Application Report

第 2 代 C2000 实时 MCU 的 EEPROM 模拟

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），这样即使在系统断电后，也可以使用或修改并重复使用这些数据。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。因应用程序明显采用模拟电可擦除可编程只读存储器 (EEPROM)来实现数据的写入、读取和修改，故本应用报告及相关代码协助定义板载闪存的一个扇区。

本应用报告中讨论的项目资料和源代码可从以下网址下载：https//www.ti.com/lit/zip/sprab69.zip。

内容

插图清单

商标

1 简介

第 2 代 [1] C2000™ MCU 带有不同配置的闪存存储器，这些存储器被排列在多个扇区中。遗憾的是，片上闪存所使用的 CMOS 工艺技术不允许在芯片上添加传统的 EEPROM。一些设计人员使用外部 EEPROM 器件作为非易失性存储器。因此，闪存存储器是 EEPROM 的一种特殊类型。好消息是所有第 2 代 C2000 MCU 都具有对闪存存储器的在线编程能力。本应用报告利用该功能，通过在闪存存储器的限制内模拟 EEPROM 功能，将片上闪存的一个扇区用作 EEPROM。请注意，一个*闪存扇区*会完全作为模拟的 EEPROM；因此，将不适用于应用程序代码。

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。

与 EEPROM 相比，闪存具有更高的密度，允许在芯片上实现更大的存储器阵列（扇区）。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。任何字都可以被重写，将位从逻辑 1 更改为逻辑 0；但反过来则不行。第 2 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。

EEPROM 与闪存操作的主要区别在于写入和擦除时间。典型的闪存写入时间为每个 16 位字需要 50μs；而 EEPROM 通常需要 5 至 10ms。EEPROM 不需要页（扇区）擦除操作。用户可以擦除需要指定时间的特定字节。闪存擦除时间为每页几秒钟。对于第 2 代 C2000 MCU，擦除时间典型值为每 8K 扇区需要 10 秒。在写入/擦除操作期间，闪存电源必须保持稳定。

由于 EEPROM 与闪存具有不同的特性，因此通过闪存模拟 EEPROM 时会遇到一些挑战。

NOTE

有关闪存擦除计时”部分。

3 实现方案

通过闪存模拟 EEPROM 时，最大的挑战是确保闪存编程/擦除耐久性和数据保留能力满足可靠性目标。其次，在应用程序控制下，需要满足数据更新和读取的实时应用要求。请注意，在闪存擦除/编程期间，无法执行应用程序，因为这段时间内无法读取闪存。

NOTE

闪存擦除和写入周期会从 V_{DD} 和 V_{DDIO} 电源电压轨获得额外的电流。系统电源应设计为能够产生这一额外电流。有关额外电流的值，请参阅特定器件数据手册中的“闪存计时”部分。

NOTE

闪存耐久性额定为每 1000 个擦除/写入周期 100 分钟。特定器件数据手册内“电气特性”的“闪存计时序”部分中显示了这一数据。通过编程将数据逐字节写入扇区有助于延长闪存寿命。这是因为在擦除之前使用了整个扇区，而不是只部分使用扇区和擦除。

3.1 基本概念

对于闪存的块擦除要求，闪存扇区必须完整地保留下来用于模拟 EEPROM。这意味着，根据 C2000 器件型号，此类闪存扇区的大小可能为 4K x 16（F2801、F2802x、F2803x、F2806x）至 32K x 16（F28335/F28235、

[1] 第 2 代是指以下 C2000 器件：TMS320F281x、TMS320F280x、TMS320F2823x、TMS320F2833x、TMS320F2802x、TMS320F2803x、TMS320F2805x、TMS320F2806x 和 TMS320F28044。

F2809、F281x）不等。这一闪存扇区被分成几个更小的部分（以下简称“*页*”），每个页具有仿真型 EEPROM 的尺寸。例如，一个 4K x 16 大小的闪存扇区可以分成 32 个页，每个页的大小为 128 x 16。这使得每个页相当于一个 256 x 8 字节的 EEPROM。待*保存*的数据会首先写入 RAM 中的缓冲区。然后，借助第 2 代 C2000 MCU 的在线*编程*工具，将数据*写入*闪存所选扇区的第一页（页 1）。

在电源循环后，应用将*保存的*数据从页复制到 RAM 中的缓冲区。下次，应用则需要保存写入到下一页（页 2）的新数据。该过程会持续进行直到写入所选扇区的最后一页。

当到达最后一页时，必须擦除整个闪存扇区，并从第 1 页重新开始执行连续的 EEPROM 数据保存操作。

除上述描述的多字节编程概念外，还支持单字节编程。在此模式下，扇区不会被分成组和页。节 4.2.7 和节 4.2.8 将对单字节编程进行深入讨论。本应用报告的其余部分将讨论多字节编程模式。

3.2 创建 EEPROM 节（页）和页标识

根据闪存扇区的大小和模拟的 EEPROM ，可以创建大量的页。必须执行以下操作：

- 从上次保存期间写入的页中读回数据。
- 将最新数据写入新页。
- 知道*最后*一页，从页 1 开始擦除所有的页。
- 若应用需求时，从之前存储的任何数据读取。

为了简化对页的跟踪，首先将闪存扇区分成*组*。每个组又细分成*页*。这样仅有较少的组，而每个组内具有较少的页。这种分区如图 3-1 所示。

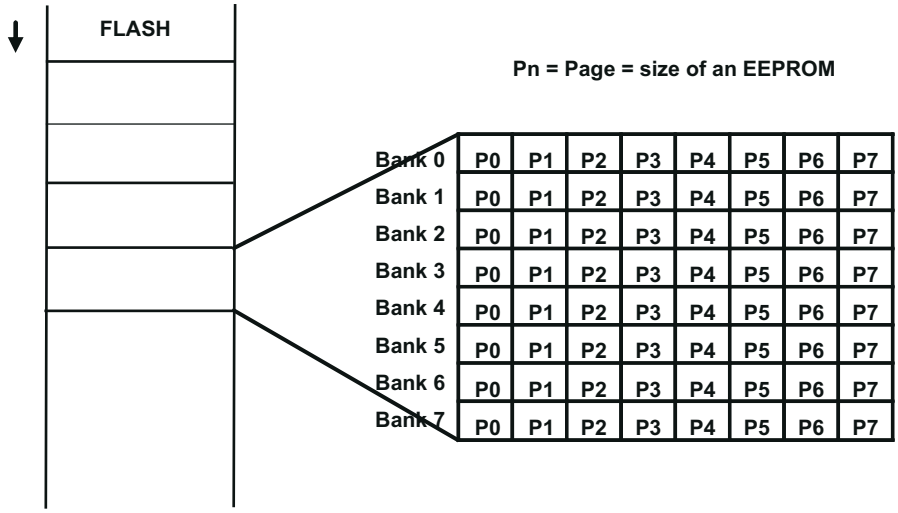

图 3-1. 组分区

每个组的首字保留为组状态信息，而每个页的首字保留为页状态信息。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。使用新组时，最后一组和当前组的组状态将被更新。

所有页均各自包含一个页状态和 64 个字节。页 0 略有不同，因为它还包含组状态（请参见图 3-2）。图中仅显示了页 0 和页 1。需要注意的是，页 2-7 与页 1 相同。

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

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

图 3-2. 页布局

4 软件说明

本应用报告随附的软件包含源代码，用于选择第 2 代 C2000 实时控制器，并用示例演示如何将源代码用于每一代所有的器件。

此软件提供了基本的 EEPROM 功能：写入、读取和擦除。闪存存储器的其中一个扇区用于模拟 EEPROM。如上所述，此扇区被分成多个组和页，每个组和页均包含状态位，用于确定数据的有效性。

软件包为自包含的，与 第 2 代 C2000 MCUs_EEPROM 一并提取 作为基本目录。此代码使用头文件 [3] 和闪存 API 库 [1], [2]，用于第 2 代 C2000 MCU（每一代）。

图 4-1 展示了第 2 代 C2000 MCUs_EEPROM 目录的目录结构。

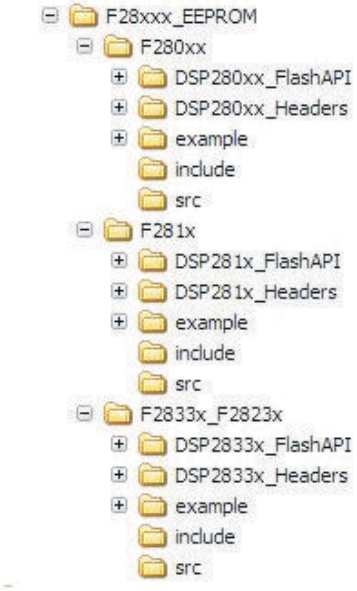

图 4-1. F28xxx_EEPROM 目录结构

如图所示，每一代 F28xxx 都有自己的文件夹，该文件夹包含子文件夹。提供了每一代的闪存 API 和头文件。示例文件夹中包含示例项目和源文件。include 和 src 文件夹中包含用于实现 EEPROM 模拟所需的头文件和源文件。此代码使用 F28xxx Code Generation Tools 5.2.0 版在 Code Composer Studio™ 3.3 版中进行了测试。

本文假定读者已查看*《在 TMS320F28xx DSP 的内部闪存上运行应用程序》*(SPRA958) [5]，并遵循闪存实现方法。

4.1 软件功能和流程

图 4-2 中显示了高层软件流程。器件必须首先通过初始化代码来初始化时钟、外设等。所用的初始化函数就是头文件软件包 [3] 中提供的函数。提供的示例遵循与头文件软件包 [3] 中的示例相同的初始化步骤。有关此序列的更多信息，请参阅头文件 [3] 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。库下载中包含了所需的完整步骤列表。此列表也可以在节 5 中找到，并附上闪存 API 的概述。F28xxx 每一代的库下载中都为该代的每个芯片提供了库文件。

此时，可以开始编程了。首先，需要捕捉数据才能进行编程。在对数据进行编程后，读取功能将读取编程写入闪存的最后一组数据。因为编程开始之前，需要将闪存 API 复制到内部 RAM，故大多数应用都应该遵循此软件流程，尤其是初始化部分。

所提供的示例项目遵循图 4-2 中所示的软件流程。

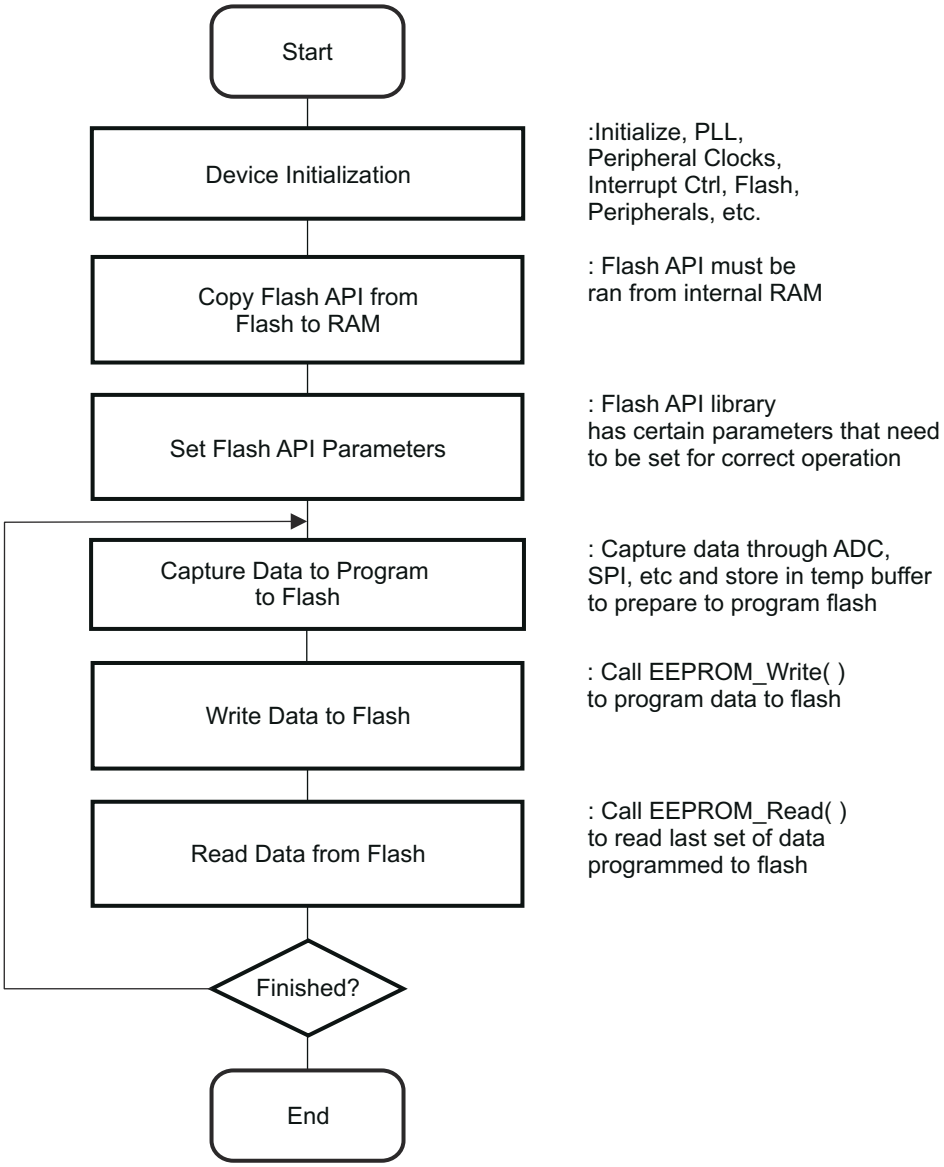

图 4-2. 软件流程

4.2 EEPROM 函数

为了实现该功能，需要在多字节中配置六个函数来执行编程、读取和擦除。单字节配置还需要另外两个函数。F28xxx 每一代生成的所有函数都包含在 EEPROM.C 文件中节 6 中列出了这些文件。

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

后续小节中会详细介绍上述每个函数。

4.2.1 EEPROM_Write

EEPROM_Write( ) 函数的功能是将数据编程写入闪存。此代码中有几个函数调用，以便为数据编程做好准备。具体的函数调用如下：

```
EEPROM_GetValidBank();        // 找到正在使用的组和当前页
EEPROM_UpdatePageStatus();    // 更新上一页的页状态
EEPROM_UpdateBankStatus();    // 更新当前组和上一组的组状态
```

后续部分会详细介绍上述每个函数。找到当前组和页后，更新上一页的页状态，如果有新组在使用，则更新组状态。接着，进行实际的编程。下面调用此过程：

```
// 在当前页中编程写入 Write_Buffer 中的数据
Length = 64;      // 将长度设置用于编程的页长度
Status = Flash_Program(Page_Pointer+1,Write_Buffer,Length,&ProgStatus);
```

必须先设置长度，然后调用 Flash_Program() 函数，以调用编程过程。需要将四个参数传递给 Flash_Program()：

- 闪存指针 (编程地址)
- 包含待写入数据的缓冲区
- 数据长度 (要编程写入的数据量)
- 由闪存 API 库定义的状态结构

最后，如果编程成功，则更新当前页的页状态。代码如下所示：

```
// 如闪存编程成功，则将页状态从空白页修改为当前页
if (Status == STATUS_SUCCESS)
{
Page_Status[0] = CURRENT_PAGE;    // 将页状态设置为当前页
  Length = 1;                     // 设置编程状态的长度
  Status = Flash_Program(Page_Pointer,Page_Status,Length,&ProgStatus);
}
```

4.2.2 EEPROM_Read

EEPROM_Read( ) 函数的功能是读取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。

找到当前的组和页，并将当前页的内容存储在 Read_Buffer 中：

```
EEPROM_GetValidBank();           // 找到正在使用的组和当前页
// 将当前页的内容传递到读取缓冲区
for(i=0;i<64;i++)
Read_Buffer[i] = *(++Page_Pointer);
```

4.2.3 EEPROM_Erase

EEPROM_Erase( ) 函数的功能是擦除存储所用的扇区。必须擦除整个扇区；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。正如所提供的，该函数仅当所有组和页已使用时才可调用。

闪存中的最后一个扇区用于 EEPROM 功能。以下代码显示了 F2833x 系列的配置：

```
#ifdef F28332         // 如果正在使用 F28332/F28232，则擦除扇区 D
Status = Flash_Erase((SECTORD),&FlashStatus);
#else                 // 否则擦除其他器件的扇区 H
Status = Flash_Erase((SECTORH),&FlashStatus);
#endif
```

正如所提供的，不包含错误检查功能，需要根据具体的应用进行添加。如果擦除失败，该算法目前会在断点处暂停。

```
if(Status != STATUS_SUCCESS)         // 如果擦除失败，则停止程序或处理错误
{
asm(" ESTOP0");
}
```

4.2.4 EEPROM_GetValidBank

EEPROM_GetValidBank( ) 函数的功能是查找当前组和页。该函数由 EEPROM_Write( ) 和 EEPROM_Read( ) 函数调用。图 4-3 展示了搜索当前组和页所需的总体流程。

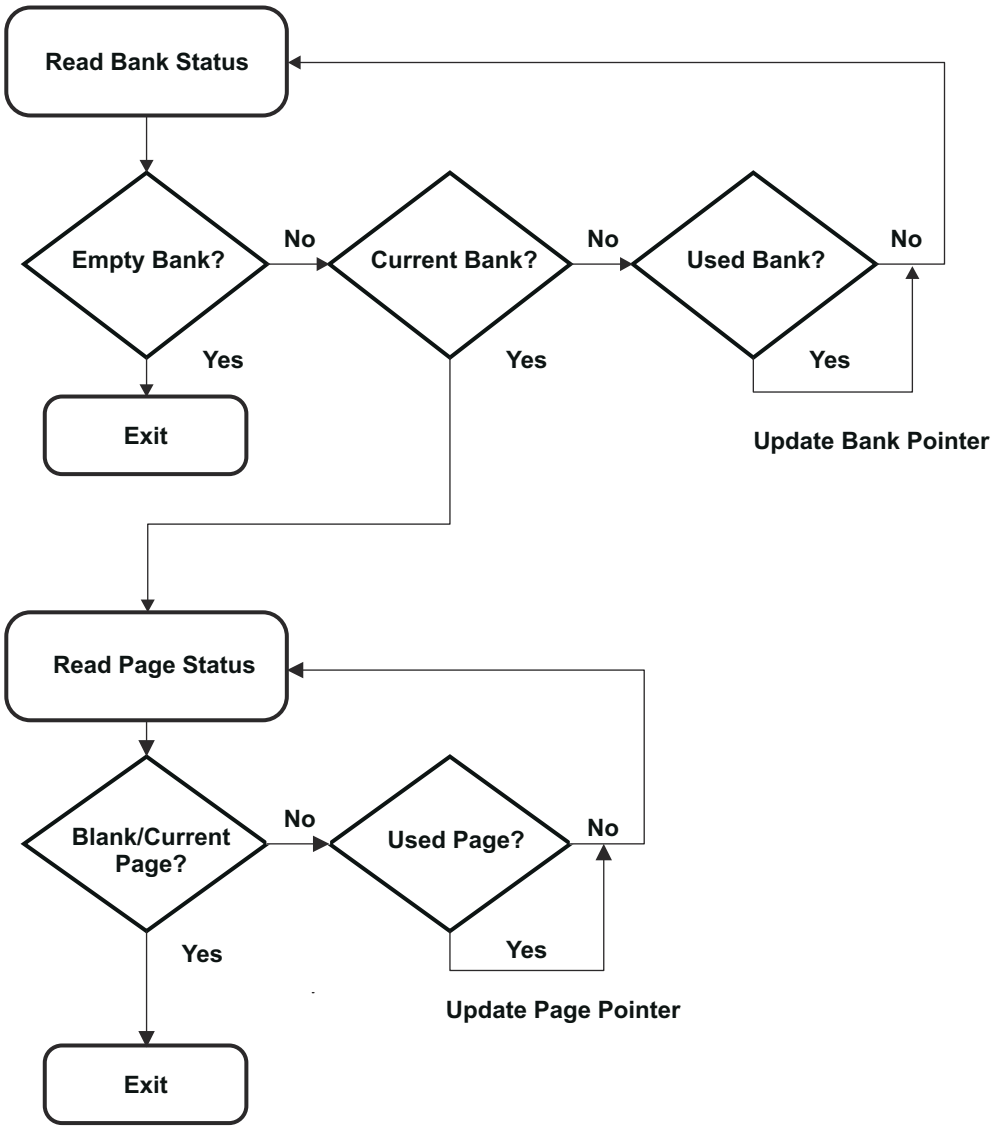

图 4-3. 当前组和页流程

进入此函数时，组指针和页指针会设置到指定扇区的开头：

```
RESET_BANK_POINTER;    // 重置组指针，以启用搜索当前组
RESET_PAGE_POINTER;    // 重置页指针，以启用搜索当前页
```

在所用的具体器件的 EEPROM.h 中定义了这些指针地址。

接着，找到当前组。如图 4-3 中所示，每个组可能有以下三种不同的状态：

- EMPTY_BANK
- USED_BANK
- CURRENT_BANK

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

```
if(Bank_Status[0] == EMPTY_BANK)  // 检查未使用的组
{
      Bank_Counter = i;           // 将组计数器设置为当前页的编号
      return;                     // 如果组未使用，返回 EEPROM 为空
}
```

如果未遇到 EMPTY_BANK，接着测试 CURRENT_BANK 状态。如果是当前组，更新组计数器，并将页指针设置到该组的第一页，以启用对当前页的测试。然后退出该循环，因为不需要进一步的组搜索。

```
if(Bank_Status[0] == CURRENT_BANK)  // 检查正在使用的组
{
        Bank_Counter = i;           // 将组计数器设置为当前组的编号
        // 将页指针设置为当前组的第一页
        Page_Pointer = Bank_Pointer + 1;
        break;                      // 找到当前组后退出循环
}
```

最后，测试 USED_BANK 状态。在这种情况下，该组已使用，组指针更新到下一个组，以测试其状态。

```
if(Bank_Status[0] == USED_BANK)       // 检查已使用的组
       Bank_Pointer += 521;         // 如果该组已使用，将指针设置为下一组
```

找到当前组后，需要找到当前页。页可能具有以下三种状态：

- BLANK_PAGE
- CURRENT_PAGE
- USED_PAGE

首先测试 BLANK_PAGE 和 CURRENT_PAGE 状态。如果页的当前状态为这两种状态之一，则表示找到了正确的页，并退出该循环，因为不需要进一步的搜索。

```
// 检查空白页或当前页
if(Page_Status[0] == BLANK_PAGE || Page_Status[0] == CURRENT_PAGE)
{
        Page_Counter = i;           // 将页计数器设置为当前页的编号
        break;                         // 找到当前页后，退出循环
}
```

如果页状态不是这两种状态中的任何一种，则唯一可用的其他状态只有 USED_PAGE。这种情况下，更新页指针到下一页，以测试其状态。

```
if(Page_Status[0] == USED_PAGE)      // 检查已使用的页
        Page_Pointer += 65;          // 如果该页已使用，则将指针设置为下一页
```

此时，当前组和页已找到，调用函数可以继续运行。最后，此函数将会检查是否使用了所有的组和页。这种情况下，需要擦除该扇区。通过测试组和页计数器来执行该检查。如上述代码片段中所示，当对当前组和页进行测试时，会设置这些计数器。

如果这两个计数器都是 7，则表示最后一组和最后一页是有效的，并且存储器已满。

```
if (Bank_Counter==7 && Page_Counter==7)      // 检查 EEPROM 是否已满
{
        EEPROM_Erase();             // 擦除被用作 EEPROM 的闪存扇区
        RESET_BANK_POINTER;         // 将组指针重置为 EEPROM 已空
        RESET_PAGE_POINTER;         // 将页指针重置为 EEPROM 已空
        asm(" ESTOP0");
}
```

如上所示，如果存储器已满，则调用 EEPROM_Erase( ) 函数，同时将组和页指针复位至第一组和第一页。

NOTE

EEPROM_Erase( ) 函数将会擦除整个闪存扇区。如果需要保存任何数据，则应在运行此函数之前进行保存。如果应用此时无需执行擦除操作，则可单独从 EEPROM_GetValidBank( ) 函数中移除 EEPROM_Erase( ) 调用，并单独调用 EEPROM_Erase( ) 函数。

正如所提供的，页和组搜索都会在 8 计数循环中内执行，因为 8 是此实现中所用的默认组数。这个值可以根据具体的应用进行更改。

4.2.5 EEPROM_UpdateBankStatus

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

```
Bank_Status[0] = *(Bank_Pointer);        // 从组指针读取组状态
```

如果此状态表示组为空，则状态会更改为 CURRENT_BANK 并进行编程：

```
// 编程组状态为空 EEPROM
if (Bank_Status[0] == EMPTY_BANK)
{
        Bank_Status[0] = CURRENT_BANK;    // 将组状态设置为正在使用组
        // 将组状态编程为当前组
        Status = Flash_Program(Bank_Pointer,Bank_Status,Length,&ProgStatus);
        Page_Counter =0;
        // 将页指针设置为当前组的第一页
        Page_Pointer = Bank_Pointer + 1;
}
```

如果状态不为空，接着检查组（已满组）中所有页的 CURRENT_BANK 状态。在这种情况下，当前组的状态将会更改为 USED_BANK，同时下一组的状态将更新为 CURRENT_BANK，以便允许对下一组进行编程。最后，页指针会更新到新组的第一页：

```
// 对已满组和以下组的组状态进行编程
if (Bank_Status[0] == CURRENT_BANK && Page_Counter == 7)
{
        Bank_Status[0] = USED_BANK;         // 将组状态设置为已使用组
        // 将组状态编程为已满组
        Status = Flash_Program(Bank_Pointer,Bank_Status,Length,&ProgStatus);
        Bank_Pointer +=521;                 // 将组指针递增至下一组
        Bank_Status[0] = CURRENT_BANK;      // 将组状态设置为正在使用组
        // 将组状态编程为当前组
       Status = Flash_Program(Bank_Pointer,Bank_Status,Length,&ProgStatus);
       Page_Counter =0;
        // 将页指针设置为当前组的第一页
       Page_Pointer = Bank_Pointer +1;
}
```

4.2.6 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。

```
Page_Status[0] = *(Page_Pointer);      // 从页指针读取页状态
```

如果此状态表示该页为空，则在 EEPROM_Write( ) 函数中更新此状态时退出该函数。否则，页状态会更新为 USED_PAGE，同时页指针会递增，以准备对下一页进行编程：

```
// 检查页状态是否为空。如果是，返回 EEPROM_WRITE。
if(Page_Status[0] == BLANK_PAGE)
        return;
// 将上一页的状态编程为已使用页
else
{
        Page_Status[0] = USED_PAGE;     // 将页状态设置为已使用页
        Length = 1;                     // 设置编程状态的长度
        Status = Flash_Program(Page_Pointer,Page_Status,Length,&ProgStatus);
        Page_Pointer +=65;              // 将页指针递增至下一页
}
```

4.2.7 EEPROM_GetSinglePointer

EEPROM_GetSinglePointer( ) 函数的功能是在单字节模式中查找第一个未用位置的指针。此函数还能用于确定是否使用了整个扇区。如果是，则使用 EEPROM_Erase( ) 函数擦除该扇区。首先，根据所使用的器件，设置该扇区的结束地址。在 F28xxx_EEPROM.h 文件中设置 END_OF_SECTOR 指令。

```
End_Address = (Uint16 *)END_OF_SECTOR;      // 设置扇区的 End_Address
```

在单字节模式中，必须找到第一个未用位置。在这种情况下，应用代码应该在编程之前调用该函数。将 1 传递给该函数，设置 First_Call，使代码能够搜索第一个未用位置。此循环只需运行一次，即可找到有效的指针。首次执行后，应该将 0 传递给该函数，从而绕过此循环。

```
if(First_Call == 1)     // 如果这是第一次调用函数，则查找有效的指针
{
        RESET_BANK_POINTER;              // 将组指针重置为扇区的开始
        while(*(Bank_Pointer) != 0xFFFF) // 测试数据的每个位置
        Bank_Pointer++;                  // 递增至下一位置
}
```

接着，将组指针与结束地址进行比较。如果组指针大于或等于结束地址，则表示该扇区已满。此时，会擦除该扇区，并将组指针设置回该扇区的开头。

```
if(Bank_Pointer >= End_Address) // 测试扇区是否已满
{
        EEPROM_Erase();         // 擦除被用作 EEPROM 的闪存扇区
        RESET_BANK_POINTER;     // 将作为 EEPROM 的组指针重置为空
        asm(" ESTOP0");
}
```

4.2.8 EEPROM_ProgramSingleByte

EEPROM_ProgramSingleByte( ) 函数的功能是将单字节编程到存储器中。单字节直接从该函数传递到名为 data 的变量。这个 data 被分配至 Write_Buffer，供 Flash_Program( ) 函数使用。一旦编程完成，调用 EEPROM_GetSinglePointer( )来确定该扇区是否已满，是否需要擦除该扇区。由于不需要确定 Bank_Pointer，将 0 传递给此函数。节 4.2.7 中对这一点进行了讨论。

```
Write_Buffer[0] = data;          // 准备要编程的数据
Length = 1;                      // 设置编程长度
Status = Flash_Program(Bank_Pointer++,Write_Buffer,Length,&ProgStatus);
EEPROM_GetSinglePointer(0);      // 全扇区测试
```

NOTE

在调用 EEPROM_GetSinglePointer() 函数来找到有效的指针位置之前，不能使用此函数。如果在此之前执行，则会产生未知结果。

4.3 测试示例

所提供的示例采用 F2812、F2808 和 F28335 eZdsp 开发板进行了测试。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。在使用 F28335 eZdsp 对该项目进行编程和测试时，遵循了以下步骤。这些步骤也可用于其他 eZdsp 开发板。

1. 使用板载 USB 连接将 F28335 eZdsp 连接至 PC；使用提供的电源连接器为电路板供电。
2. 使用 Code Composer Studio 安装工具中选择的 F28335 eZdsp 模拟驱动程序启动 Code Composer Studio。
3. 通过选择“Project”→“Open”，打开 F2833x_EEPROM_Example.pjt。
4. 选择项目配置中使用的器件，如图 4-4 中所示。

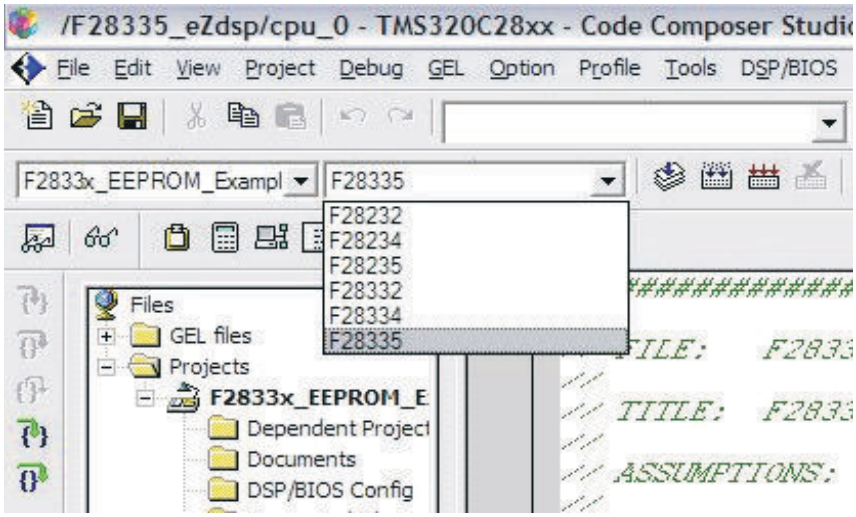

图 4-4. 器件选择

5. 通过选择“Project”→“Rebuild All”，构建项目。
6. 使用“Tools”菜单中的“Code Composer Studio On-Chip Flash Programmer”将生成的 .out 文件编程写入闪存。如果目前未安装 Code Composer Studio On-Chip Flash Programmer，可从 Update Advisor 下载。
7. 通过选择“File”→“Load Symbols”→“Load Symbols Only”，加载符号来调试程序。
8. 一旦项目被加载，通过选择“Debug”→“Go Main”，进入 main() 函数。

9. 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，如图 4-5 中所示。

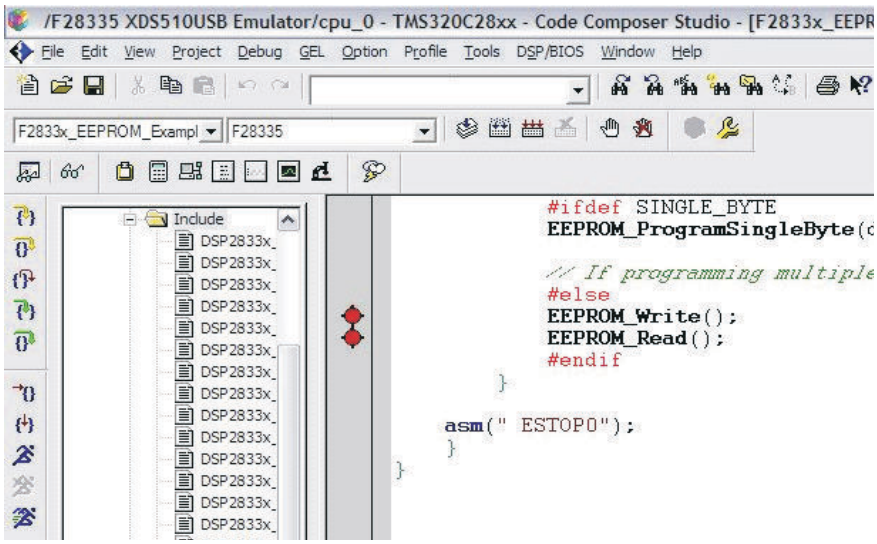

图 4-5. 断点

10. 运行至第一个断点，然后打开 Memory Window（“View”→“Memory”）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，如图 4-6 所示。

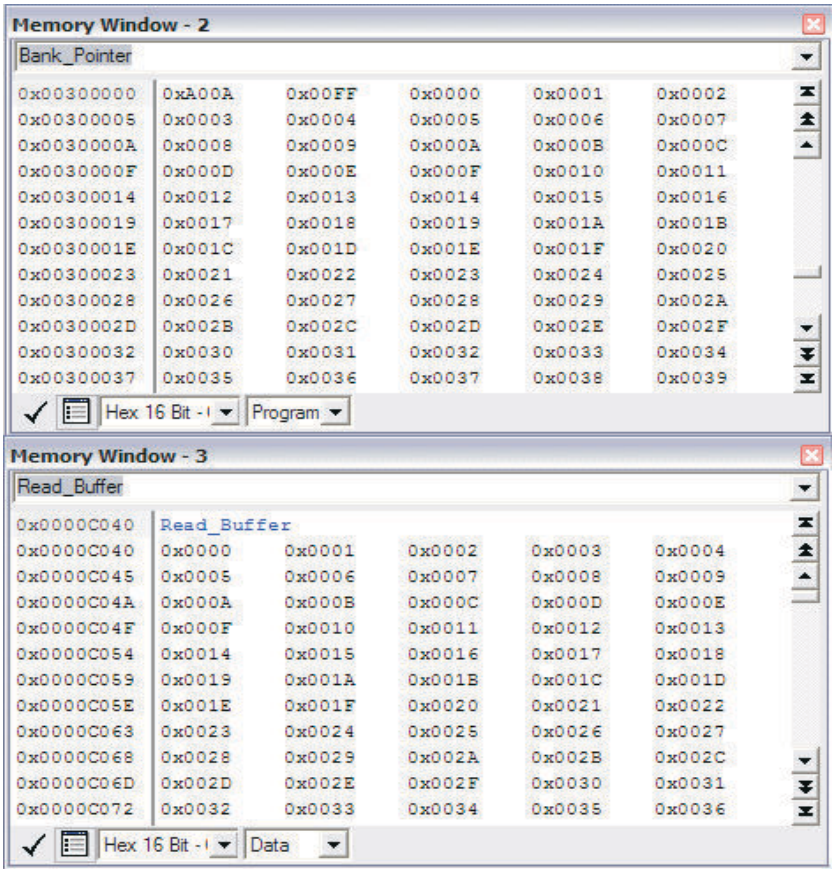

图 4-6. 存储器窗口

11. 继续从一个断点运行到另一个断点，观察每次写入的情况，直到程序运行完成。

上述步骤用于测试多字节配置。单字节配置也可以使用相同的步骤进行测试。唯一一个不需要的步骤是读取步骤，因为单字节模式中没有读取步骤。要启用单字节模式，通过取消 SINGLE_BYTE 指令的注释并注释 MULTI_BYTE 指令来更改 F28xxx_EEPROM.h 文件中的定义：

```
// 项目具体定义
#define SINGLE_BYTE 1
//#定义 MULTI_BYTE 1
```

4.4 应用集成

需要此功能的应用需要包含为每个 F28xxx 生成提供的 EEPROM.c 和 EEPROM.h 文件。另外，还需包含面向特定器件的闪存 API。例如，对于 TMS320F28335，需要包含以下文件：

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

所有闪存 API 文件都包含在下载的库中。有关闪存 API 的更多详细信息，请参阅节 5。一旦包含了上述文件，就可以遵循上述描述的软件流程和功能来确保正常工作。

NOTE

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。

NOTE

添加 EEPROM.C 和 EEPROM.h 文件会使整个代码量增加 356 个字的程序和 192 字的数据。

5 闪存 API

5.1 描述

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。闪存 API 擦除函数包括闪存预调节，无需单独的*清除*步骤。编程函数同时对闪存阵列和 OTP 块进行操作。编程函数只能将位从 1 改为 0。编程函数不能将位从 0 改回为 1。编程函数每次在一个 16 位字上运行。

本节简单介绍了闪存 API。有关闪存 API 库中的所有选项和操作流程的完整信息，请参阅 API 压缩文件内的 API 文档 [1], [2]。还有一份应用报告讨论了所有的更新选项和更新步骤 [4]。

5.2 闪存 API 检查清单

为完整起见，以下数据取自 API 文档作为参考。

将闪存 API 集成到用户软件中要求系统设计人员实现一些操作来满足几个关键要求。下列检查清单概述了使用 API 所需的步骤。这些步骤将会在指明的参考部分中进行详细讨论。这个检查清单适用于所有 F2823x 闪存 API，并取自 F2823x 闪存 API 文档。这个通用检查清单适用于所有 F28xxx 闪存 API 库，并且可在每个库的相应文档中找到 [1], [2]。

在使用 API 前，请完成以下操作：

1. 针对目标工作条件修改 Flash823x_API_Config.h。
2. 将 Flash2823x_API_Library.h 包括在源代码中。
3. 将合适的闪存 API 库添加到项目中。

   使用闪存 API 时，请使用大型存储器模型构建代码。API 库是在 28x Object 代码中构建的（OBJMODE=1，AMODE=1）。

在应用中，调用任何闪存 API 函数之前，请完成以下操作：

1. 初始化 PLL 控制寄存器 (PLLCR) 并等待 PLL 锁定。
2. 确保 PLL 没有在跛行模式下运行。如果 PLL 处于跛行模式，务调用任何 API 函数，因为器件将不能以适当的频率运行。
3. 可选：API 必须从零等待状态内部 SARAM 执行。如果要将 API 从闪存/OTP 复制到内部 SARAM 存储器中，那么请按照本节说明操作。
4. 初始化 32 位全局变量 Flash_CPUScaleFactor。
5. 初始化全局函数指针 Flash_CallbackPtr 以指向应用的回调函数。或者，将指针设定为 NULL。
6. 可选：在调用 API 函数前禁用全局中断。
7. 在进行任何 API 调用前，先了解本节中详细说明的 API 限制。
8. 可选：运行频率切换测试以确认闪存 API 的正确频率配置。注意：ToggleTest 函数将会一直执行。您必须暂停处理器来停止这个测试。
9. 可选：解锁代码安全模块 (CSM)。
10. 调用“API 参考”中描述的闪存 API 函数。

被调用的闪存 API 函数将进行以下操作：

- 禁用看门狗计时器。
- 检查 PARTID（内存位置 0x882）寄存器来确保此器件为正确的器件。
- 检查引导 ROM 中 0x3FFFB9 的内容，了解 API 版本与硅版本的兼容性。
  - 执行调用的操作，并且：
    - 在时间关键代码段禁用并恢复全局中断（通过 INTM、DBGM、XNMICR）。
    - 如果 Flash_CallbackPtr 不为 NULL，则调用回调函数。
  - 返回 success 或者错误代码。这些是在 F2823x_API_Library.h 中定义的。

然后，用户的代码应执行以下操作：

- 根据错误代码检查返回状态。
- 可选：重新启用看门狗计时器。

5.3 闪存 API 的注意事项

5.3.1 API 进行的操作

- 从零等待状态的内部 SARAM 存储器中执行闪存 API 代码。F2823x 和 F2833x 器件包含零等待状态 (L0-L3) 和 1 等待状态 SARAM (L4-L7)。闪存 APIs 应该从 L0-L3 SARAM 运行。
- 为正确的 CPU 工作频率配置 API。
- 按照节 5.2 中的“闪存 API 检查清单”将 API 集成到应用中。
- 在调用 API 函数之前，初始化 PLL 控制寄存器 (PLLCR) 并等待 PLL 锁定。
- 初始化 API 回调函数指针 (Flash_CallbackPtr)。如果不使用回调函数，那么最好将回调函数指针显式设定为 NULL。如果回调函数指针初始化失败，会导致代码跳转到未定义的位置。
- 仔细查看文档中描述的回调函数、中断和看门狗的 API 限制。

5.3.2 使用 API 时的禁止事项

- 禁止从闪存或者 OTP 执行闪存 API。如果 API 存储在闪存或者 OTP 内存中，则必须先将 API 复制到内部零等待状态 SARAM，再执行 API。
- 禁止在从闪存或 OTP 存储块中对 API 函数进行擦除、编程或删除恢复期间执行可能出现的任何中断服务程序 (ISR)。直到 API 函数运行完成并退出且闪存和 OTP 无法用于程序执行和数据存储。
- 禁止从闪存或者 OTP 执行 API 回调函数。当 API 在擦除、编程或删除恢复例程中调用回调函数时，闪存和 OTP 无法用于程序执行和数据存储。只有 API 函数运行完成并退出后，闪存和 OTP 才可用。
- 禁止停止正在执行的擦除、编程或者删除恢复函数（例如，禁止停止 API 代码内的调试器，禁止复位器件等）。
- 在闪存和/或 OTP 擦除、编程或者删除恢复期间，禁止执行代码或从闪存阵列或 OTP 提取数据。

6 源文件清单

表 6-1. 表 1.函数清单

文件	函数	描述
F280xx_EEPROM.c	EEPROM_Write()	执行写入操作
F281x_EEPROM.c	EEPROM_Read()	执行读取操作
F28335_EEPROM.c	EEPROM_Erase()	擦除闪存扇区
	EEPROM_GetValidBank()	查找有效的组和页
	EEPROM_UpdateBankStatus()	更新组状态
	EEPROM_UpdatePageStatus()	更新页状态
	EEPROM_GetSinglePointer(Uint16 First_Call)	为单字节操作找到有效指针，并测试全扇区
	EEPROM_ProgramSingleByte(Uint16 data)	将单字节编程写入扇区
F280xx_EEPROM.h		包含函数原型，包括
F281x_EEPROM.h		闪存 API 头、全局变量
F2833x_EEPROM.h		指针初始化、状态定义

7 结论

本应用报告证明了第 2 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。

8 参考文献

1. *TMS320F28x 定点闪存 API*：
   a. 《*TMS320F2801x 闪存 API*》(SPRC327)
   b. 《*TMS320F2804x 闪存 API*》(SPRC325)
   c. 《*TMS320F280x 闪存 API*》(SPRC193)
   d. 《*TMS320F2810、TMS320F2811 和 TMS320F2812 闪存 API*》(SPRC125)
2. *TMS320F28x 浮点闪存 API*：
   a. 《*TMS320F2833x 闪存 API*》(SPRC539)
3. TMS320C28x 头文件：
   a. 《*F2803x (Piccolo) C/C++ 头文件和外设示例*》(SPRC892)
   b. 《*C2834x (Delfino) C/C++ 头文件和外设示例*》(SPRC876)
   c. 《*F2802x (Piccolo) C/C++ 头文件和外设示例*》(SPRC832)
   d. 《*F2833x/F2823x C/C++ 头文件和外设示例*》(SPRC530)
   e. 《*C281x C/C++ 头文件和外设示例*》(SPRC097)
   f. 《*C280x C/C++ 头文件和外设示例*》(SPRC191)
   g. 《*C2804x C/C++ 头文件和外设示例*》(SPRC324)
4. 《*面向 TMS320F28xxx DSC 的闪存编程解决方案*》(SPRAAL3)
5. 《*在 TMS320F28xxx DSP 的内部闪存上运行应用程序*》(SPRA958)

9 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 * (September 2009) to Revision A (October 2020)	Page
• 更新了整个文档中的表格、图和交叉引用的编号格式	2
• 将“F28xxx”替换为“第 2 代 C2000 MCU”	2
• 删除了对术语“DSC”的所有引用，并替换为“实时控制器”	2
• 节 3.1 中进行了更新	2
• 节 4 中进行了更新	4

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