并设置适当的保护掩码。

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

ClearFSMStatus();

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

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

然后，函数会将来自写入缓冲器的数据编程到闪存中。

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

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

最后，函数会检查是否存在任何编程错误并验证写入的数据是否正确。

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

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

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

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

5.2.5.2 Erase_Bank

Erase_Bank 函数利用闪存 API 来擦除已满 EEPROM 单元。此函数只是围绕闪存 API 的包装程序，并且保护掩码已在 EEPROM_Erase() 函数中设置。

首先，函数会清除 FSM 状态并将保护掩码复制到闪存 API 中。

```
ClearFSMStatus(FLASH_BANK_SELECT, u32UserFlashConfig);

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

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

然后，函数会擦除闪存并检查是否存在编程错误。

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

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

if (oReturnCheck != Fapi_Status_Success)
    Sample_Error();

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

最后，如果已设置 Erase_Blank_Check，则执行空白检查。

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

5.2.5.3 Set_Protection_Masks

Set_Protection_Masks() 是一个围绕 Configure_Protection_Masks() 的简单包装器，用当前活动单元的正确值来更新全局掩码变量（WE_Protection_A_Mask 和 WE_Protection_B_Mask）。

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

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

5.2.5.4 Configure_Protection_Masks

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 函数实现的更多信息，请参阅 *F29H85x 闪存 API 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。

```
uint64 Protection_Mask_Sectors = 0;

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

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

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

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

return Protection_Mask_Sectors;
```

5.2.5.5 Fill_Buffer

Fill_Buffer() 是一个非常基本的辅助函数，用于填充各种状态和写入缓冲区。它将目标缓冲区、其长度和一个值作为输入，并用该值填充缓冲区。

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

5.2.5.6 ClearFSMStatus

ClearFSMStatus() 函数负责清除之前闪存操作的状态。该函数必须按原样使用。

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_StatusType  oReturnCheck;

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

oReturnCheck = Fapi_issueAsyncCommand(u32StartAddress, u32UserFlashConfig, Fapi_ClearStatus);

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

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

5.3 测试示例

提供的示例使用 F29H859TU8 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 F29H859TU8 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 F29H85x_EEPROM_Example.pjt。
4. 通过选择“Project”→“Build Project”构建工程。
5. 在资源管理器中右键单击工程即可将其启动，然后选择“调试工程”。
6. 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。

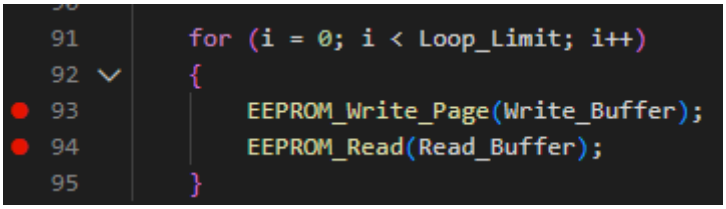

图 5-2. 断点

7. 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。

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

图 5-3. 对 EEPROM 进行写入

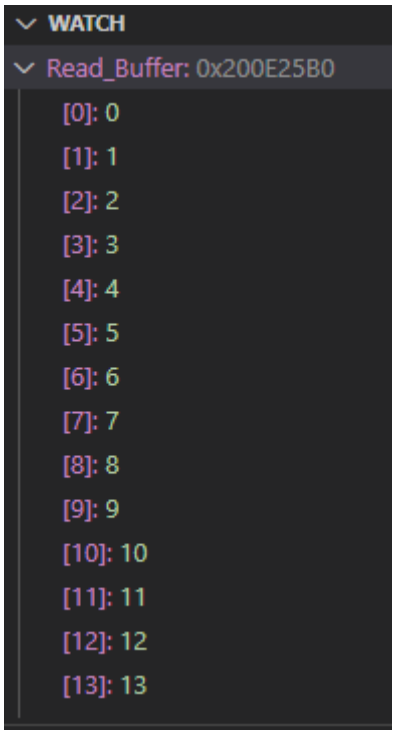

图 5-4. 读数据

8. 继续从断点运行到断点，直到程序运行完成或 EEPROM 已满。
9. 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。

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

图 5-5. 擦除已满 EEPROM 单元

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

图 5-6. 对 EEPROM 进行写入

10. 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，请取消对 _64_BIT_MODE 指令进行注释并对 PAGE_MODE 指令进行注释，以更改 EEPROM_Config.h 文件中的定义。

6 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。

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。这些变量可在 EEPROM_PingPong_Config.h 和 F29H85x_EEPROM_PingPong.c 中找到。

6.1.1 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：

- 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 5
```

- 选择每个 EEPROM 页面中包含的数据空间大小（单位为字节）。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数字的最接近的八的倍数。例如，每页六个字节的指定大小将被编程为每页八个字节，最后两个字节被视为 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 F29H85x_EEPROM_PingPong.c

在 F29H85x_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}。

```
uint32 FIRST_AND_LAST_SECTOR[2][2] = {{0,0},{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
  - F29H85x 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 *F29H85x 闪存 API 参考指南* (SPRUJE7)。

有关无效或危险配置的更多详细信息，请参阅节 6.2.1.2。

用户还可以在擦除后启用空白检查，并选择在哪个 EEPROM 单元中开始执行仿真操作。

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

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。

6.2 EEPROM 函数

下面列出了实现所需的所有函数，均包含在 F29H85x_EEPROM_PingPong.c 或 F29H85x_EEPROM_PingPong_Example.c 文件中。

初始化和设置函数

- Configure_Device()
- EEPROM_Config_Check()

页面模式函数

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

64 位模式函数

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

同时使用

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

实用功能

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

后续小节中会详细介绍上述每个函数。

6.2.1 初始化和设置函数

6.2.1.1 Configure_Device

此函数包含所有标准器件和 FlashAPI 设置。

首先，函数对器件及其外设进行初始化。

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

Device_initGPIO();

Interrupt_initModule();

Interrupt_initVectorTable();

__asm(" ENINT")
```

然后，它请求获取闪存信标并使用所选的用户配置对闪存 API 进行初始化。

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

u32UserFlashConfig = Fapi_getUserConfiguration(BankType, FOTAStatus);

Fapi_SetFlashCPUConfiguration(u32UserFlashConfig);

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

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

6.2.1.2 EEPROM_Config_Check

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。只有数据存储体是 F29x 上的有效选择。

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

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

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

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

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

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

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

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

- 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，请检查 EEPROM 组 + 页面的总大小是否适合所选的闪存扇区。

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

uint32_t Available_Words = NUM_EEPROM_SECTORS * FLASH_SECTOR_SIZE;

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

- 验证两个 EEPROM 单元是否没有重叠的保护掩码

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

如果检测到以下非致命条件之一，则会发出警告。每个标志对应于函数返回值中的位：

- 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

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

- 如果每个页面包含的位少于或等于 64（8 个字节）（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 <= WRITE_SIZE_BYTES)
{
    Warning_Flags += 2;
}
```

如果使用 32-127 范围内的扇区（对于 F29H85x 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 *F29H85x 闪存 API 参考指南*

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

6.2.2 页面模式函数

6.2.2.1 EEPROM_GetValidBank

EEPROM_GetValidBank() 函数的功能是查找当前 EEPROM 组和页面。EEPROM_Write_Page() 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。

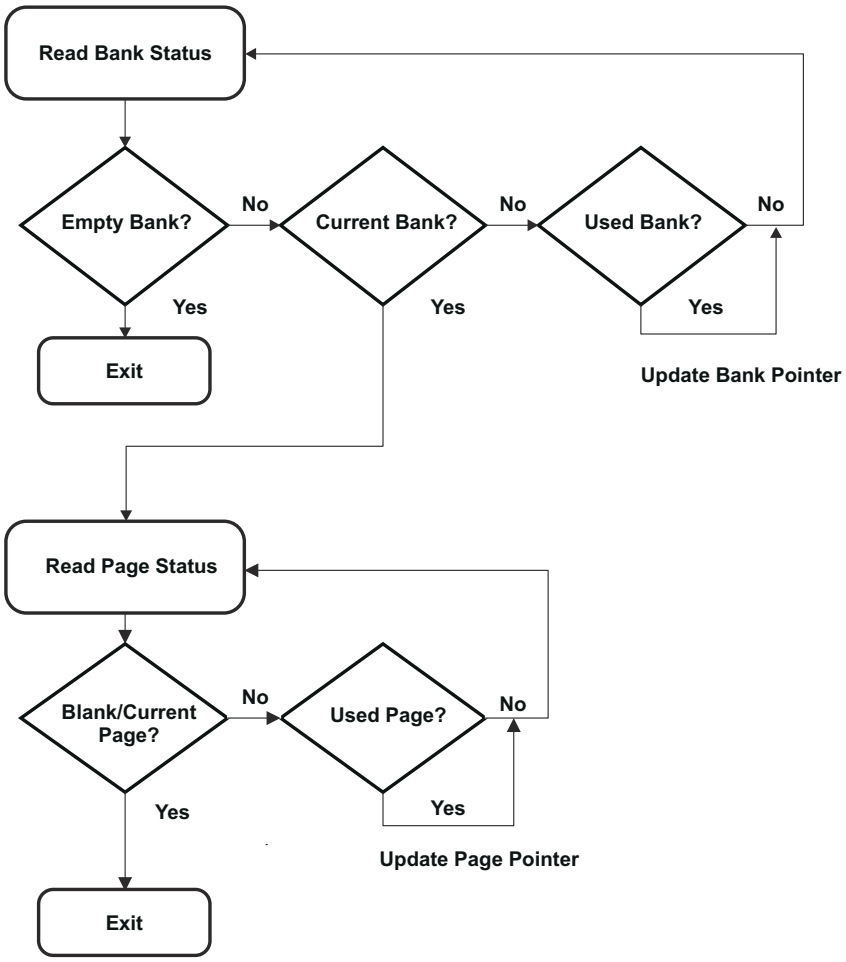

图 6-1. GetValidBank 流程

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，程序会检查是否有空的 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_in_Use == EMPTY_BANK)
{
    Bank_Counter = i;
    return;
}
```

如果未遇到空的 EEPROM 组，则接下来会检查当前状态。如果存在，则会相应地更新 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搜索。

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

最后，检查是否有“已使用”状态。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。

```
if (Bank_in_Use == CURRENT_BANK && Bank_Full == CURRENT_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 0xA5A5A5A5A5A5A5A5、其余 64 位被设置为 1 (0xA5A5A5A5A5A5A5A5FFFFFFFFFFFFFFFF) 表示。已使用的页面由全部 128 位被设置为 0xA5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5 表示。可以根据需要更改这些值。

首先查找“空”和“当前”状态。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。

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

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

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。

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

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。将完整单元中的最后一个组和页面标记为用于完整性，切换活动单元，重新配置写入/擦除保护掩码、并设置 Erase_Inactive_Unit 标志。

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

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

   RESET_BANK_POINTER;
   RESET_PAGE_POINTER;
}
```

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。

6.2.2.2 EEPROM_UpdateBankStatus

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

```
uint8_t Current_Bank_Status = *(Bank_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。

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

EEPROM_Write_Buffer(Bank_Pointer, Bank_Status);

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

如果状态不为空，则接下来检查是否存在已满的 EEPROM 组。在这种情况下，当前 EEPROM 组状态将更新以显示 EEPROM 组已满，并且下一个组的状态将设置为当前，以便允许对下一个 EEPROM 组进行编程。最后，页面指针会更新为新 EEPROM 组的第一页。

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

EEPROM_Write_Buffer(Bank_Pointer + WRITE_SIZE_BYTES, Bank_Status);

Bank_Pointer += Bank_Size;

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

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

6.2.2.3 EEPROM_UpdatePageStatus

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

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

如果此状态表示该页为空，函数则会退出。此状态将在 EEPROM_Write_Page() 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：

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

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

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

6.2.2.4 EEPROM_UpdatePageData

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

来自写入缓冲区的数据一次写入闪存 64 位，每次循环迭代都会计算偏移。

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

如果编程成功，则会将页面标记为当前并清除 Empty_EEPROM 标志。代码如下所示：

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

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。如果是这样，函数会擦除单元并清除标志。

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

6.2.2.5 EEPROM_Write_Page

EEPROM_Write_Page() 函数的功能是将数据编程到闪存中。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(0);
EEPROM_UpdatePageStatus();
EEPROM_UpdateBankStatus();
EEPROM_UpdatePageData(Write_Buffer);
```

6.2.3 64 位模式函数

6.2.3.1 EEPROM_64_Bit_Mode_Check_EOS

EEPROM_64_Bit_Mode_Check_EOS() 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，则标记该单元以进行擦除，并将指针移动到另一个干净单元。

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

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

接下来，将 EEPROM 组指针与结束地址进行比较。如果写入 64 个新位会超出结束地址，则表示该单元已满并且需要进行擦除。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 - WRITE_SIZE_BYTES)
{
    EEPROM_ACTIVE_UNIT ^= 1;
    Set_Protection_Masks();
    Erase_Inactive_Unit = 1;
    RESET_BANK_POINTER;
}
```

6.2.3.2 EEPROM_Write_64_Bits

EEPROM_Write_64_Bits() 函数的功能是面向存储器进行 64 位（8 个字节）编码。第一个参数 Num_Bytes 允许用户指定将写入多少个有效字节。ECC 需要进行至少 64 位的写入操作。如果少于 8 个字节，数据将用 0xFF 填充，直到实现 64 位缓冲区。数据字节应分配给 Write_Buffer 的前 8 个位置，以供 Fapi_issueProgrammingCommand 函数使用。

首先，测试是否存在已满的 EEPROM 单元。

```
EEPROM_64_Bit_Mode_Check_EOS();
```

接下来，如果指定的字节少于 8，则写入缓冲区中将填充 1。

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

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。

```
EEPROM_Write_Buffer(Bank_Pointer, Write_Buffer);

Empty_EEPROM = 0;

Bank_Pointer += WRITE_SIZE_BYTES;
```

编程完成后，将检查 Erase_Inactive_Unit 标志。如果已设置，无效单元将被擦除并显示复位标志。

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

备注

在执行 RESET_BANK_POINTER 设置指针之前，无法使用该函数。如果在此之前执行，则会生成未知结果。

6.2.4 两种模式下使用的函数

6.2.4.1 EEPROM_Erase_Inactive_Unit

EEPROM_Erase_Inactive_Unit() 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。

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

Erase_Bank();

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

6.2.4.2 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。

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。

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

页面模式：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。

```
EEPROM_GetValidBank(1);

Page_Pointer += WRITE_SIZE_BYTES*2;

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

64 位模式：指针向后移动八个地址（共 64 位），读取缓冲区被数据填满。

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

6.2.4.3 EEPROM_Erase_All

EEPROM_Erase_All 函数在程序启动时运行，并计算组合的保护掩码，然后再擦除所有扇区并重置仿真中使用的所有指针。

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

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

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

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

WE_Protection_A_Mask = 0xFFFFFFFF ^ Combined_WE_Protection_A_Sectors;
```

```
WE_Protection_B_Mask = 0x00000FFF ^ Combined_WE_Protection_B_Sectors;

Erase_Bank();

Set_Protection_Masks();

RESET_BANK_POINTER;
RESET_PAGE_POINTER;
```

6.2.5 实用功能

6.2.5.1 EEPROM_Write_Buffer

EEPROM_Write_Buffer() 将指向闪存地址的指针作为写入位置，并将指向 64 位写入缓冲区的指针作为输入。该函数会调用所有必要的 FlashAPI 函数，将写入缓冲区提交到位于指定地址的闪存。

首先，函数会清除 FSM 状态并设置适当的保护掩码。

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

ClearFSMStatus();

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

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

然后，函数会将来自写入缓冲器的数据编程到闪存中。

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

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

最后，函数会检查是否存在任何编程错误并验证写入的数据是否正确。

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

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

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

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

6.2.5.2 Erase_Bank

Erase_Bank 函数利用闪存 API 来擦除已满 EEPROM 单元。此函数只是围绕闪存 API 的包装程序，并且保护掩码已在 EEPROM_Erase() 函数中设置。

首先，函数会清除 FSM 状态并将保护掩码复制到闪存 API 中。

```
ClearFSMStatus(FLASH_BANK_SELECT, u32UserFlashConfig);

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

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

然后，函数会擦除闪存并检查是否存在编程错误。

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

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

if (oReturnCheck != Fapi_Status_Success)
    Sample_Error();

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

最后，如果已设置 Erase_Blank_Check，则执行空白检查。

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

6.2.5.3 Configure_Protection_Masks

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用于仿真的扇区数量。有关 Fapi_setupBankSectorEnable() 函数实现的更多信息，请参阅 *F29H85x 闪存 API 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。

```
uint64_t Protection_Mask_Sectors = 0;

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

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

        Protection_Mask_Sectors = Protection_Mask_Sectors << 32;

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

return Protection_Mask_Sectors;
```

6.2.5.4 Set_Protection_Masks

Set_Protection_Masks() 是一个围绕 Configure_Protection_Masks() 的简单包装器，用当前活动单元的正确值来更新全局掩码变量（WE_Protection_A_Mask 和 WE_Protection_B_Mask）。

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

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

6.2.5.5 Fill_Buffer

Fill_Buffer() 是一个非常基本的辅助函数，用于填充各种状态和写入缓冲区。它将目标缓冲区、其长度和一个值作为输入，并用该值填充缓冲区。

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

6.2.5.6 ClearFSMStatus

ClearFSMStatus() 函数负责清除之前闪存操作的状态。

```
Fapi_FlashStatusType  oFlashStatus;
Fapi_StatusType  oReturnCheck;

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


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

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

6.3 测试示例

提供的示例使用 F29H859TU8 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 F29H859TU8 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 F29H85x_EEPROM_PingPong_Example.pjt。
4. 通过选择“Project”→“Build Project”构建工程。
5. 在资源管理器中右键单击工程即可将其启动，然后选择“调试工程”。
6. 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。

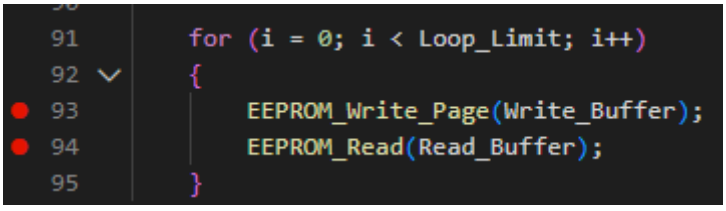

图 6-2. 断点

7. 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。

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

图 6-3. 对 EEPROM 单元进行写入

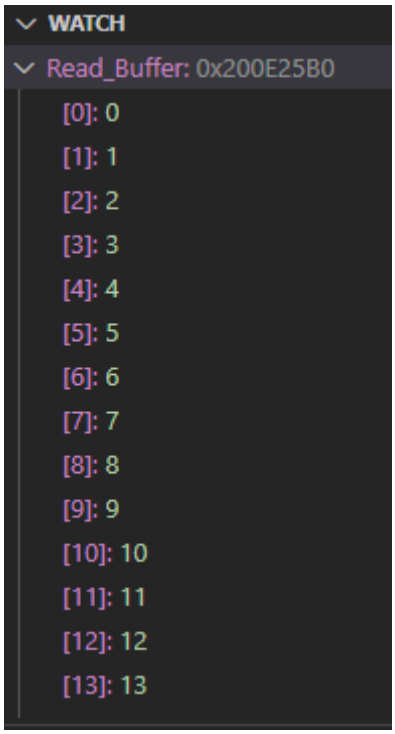

图 6-4. 读数据

8. 继续从断点运行到断点，直到程序运行完成或 EEPROM 已满。
9. 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。

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

图 6-5. 对新 EEPROM 单元进行写入

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

图 6-6. 擦除已满 EEPROM 单元

10. 可以根据需要在两个 EEPROM 单元之间重复该过程。

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_PingPong_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 F29H85x 上的单存储单元仿真，需要以下文件：

- F29H85x_EEPROM.c
- EEPROM_Config.h
- Device.c 和 device.h
- flash_programming_F29H85x.h
- F29H85x_NWFlashAPI_v21.00.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 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。编程功能一次对单字节进行操作，但每次都必须写入 64 位，以符合 ECC 要求。

8.1 闪存 API 检查清单

以下部分摘自 *F29H85x 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。

8.1.1 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。
- 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 *F29H85x 闪存 API 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

9 源文件清单

文件	功能	说明
F29H85x_EEPROM_PingPong.c	EEPROM_Config_Check() Configure_Protection_Masks() EEPROM_Write_Page() EEPROM_Read() EEPROM_Erase() Erase_Bank() EEPROM_GetValidBank() EEPROM_UpdateBankStatus() EEPROM_UpdatePageStatus() EEPROM_UpdatePageData() EEPROM_64_Bit_Mode_Check_EOS() EEPROM_Write_64_Bits() EEPROM_CheckStatus() ClearFSMStatus()	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
F29H85x_EEPROM.c	EEPROM_Config_Check() Configure_Protection_Masks() EEPROM_Write_Page() EEPROM_Read() EEPROM_Erase() Erase_Bank()EEPROM_GetValidBank() EEPROM_UpdateBankStatus() EEPROM_UpdatePageStatus() EEPROM_UpdatePageData() EEPROM_64_Bit_Mode_Check_EOS() EEPROM__64_Bits() EEPROM_CheckStatus() ClearFSMStatus()	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

10 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。

10.1 一般

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
F29H85x	RAM、闪存	f29h85x-sdk > examples > driverlib > single_core > 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）并检查提供的以下选项：编程模式（64 位与页面）、EEPROM 组数量、EEPROM 页面数量数以及 EEPROM 页面的数据大小。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。
- 确保为器件的 EEPROM 仿真所选择的相应扇区启用或禁用保护掩码。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)”。

11 结语

本应用报告证明了 F29H85x 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。

12 参考资料

- 德州仪器 (TI)：*F29H85x 闪存 API 参考指南*
- 德州仪器 (TI)：*F29H85x 和 F29P58x 实时微控制器技术参考手册*

重要通知和免责声明

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 产品的相关应用。 严禁以其他方式对这些资源进行复制或展示。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