在第三代 C2000 器件上实现 EEPROM 的模拟操作

Emma Wang *China FAE*

摘要

在很多应用场景里，应用程序都需要将少量系统相关数据（校准值、设备配置等信息）存储在非易失性存储器中，以便在系统重启后也可以重复使用这些数据或配置。在工业应用中，常使用 EEPROM（电可擦除可编程存储器）来存储用户配置数据，EEPROM 能够多次擦除和写入存储器的任意字节，即使系统断电，EEPROM 也能长时间保留数据。考虑到 C2000 芯片片上内置 Flash，出于 PCB 面积及成本的考虑，部分应用场景中也会使用片上 Flash 存储用户数据。本应用手册介绍了基于 TI 第三代 C2000 F280025C 内部 Flash 实现的 EEPROM 的功能，同时给出了具体的实现方法及示例代码，用户可参考本应用手册提供的方法，对 Flash 进行读取、写入、修改等操作，也可以快捷地拓展到 TI 其他的第三代 C2000 平台。

修改记录

Version	Date	Author	Notes
1.0	October 2022	Emma Wang	First release

目录

图

表

1. 用 Flash 模拟 EEPROM 的基本思想

第 3 代 C2000 实时 MCU 控制器 F28002x 系列是在 F2838x、F2837x 和 F28004x 系列的基础上衍生发展而来。F28002x 系列扩展了第 3 代中的产品组合，允许客户从高端扩展到中端及低端应用，同时保持功能差异化和高性能。 第 3 代 C2000 微控制器产品组合提供不同器件系列间的代码兼容性，从而减轻了开发人员在产品迭代、功能拓展开发过程中的工作量，实现了可持续的平台解决方案。

TMS32F28002x 是 C2000™ 实时微控制器系列中的一个器件，实时控制子系统基于 TI 的 32 位 C28x DSP 内核，可针对从片上 Flash 或 SRAM 运行的浮点或定点代码提供 100MHz 的信号处理性能。三角函数数学单元 (TMU) 和 VCRC（循环冗余校验）扩展指令集进一步增强了 C28x CPU 的性能，从而加快了实时控制系统关键常用算法的速度。高性能模拟模块集成在 F28002x 实时微控制器 (MCU) 上，并与处理单元和 PWM 单元紧密耦合，以提供更好的实时信号链性能。14 个 PWM 通道，可控制从三相逆变器到高级电源拓扑的各种功率级。通过加入可配置逻辑模块 (CLB)，用户可以添加自定义逻辑，还可将类似 FPGA 的功能集成到 C2000 实时 MCU 中。各种业界通用通信端口（如 SPI、SCI、I2C、PMBus、LIN 和 CAN）不仅支持通讯，还提供了多个引脚复用选项，可实现灵活的信号布局。快速串行接口 (FSI) 可跨隔离边界实现高达 200Mbps 的稳健通信。

本应用手册关注在使用 Flash 模拟 EEPROM 的操作，从而实现无需外部 EEPROM 实现用户数据的读写、修改和存储操作。

1.1. EEPROM 与 Flash 的区别

EEPROM 具有不同的容量，并通过串行接口（有时是并行接口）与主机微控制器连接。由于引脚数量需求最少， I2C 和 SPI 接口的 EEPROM 器件比较流行。 EEPROM 可以进行电气编程和擦除，并且大多数串行 EEPROM 允许逐字节编程或擦除操作。

与 EEPROM 相比，Flash 具有更高的密度，允许在芯片上实现更大的存储器阵列(扇区)。通过向每个单元施加时间控制的电压来执行闪存擦除和写入周期。在擦除条件下，每个单元(位)读取逻辑 1。因此，C2000 实时控制器的每个闪存位置在擦除时读取 0xFFFF。通过编程，可以将单元更改为逻辑 0。可以覆盖任何字以将位从逻辑 1 更改为 0。第 3 代 C2000 MCU 部件上的片上 Flash 需要 TI 提供的特定算法 (Flash API) 才能进行擦除和写入操作。

EEPROM 和闪存操作之间的主要区别在于写入和擦除时序。典型的 Flash 写入时间为 50us/16 位字；而 EEPROM 通常需要 5 到 10ms。 EEPROM 不需要 Page(扇区)擦除操作。可以擦除特定字节。Flash 的 Page 擦除时间通常较长。对于第 3 代 C2000 MCU，擦除时间的典型值为 50ms/8K 扇区。 Flash 电源在写入/擦除操作期间必须稳定。由于其不同的特性，使用闪存模拟 EEPROM 存在一些挑战。

1.2. 用 Flash 模拟 EEPROM 的特殊性

由于 Flash 具有整个扇区擦除的特性，每次擦除都是以扇区为单位，从而需要从 F28002X 中拿出一个单独的扇区(8K Byte, 4K Word)用于模拟 EEPROM。一个扇区可以划分为几个空间更小的存储空间(Page)，每一个 Page 的大小即可作为模拟的 EEPROM 存储空间大小。虽然 Flash 与 EEPROM 一样具有可重复擦写的特点，但是其寿命比 EEPROM 少，所以一般情况下会使用循环使用的方式。具体操作过程是，将需要写入的数据放置到 RAM 的数据缓存区(Write_Buffer)，然后调用 Flash API 将保存在 Write_Buffer 中的数据写入到 Flash 的第一个 Page(Page 0)中。在之后的循环中，应用程序会将保存的数据从 Page 复制到 RAM 中的缓冲区(Read_Buffer)。等到程序运行需要保存新的数据时候，应用程序会找到下一个 Page(Page 1)，并且将数据写入到该 Page 中。这个过程将一直循环，直到将规划 Flash 区域内所有的 Page 都用完为止。当最后一个 Page 的空间也被使用，此时程序将会把整个 Flash 扇区擦除，并从 Page0 开始进行新一轮的连续的 EEPROM 数据保存操作，如此往复循环。

模拟 EEPROM 除了支持每次写一个 Page 之外，也支持每次写 4 个 Word 空间的数值，但是由于 F28002X 的 Flash 使用了 Error Correction Code(ECC)进行保护，从而导致每次写的数据需要 64bit 对齐或者 128bit 对齐，所以在实际使用的时候，该模拟 EEPROM 底层驱动不支持单 Byte 操作，如果需要使用单 Byte 操作，需要读出、改写、再写入 flash。

1.3. 用 Flash 模拟 EEPROM 的基本思想

1.3.1. 用空间换次数的思想

用 Flash 模拟 EEPROM 就是利用 Flash 与 EEPROM 相近的特性，比如能够多次读写、掉电保持等。模拟的方式是从 F28002X 的 Flash 中，取出一个扇区(例程中取的是第 15 个扇区，该扇区的大小是 4k Word)，本文示例中，设置 EEPROM 的大小是 64 Word，如果直接用整个扇区只保存 64 Word 的数值，这会导致利用效率较低，更为重要的是使用每写一个 Page 之后就得将这个扇区擦除一次，这会严重影响 Flash 的使用寿命。考虑到这个情况，可以将整个扇区分为 7 个 Bank，每个 Bank 分为 8 个 Page(总共占有 4088 个 Word)，每写 56 次 Page 之后才会擦除一次 Flash 的扇区，大幅度提高整个扇区的使用寿命。根据 F280025C 的 datasheet，flash 扇区的擦写次数寿命是 20000 次，经过分区操作后，预计可将擦写次数拓展到 1120000 次。

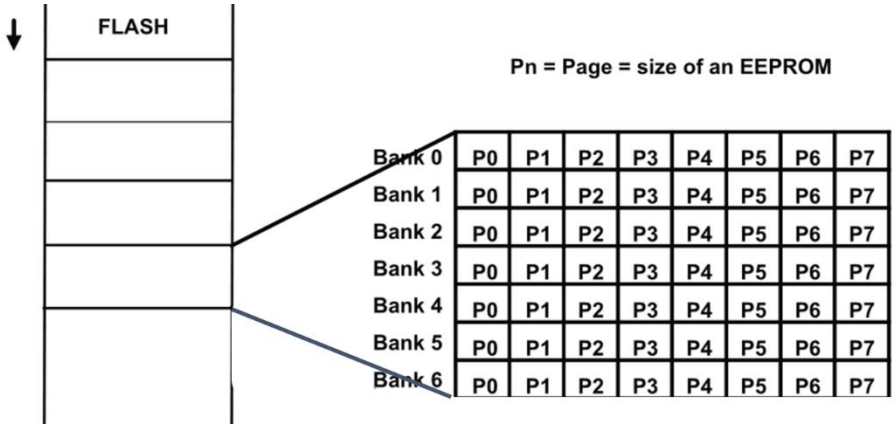


图 **1**. Page 和 Bank 划分示意图

Table 7-5. Flash Parameters

PARAMETER		MIN	TYP	MAX	UNIT
Program Time(1)	128 data bits + 16 ECC bits		150	300	µs
	8KB sector		50	100	ms
EraseTime(2) (3) at < 25 cycles	8KB sector		15	100	ms
EraseTime(2) (3) at 1000 cycles	8KB sector		25	350	ms
EraseTime(2) (3) at 2000 cycles	8KB sector		30	600	ms
EraseTime(2) (3) at 20K cycles	8KB sector		120	4000	ms
N_{wec} Write/Erase Cycles				20000	cycles
$t_{retention}$ Data retention duration at T_J = 85°C		20			years

图 **2**. F280025 Flash 擦写次数说明

1.3.2. 块和页的状态字的定义

F28002X 系列芯片的内部 Flash 受到了 ECC 的保护，使用纠错码(ECC) 来检测和纠正内存中发生的 n 位数据损坏。由于安全要求的不同，ECC 功能可以用于提高例如工业控制应用程序、关键存储数据的准确性及可靠性。但是这个功能对于模拟 EEPROM 操作也会带来一点限制，比如说在 ECC 使能的情况下(必须要使能)，对 Flash 每次需要操作 4 个 Word，而且在扇区擦除之后，每个 4 word 的空间就只能写一次。所以在实际使用过程中需要标注 Flash 的块和页是否已经写过，需要找到当前擦除所没有写过的区域进行写入，此处引入块和页的状态标记定义：没使用(Empty)、正在使用(Using)以及使用完毕(Used)三个状态组成，为了表示块或页的 3 种状态，本手册应用额外的两个 64 位的空间来存储状态，如图 3 所示。页和块的状态字由两个 64 位的空间构成，分别记为 status 0 和 status 1，不同 status 的组合对应的不同状态说明见表 1。

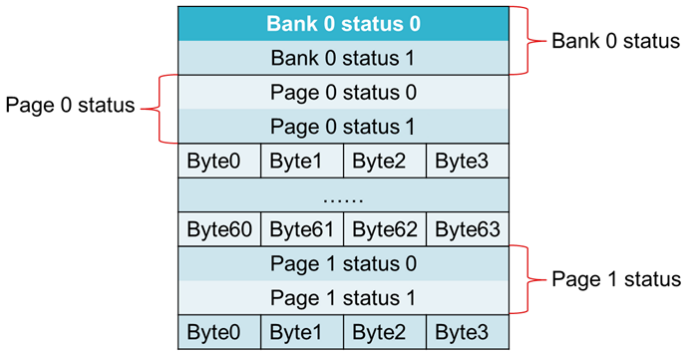


图 **3**. 块与页的状态字

表 1. 状态字以及状态

status 0	status 1	状态	说明
位全是 1	位全是 1	Empty	块或页还没有被使用
位全是 0	位全是 1	Using	块或页正在被使用，该页中的数据是有效
位全是 0	位全是 0	Used	块或页已经使用完毕

配合使用图 4 对表 1 进行说明，由于划分了不同的 Page，程序需要对当前使用中的 Page 进行标记。在程序运行的过程中，程序将最新数据写入 Page 区域的状态字设为 Using，则可知 Page C 目前正在被使用，并且程序如果需要读取 EEPROM 的数据，就会查到目前处于 Using 状态的 Page，然后将其中的数据读取出来。当又有一页新的数据需要写入到下一个 Page 中时，程序就会找临近的 Empty(也就是 Page D)，此时就会把 D 中的状态设为 Using(status 0 清零)，C 中的状态设为 Used(该 Page 的 status 1 清零)，所以当程序需要读取 EEPROM 数据的时候，就会查找 Using(Page D)标记的 Page，然后将该 Page 中的数据读回到 RAM。以上就是块或页状态使用的标志作用。

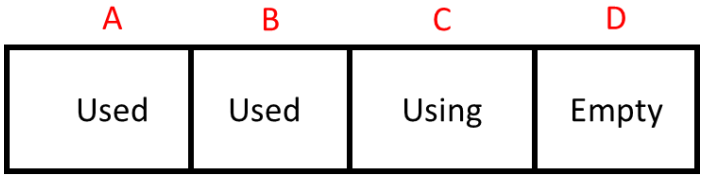


图 **4**. Page 的存储状态

1.4. 用 Flash 模拟 EEPROM 需要实现的功能

根据上面的分析，可以总结出模拟 EEPROM 要实现下面四个具体功能：

1) 能够读取前一个时刻写入的数据；
2) 能够向 Flash 的一个新 Page 中写入数据；
3) 在写的时候，如果最后一个 Page 都已经被占用，需要擦除整个扇区，重新从 Page A 开始；
4) 能够按特定地址读取之前存储的数据。

2. 用 Flash 模拟 EEPROM 的代码实现

在实现代码的过程中，为了保证代码具有可读性以及层次感的同时，方便用户不用关心底层的操作，本文使用分层的思想来实现代码，分层架构如图 5 所示。最底层的 LIB 库是 TI 官方提供的操作 Flash 的 API 函数，如果要写 Flash 中的特定地址空间，此时需要调用 LIB 库的函数进行，比如写 Flash 的操作就如图 6 所展示的那样，这也就是图 5 的①所实现的写 Flash 函数功能。由图 5 可知，与用户有关的是封装好的顶层 EEPROM 操作的 API 函数，本小节主要介绍顶层 API 函数的功能。

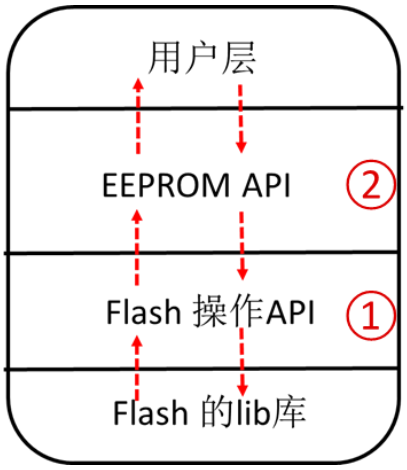


图 5. 模拟 EEPROM 函数的分层

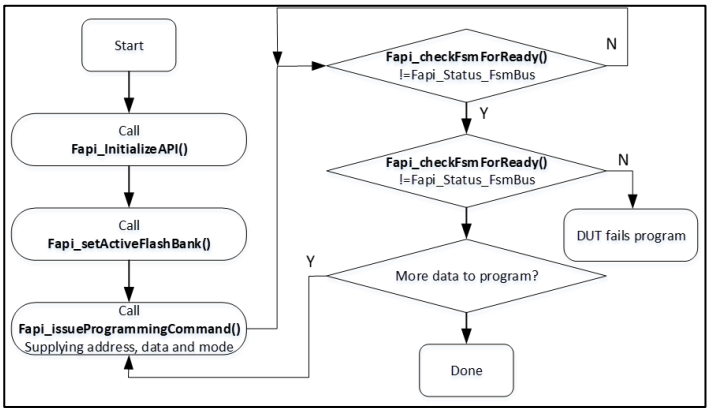


图 6. 写 Flash 的操作流程图

2.1. void EEPROM_init(void)

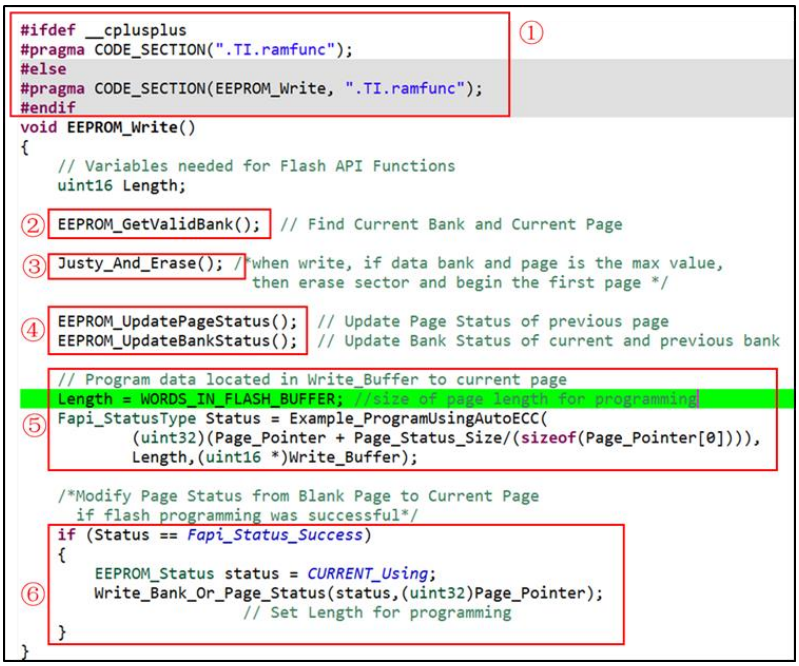
该函数主要功能则是初始化 Flash，设置 Flash 的操作频率以及激活 Flash 的操作，这个函数是操作 Flash 的必要条件，必须在操作 Flash 之前进行调用。

2.2. void EEPROM_Erase(void)

该函数则是擦除模拟 EEPROM 的扇区(所有的 Page)。根据章节 1.3 的说明，在写 Page 的时候，如果发现最后一个 Page 都已经被使用，说明所有的 Page 都已经被使用，为了能够将最新需要保存的数据写入到 Flash 中，需要先对 Flash 进行擦除，使新的数据从第一个 Page 开始写。上述的擦除过程需要调用此函数。

2.3. void EEPROM_Write(void)

该函数是写 EEPROM 整个 Page 的函数，是模拟 EEPROM 最核心的部分，该函数代码如图 7 所示，下面对这个函数进行说明。



```
#ifdef __cplusplus
#pragma CODE_SECTION(".TI.ramfunc");
#else
#pragma CODE_SECTION(EEPROM_Write, ".TI.ramfunc");
#endif
void EEPROM_Write()
{
    // Variables needed for Flash API Functions
    uint16 Length;

    EEPROM_GetValidBank(); // Find Current Bank and Current Page

    Justy_And_Erase(); /*when write, if data bank and page is the max value,
                         then erase sector and begin the first page */

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

    // Program data located in Write_Buffer to current page
    Length = WORDS_IN_FLASH_BUFFER; //size of page length for programming
    Fapi_StatusType Status = Example_ProgramUsingAutoECC(
            (uint32)(Page_Pointer + Page_Status_Size/(sizeof(Page_Pointer[0]))),
            Length,(uint16 *)Write_Buffer);

    /*Modify Page Status from Blank Page to Current Page
      if flash programming was successful*/
    if (Status == Fapi_Status_Success)
    {
        EEPROM_Status status = CURRENT_Using;
        Write_Bank_Or_Page_Status(status,(uint32)Page_Pointer);
                        // Set Length for programming
    }
}
```

图 **7**. EEPROM_Write 的示例代码

①是将该函数存放在段“.TI.ramfunc”中，由于函数会操作 Flash，为了防止一边在 Flash 中读取函数的代码一边改变 Flash 的寄存器状态，导致发生不可预知错误，需要在初始化的时候会将该函数代码复制到 RAM 中执行。

②从 7 个 Bank 中找到第一个 Empty/Using 的 Bank，同时，在此 Bank 中找到当前所指向的 Empty/Using 的 Page。

③如果判断所有的 Bank 和 Page 都被用完，该函数就会擦除整块扇区。

④如果当前的 Page/Bank 是空闲的，就什么也不操作，如果是 Using 的 Page/Bank，则将该 Page/Bank 的状态设为 Used 并且寻找下一个 Empty 的 Page/Bank。

⑤则是调用“Flash 操作 API”层的写 Flash 函数，该函数的写流程图如图 6 所示。

⑥则是在写成功的情况下，会将 Page 的状态设为 Using。

通过上面的解释可以看出该函数就是找到下一个未使用的 Page(如果所有的 Page 都使用完毕，则擦除扇区)，然后将用户存储在数组 Write_Buffer 中的数值写入到该 Page 中并且改变该 Page 的状态。

该函数的流程图如图 8 所示。

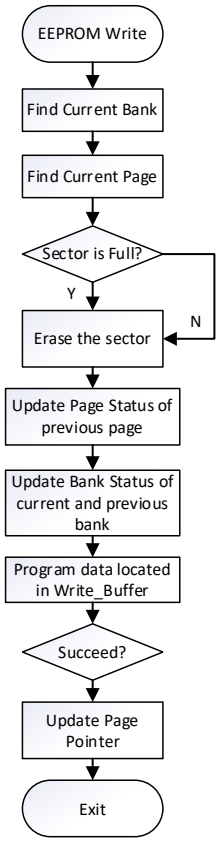


图 **8**. EEPROM_Write 的流程图

2.4. void EEPROM_Read(void)

该函数是读取最近一次写入到模拟 EEPROM 的 Page 中的数据，是通过指针的形式读取。该函数的流程图如图 9 所示。

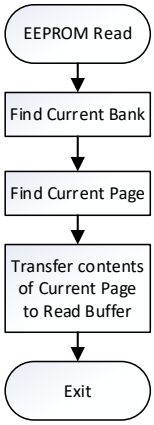


图 **9**. EEPROM_Read 的流程图

2.5. void EEPROM_GetSinglePointer(uint16 First_Call)

该函数则是找到下一个写 4 Word 数据的地址指针，在第一次写入数据的时候需要传入参数 1，使得函数查询到第一个 64 位数据全为 1 的地址。如果非第一次写入数据，则传入数据 0，函数将判断指针指向的地址是否已经是最大的地址值，如果是最大的地址值，则需要擦除扇区并且将指针指向扇区的开始地址处。

2.6. void EEPROM_ProgramSingleByte (uint64 data)

在一个 64 位对齐的地址写一个 64 位大小的数据。注意：由于该 Flash 带 ECC，因此每次操作无法只写一个 Word，这也是与外挂 EEPROM 芯片的区别。

3. 在应用代码中使用模拟 EEPROM 功能

3.1. 更改 Flash 的参数

在实际代码中，如果需要更改扇区(如从扇区 15 变成扇区 14)，每个 Page 的大小，块和页的数量等参数，可以在头文件”Meddle_Flash_Source.h”中进行更改，更改的部分如图 10 所示。

```
// Bank0 Sector start addresses
#define FlashStartAddress           0x80000U
#define Bzero_Sector0_start         0x80000U
#define Bzero_Sector1_start         0x81000U
#define Bzero_Sector2_start         0x82000U
#define Bzero_Sector3_start         0x83000U
#define Bzero_Sector4_start         0x84000U
#define Bzero_Sector5_start         0x85000U
#define Bzero_Sector6_start         0x86000U
#define Bzero_Sector7_start         0x87000U
#define Bzero_Sector8_start         0x88000U
#define Bzero_Sector9_start         0x89000U
#define Bzero_Sector10_start        0x8A000U
#define Bzero_Sector11_start        0x8B000U
#define Bzero_Sector12_start        0x8C000U
#define Bzero_Sector13_start        0x8D000U
#define Bzero_Sector14_start        0x8E000U
#define Bzero_Sector15_start        0x8F000U
#define FlashEndAddress             0x8FFF7U

//Sector length in number of 16bits
#define Sector8KB_u16length   0x1000U

//Sector length in number of 32bits
#define Sector8KB_u32length   0x800U

/*.........Define region of Emulating EEPROM.........*/
#define Flash_Region_of_Emulating_EEPROM Bzero_Sector15_start

// Length (in 16-bit words) of data buffer used for program
#define  WORDS_IN_FLASH_BUFFER    64
extern uint16   Buffer[WORDS_IN_FLASH_BUFFER];

#define Bank_Number ( 7 ) //the number of bank in EEPROM
#define Page_Number ( 8 ) //The number of page in EEPROM

#define Bank_Status_Size (8) // Status size
#define Page_Status_Size (8) //
// The register address of ECC Enable
#define ECC_Enable_REG_Addr (0x005FB00)
```

①
⑨
④
③
②
⑤
⑥
⑦
⑧

图 **10**. 用户层宏定义的修改

①是定义 F28002X 每个扇区的起始地址。

②是定义用于模拟 EEPROM 的扇区，目前选择的扇区是第 15 个扇区

③是定义该扇区有多少个 32bit(2 个 Word)的大小，目前第 15 个扇区有 4096 个 Word，2048 个 32 位的空间。

④是定义该扇区有多少个 Word，扇区 15 有 4096 个 Word。

⑤是定义每个 Page 的大小，目前每个 Page 有 64 个 Word，该数值大小需要是 4 的整数倍。

⑥是定义使用的扇区 15 被分成 7 个 Bank 和 8 个 Page，至于具体的数值大小需要根据扇区的大小来决定，本次使用的空间总大小为(Page_Number*(Page_Status_Size+⑤)+⑦)*Bank_Number = (8*(8+64)+8)*7=4088 Word。

⑦是定义目前 Bank 的状态和 Page 的状态用几个 Word 来表示，由于每个 Bank 或 Page 都是使用 2 个 64 位的空间表示，从而表示状态的空间都是 8 个 Word。

⑧是定义 ECC Enable 这个寄存器的地址，用于主函数中 Disable 或者 Enable ECC 的操作。

⑨是定义操作块区的最后一个地址，也就是扇区②的最后一个地址，用于判断是否是写到扇区的最后一个地址。

3.2. 移植到其他 C2000 平台

由于本代码采用分层处理，而且绝大部分的变量都是使用宏定义的方式，从而移植起来相对较为方便。需要注意的是，如果其他 C2000 平台操作 Flash 所使用的.lib 库中，擦除 Flash、写 Flash 的时序与 F28002X 不一致，则需要修改 EERPOM API，如图 5 所示。

下面，我们举例说明如何将 F28002X 的代码移植到新的平台 F28004X 上，由于第三代 C2000（这两个系列同为第三代 C2000）采用了统一的 Driverlib 函数的方式，F28004X 操作 Flash 的流程与 F28002X 一致，Flash 的 API 函数名称相同，从而不需要修改操作 Flash 的 API 函数。下面是 F28004X 的移植步骤。

1) 在 C2000Ware 中找到 F28004X 操作 Flash 的 example code，导入例程“flashapi_ex1_program_autoecc”。
2) 将例程“Flash_Emulated_EEPROM_f28002X”中的 user 文件夹复制到上面新建的工程中，且与 device 文件夹在同一路径下。
3) 添加.lib 库文件。将工程中的 F021_API_F28004x_FPU32.lib 剪切到...\user\Low_Level_Flash 文件夹下，删除原来的.lib 库。
4) 在 28004x_flash_api_lnk.cmd 文件中添加图 11 所示链接。

DataBufferSection : > RAMGS0, PAGE = 1, ALIGN(8)
StatusBufferSection : > RAMGS0, PAGE = 1, ALIGN(8)

```
    RUN = RAMLS03,
    LOAD_START(_RamfuncsLoadStart),
    LOAD_SIZE(_RamfuncsLoadSize),
    LOAD_END(_RamfuncsLoadEnd),
    RUN_START(_RamfuncsRunStart),
    RUN_SIZE(_RamfuncsRunSize),
    RUN_END(_RamfuncsRunEnd),
    PAGE = 0, ALIGN(4)

DataBufferSection     : > RAMGS0, PAGE = 1, ALIGN(8)
StatusBufferSection   : > RAMGS0, PAGE = 1, ALIGN(8)
```

图 **11**. 添加链接文件

5) 在 Meddle_Flash_Source.h 中将#include "F021_F28002x_C28x.h"改成#include "F021_F28004x_C28x.h"。
6) 将 flash_programming_f28004x.h 中的 Flash 地址复制到 Meddle_Flash_Source.h 中，即将图 12 中的 b) F28002X 的 Flash 地址定义替换成 a) F28004X 的 Flash 地址定义。

```
// Bank0 Sector start addresses
#define Bzero_Sector0_start         0x80000
#define Bzero_Sector1_start         0x81000
#define Bzero_Sector2_start         0x82000
#define Bzero_Sector3_start         0x83000
#define Bzero_Sector4_start         0x84000
#define Bzero_Sector5_start         0x85000
#define Bzero_Sector6_start         0x86000
#define Bzero_Sector7_start         0x87000
#define Bzero_Sector8_start         0x88000
#define Bzero_Sector9_start         0x89000
#define Bzero_Sector10_start        0x8A000
#define Bzero_Sector11_start        0x8B000
#define Bzero_Sector12_start        0x8C000
#define Bzero_Sector13_start        0x8D000
#define Bzero_Sector14_start        0x8E000
#define Bzero_Sector15_start        0x8F000
// Bank1 Sector start addresses
#define Bone_Sector0_start          0x90000
#define Bone_Sector1_start          0x91000
#define Bone_Sector2_start          0x92000
#define Bone_Sector3_start          0x93000
#define Bone_Sector4_start          0x94000
#define Bone_Sector5_start          0x95000
#define Bone_Sector6_start          0x96000
#define Bone_Sector7_start          0x97000
#define Bone_Sector8_start          0x98000
#define Bone_Sector9_start          0x99000
#define Bone_Sector10_start         0x9A000
#define Bone_Sector11_start         0x9B000
#define Bone_Sector12_start         0x9C000
#define Bone_Sector13_start         0x9D000
#define Bone_Sector14_start         0x9E000
#define Bone_Sector15_start         0x9F000
```

a) F28004X 的 Flash 地址定义

```
// Bank0 Sector start addresses
#define FlashStartAddress           0x80000U
#define Bzero_Sector0_start         0x80000U
#define Bzero_Sector1_start         0x81000U
#define Bzero_Sector2_start         0x82000U
#define Bzero_Sector3_start         0x83000U
#define Bzero_Sector4_start         0x84000U
#define Bzero_Sector5_start         0x85000U
#define Bzero_Sector6_start         0x86000U
#define Bzero_Sector7_start         0x87000U
#define Bzero_Sector8_start         0x88000U
#define Bzero_Sector9_start         0x89000U
#define Bzero_Sector10_start        0x8A000U
#define Bzero_Sector11_start        0x8B000U
#define Bzero_Sector12_start        0x8C000U
#define Bzero_Sector13_start        0x8D000U
#define Bzero_Sector14_start        0x8E000U
#define Bzero_Sector15_start        0x8F000U
```

b) F28002X 的 Flash 地址定义

图 **12**. Flash 地址定义的替换

7) 在当前工程中添加头文件 Meddle_Flash_Source.h 路径和 F28002x_EEPROM.h 路径。
8) 在工程 Flash_Emulated_EEPROM_f28002X 中将 main.c 和 main.h 复制到 flashapi_ex1_program_autoecc.c 所在的路径下，并且删除新工程中的 flashapi_ex1_program_autoecc.c 和 flash_programming_f28004x.h 文件。
9) 直接在新的工程中修改将 F28002x_EEPROM.c 的文件名修改为 F28004x_EEPROM.c；修改 F28002x_EEPROM.h 为 F28004x_EEPROM.h(在工程中修改文件名可以使得引用这个文件地方的文件名全部更改)。
10) 找到 ECC 的使能寄存器地址(ECC Enable)，将其复制给 Meddle_Flash_Source.h 中的 ECC_Enable_REG_Addr(F28004X 中的 ECC 使能地址与 F28002X 一样，所以不用更改)。
11) 编译的时候选择 “CPU1_FLASH”模式。

3.3. 如何处理突然断电导致的 ECC 错误

在设备正在向 Flash 写数据的时候突然出现掉电，可能会出现 ECC 写入错误。当设备上电重启之后，程序运行到读取 EEPROM 数据时候，由于 ECC 之前写入错误，从而现在会出现 NMI 中断错误，若不对此错误进行处理，则会有风险导致产品需要重新烧写 Flash 程序。

为了解决这个问题，在设备启动过程中，在基本的外设初始化之后，增加代码手动关闭 ECC，然后读取之前写入的数据进行检查，如果检测通过，则说明 ECC 没出错。如果检查失败，数据出错则需要擦除扇区并将初始化值写入到 EEPROM。具体程序流程如图 13 所示。

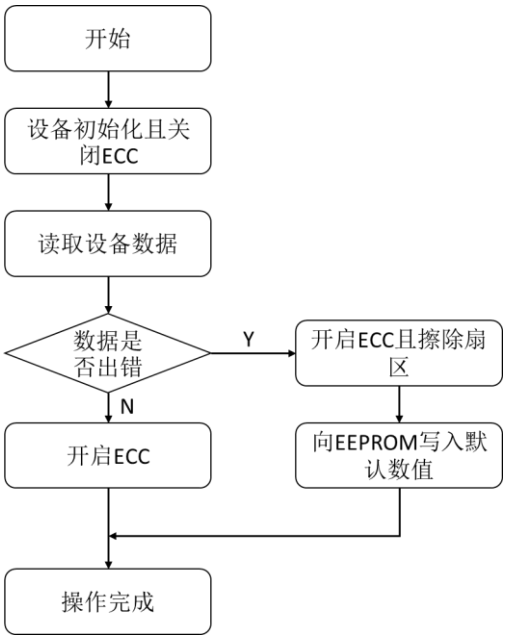


图 **13**. 停电启动检查图

4. 测试结果及总结

4.1. 用 Flash 模拟 EEPROM 与实际 EEPROM 的区别

表 2. EEPROM 与 Flash 的区别

类别	EEPROM	模拟 EEPROM
写入速度	5 到 10ms 写一个 Byte	写一个 Byte 不超过 50us
写的最小单位量	可以每次写一个 Byte	每次写 4 个 Word
写的地址对齐要求	无要求	需要 64-bit 对齐

4.2. 总结

本文使用 C2000 内部的 Flash 去模拟 EEPROM, 实现了数据的擦除、写入和读取，并且能够处理掉电的特殊情况。代码实现采用了分层的软件结构，能够快速移植到其他第三代 C2000 的产品上，方便用户快速移植此功能。

5. 参考文献

1. EEPROM Emulation for Gen 2 C2000 Real-Time MCUs
2. TMS320F28002x Real-Time Microcontrollers Technical Reference Manual (Rev. A)
3. TMS320F28004x Real-Time Microcontrollers Technical Reference Manual (Rev. D)

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