Reference Guide

TMS320F28003x 闪存 API 版本 1.58.11.00

TEXAS INSTRUMENTS

内容

插图清单

表格清单

商标

1 引言

本参考指南详细介绍了德州仪器 (TI) 的 TMS320F28003x 闪存 API 库（FAPI_F28003x_EABI_v1.58.xx.xx.lib 或 FAPI_F28003x_COFF_v1.58.xx.xx.lib）函数，可用于对 TMS320F28003x 器件进行擦除、编程和验证。请注意，闪存 API V1.58.xx.xx 应仅与 TMS320F28003x 器件一起使用。在 C2000Ware 中提供了闪存 API 库，位置为 C2000Ware_x_xx_xx_xx\libraries\flash_api\f28003x。

1.1 参考书籍

请将本指南与 *TMS320F28003x 微控制器数据手册* 和 *TMS320F28003x 微控制器技术参考手册* 结合使用。

1.2 函数清单格式

针对函数、编译器内在函数或宏的条目的通用格式。

对 **function_name()** 的作用的简短描述。

概要

为 **function_name()** 提供原型。

```
<return_type> function_name(
                  <type_1> parameter_1,
                  <type_2> parameter_2,
                  <type_n> parameter_n
                            )
```

参数

parameter_1 [in]	parameter_1 的类型详细信息
parameter_2 [out]	parameter_2 的类型详细信息
parameter_n [in/out]	parameter_3 的类型详细信息

参数传递分类如下：

- *in*：表示该函数使用提供的参数中的一个或多个值而不保存任何更改。
- *out*：表示该函数将一个或多个值保存在提供的参数中。您可以检查保存的值从而找出有关您的应用程序的有用信息。
- *in/out*：表示函数更改提供的参数中的一个或多个值并保存结果。您可以检查保存的值从而找出有关您的应用程序的有用信息。

说明

描述了该函数。本节还描述了可能适用的任何特殊特性或限制：

- 在某些情况下，该函数阻止或可能阻止所请求的操作
- 函数有预设条件，但可能不明显
- 函数有限制或特殊行为

限制

指定使用该函数时的任何限制。

返回值

指定该函数返回的任何一个或多个值。

另请参见

列出了与该函数相关的其他函数或数据类型。

实现示例

提供了演示函数用法的示例（或对示例的引用）。除了闪存 API 函数之外，此类示例还可以使用 C2000Ware 中提供的 device_support 文件夹或 driverlib 文件夹中的函数来演示在应用程序环境中如何使用给定的闪存 API 函数。

2 TMS320F28003x 闪存 API 概述

2.1 引言

闪存 API 是一个例程库，当以正确的顺序使用正确的参数调用时，可以对闪存进行擦除、编程或验证。闪存 API 也可用于对 OTP 存储器进行编程和验证。

备注

请阅读有关闪存和 OTP 存储器映射和闪存等待状态规范的数据手册。另请注意，本参考指南假定用户已阅读 *TMS320F28003x 微控制器技术参考手册* 中的*闪存和 OTP 存储器* 章节。

备注

- F280033/34 用户必须改用闪存 API 库 FAPI_F280033_34_EABI_v1.58.11.lib，而不是 FAPI_F28003x_EABI_v1.58.10.lib。版本 1.58.10 不支持 F280033/34 的 DCSM OTP 编程。
- 非 F280033/34 用户应继续使用 FAPI_F28003x_EABI_v1.58.10.lib。但是，FAPI_F28003x_EABI_v1.58.10.lib 的 Fapi_getLibraryInfo () 会返回修订版号 01，而不是 10。这是一个已知问题。

2.2 API 概述

表 2-1. 初始化函数汇总

API 函数	说明
Fapi_initializeAPI()	为供首次使用或更改频率，对 API 进行初始化

表 2-2. 闪存状态机 (FSM) 函数汇总

API 函数	说明
Fapi_setActiveFlashBank()	对闪存控制器 (FMC) 和闪存存储体进行初始化从而执行擦除或编程命令
Fapi_issueBankEraseCommand()	在应用扇区屏蔽后，针对给定的存储体地址向闪存状态机发出存储体擦除命令。
Fapi_issueAsyncCommandWithAddress()	针对给定地址向 FSM 发出擦除扇区命令
Fapi_issueProgrammingCommand()	设置编程所需的寄存器并向 FSM 发出命令
Fapi_issueProgrammingCommandForEccAddress()	将 ECC 地址重新映射到主数据空间，然后调用 Fapi_issueProgrammingCommand() 对 ECC 进行编程
Fapi_issueFsmSuspendCommand()	暂停 FSM 命令 Program Data 和 Erase Sector
Fapi_issueAsyncCommand()	向 FSM 发出命令（Clear Status、Program Resume、Erase Resume、Clear_More），用于进行无需地址的操作
Fapi_checkFsmForReady()	返回闪存状态机 (FSM) 是否处于就绪或繁忙状态
Fapi_getFsmStatus()	返回闪存控制器的 FMSTAT 状态寄存器值

表 2-3. 读取函数汇总

API 函数	说明
Fapi_doBlankCheck()	根据擦除状态验证指定的闪存范围
Fapi_doVerify()	根据提供的值验证指定的闪存范围

表 2-3. 读取函数汇总（续）

API 函数	说明
Fapi_calculatePsa()	计算指定闪存范围的 PSA 值
Fapi_doPsaVerify()	根据提供的 PSA 值验证指定的闪存范围

备注

为了提升闪存 API 读取函数中的闪存读取性能，在读取函数中启用了闪存包装程序（闪存模块控制器 (FMC)）的数据缓存。在启用数据缓存之前，读取函数会保存数据缓存以前的配置（由用户应用程序使用 DriverLib 中的闪存初始化例程设置）并在此类函数中的闪存读取完成后恢复相同的配置。

表 2-4. 信息函数汇总

API 函数	说明
Fapi_getLibraryInfo()	返回特定于 API 库编译版本的信息

表 2-5. 实用程序函数汇总

API 函数	说明
Fapi_flushPipeline()	刷新 FMC 中的数据缓存
Fapi_calculateEcc()	计算所提供地址和 64 位字的 ECC
Fapi_isAddressEcc()	确定地址是否在 ECC 范围内
Fapi_remapEccAddress()	将 ECC 地址重新映射到相应的主地址
Fapi_calculateFletcherChecksum()	函数计算指定内存范围的 Fletcher 校验和

请注意，TMS320F28003x 闪存 API 中删除了 Fapi_getDeviceInfo() 和 Fapi_getBankSectors()，因为用户可以从器件特定技术参考手册中提供的其他资源中获取该信息（例如，存储体数量、引脚数、扇区数等等)。

不再提供 Fapi_UserDefinedFunctions.c 文件，因为该文件中的函数现已合并到闪存 API 库中。有关在使用闪存 API 时维护看门狗功能的信息，请查看闪存 API 使用的关键事实。

2.3 使用 API

本节介绍各种 API 函数的使用流程。

2.3.1 初始化流程

2.3.1.1 器件上电后

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）。该过程对 API 内部结构进行初始化。

2.3.1.2 FMC 和存储体设置

在首次进行闪存操作之前，必须调用 *Fapi_setActiveFlashBank()* 函数。

2.3.1.3 关于系统频率变化

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）。该过程将更新 API 内部状态变量。

2.3.2 使用 API 进行构建

2.3.2.1 目标库文件

闪存 API 目标文件以 Arm® 标准 EABI elf 和 COFF 目标格式分发。

备注

编译需要启用“启用对 GCC 扩展的支持”选项。编译器版本 6.4.0 及更高版本默认启用该选项。

2.3.2.2 分布文件

以下 API 文件分布在 C2000Ware\libraries\flash_api\f28003x\ 文件夹中：

- 库文件
  - TMS320F28003x 闪存 API 未嵌入到该器件的引导 ROM 中，该 API 完全属于软件类型。提供的软件库采用 EABI elf (FlashAPI_F28003x_FPU32_EABI.lib) 和 COFF (FlashAPI_F28003x_FPU32_COFF.lib) 目标文件格式。应用程序中应包含这两种库文件的其中一个（具体取决于应用程序所使用的输出目标文件格式），以便能够擦除闪存/OTP 或对其进行编程。
    - FlashAPI_F28003x_FPU32_EABI.lib – 采用 EABI elf 目标文件格式的 TMS320F28003x 器件闪存 API 库。
    - FlashAPI_F28003x_FPU32_COFF.lib – 采用 COFF 目标文件格式的 TMS320F28003x 器件闪存 API 库。
    - 未提供 API 库的定点版本。
- 头文件
  - F021_F28003x_C28x.h – TMS320F28003x 器件的主头文件。该文件设置特定于编译的定义并包括 F021.h 主头文件。
- 以下头文件不应直接包含在用户代码中，但此处列出了此类文件以供用户参考：
  - F021.h – 该头文件列出了所有公共 API 函数并包括所有其他头文件。
  - Init.h – 定义 API 初始化结构。
  - Registers.h – 所有寄存器实现通用的定义，包括所选器件类型的相应寄存器头文件。
  - Types.h – 包含 API 使用的所有枚举和结构。
  - Constants/Constants.h – 某些 C2000™ 器件通用的常量定义。
  - Constants/F28003x.h – F28003x 器件的常量定义。

2.3.3 闪存 API 使用的关键事实

以下是有关 API 使用的一些重要事实：

- 闪存 API 函数的名称以前缀 "Fapi_" 开头。
- 闪存 API 不配置 PLL。用户应用程序应根据需要配置 PLL 并将配置的 CPUCLK 值传递给 Fapi_initializeAPI() 函数（该函数的详细信息在本文档后面给出）。
- 闪存 API 不会检查 PLL 配置来确认用户输入频率。这由系统集成商决定。TI 建议使用 DCC 模块来检查系统频率。有关实现示例，请参阅 C2000Ware driverlib 时钟配置函数。
- 在调用闪存 API 函数之前，请始终根据器件特定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 将发出错误。有关更多详细信息，请参阅 Fapi_Set ActiveFlashBank() 函数。
- 闪存 API 执行可中断。但是，不应从正在进行擦除/编程操作的闪存存储体进行任何读取/获取访问。因此，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中断服务例程 (ISR) 必须从 RAM 执行或从未作为擦除/编程操作目标的其他闪存存储体执行。例如，除了闪存 API 函数之外，上述条件还适用于下面显示的整个代码片段。

之所以会这样，是因为 Fapi_issueAsyncCommandWithAddress() 函数向 FSM 发出了擦除命令，但并没有等到擦除操作结束。只要 FSM 忙于进行当前操作，就不应访问正在擦除的闪存存储体。

```
//
// Erase a Sector
//
oReturnCheck = Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector,(uint32*)0x0080000);
//
// Wait until the erase operation is over
//
while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}
```

- 闪存 API 不配置（启用/禁用）看门狗。用户应用程序可以配置看门狗并根据需要对其进行维护。因此，不再提供 Fapi_ServiceWatchdogTimer() 函数。
- 闪存 API 根据需要在内部使用 EALLOW 和 EDIS 来允许/禁止对受保护的寄存器进行写入。
- 主阵列闪存编程必须与 64 位地址边界对齐，并且每个 64 位字在每个写/擦除周期只能编程一次。
- 允许单独对数据和 ECC 进行编程。但是，每个 64 位数据字和相应的 ECC 字在每个写入/擦除周期只能编程一次。
- DCSM OTP 编程必须与 128 位地址边界对齐，并且每个 128 位字只能编程一次。例外包括：
  - DCSM OTP 中的 DCSM Zx-LINKPOINTER1 和 Zx-LINKPOINTER2 值应一起编程，并且可以按照 DCSM 操作的要求一次编程 1 位。
  - DCSM OTP 中的 DCSM Zx-LINKPOINTER3 值可按照 DCSM 操作的要求，一次编程 1 位。
  - 如果需要，用户可以单独对 JLM_Enable 进行编程。如果用户要同时对所有 64 位进行编程，则必须读取链路指针（ZxOTP_LINKPOINTER1、ZxOTP_LINKPOINTER2、ZxOTP_LINKPOINTER3），并将该值与 JLM_Enable 值一起编程。
- TMS320F28003x 器件中没有泵信标。
- 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。即使用户将 Fapi_AutoEccGeneration 或 Fapi_DataAndEcc 模式作为编程模式参数传递，API 也会使用 Fapi_DataOnly 模式对此类位置进行编程。Fapi_EccOnly 模式不支持对此类位置进行编程。用户应用程序应谨慎处理这一点。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。如果其他字段与链接指针混合，API 也会跳过对非链接指针位置的 ECC 编程。这将导致应用程序中出现 ECC 错误。
- 当使用 INTOSC 作为时钟源时，一些 SYSCLK 频率范围需要额外的等待状态来进行擦除和编程操作。操作结束后，就不需要额外的等待状态了。有关更多详细信息，请参阅 *TMS320F28003x 微控制器数据手册*。
- 为了避免 zone1 和 zone2 之间的冲突，DCSM 寄存器中提供了信标 (FLSEM) 来配置闪存寄存器。在对闪存进行初始化和调用闪存 API 函数之前，用户应用程序应该配置该信标寄存器。有关该寄存器的更多详细信息，请参阅 *TMS320F28003x 微控制器技术参考手册*。
- 请注意，闪存 API 函数不配置任何 DCSM 寄存器。用户应用程序应确保配置所需的 DCSM 设置。例如，如果某个区域受到保护，则应从同一区域执行闪存 API，以便能够擦除或编程该区域的闪存扇区。或者应解锁该区域。否则，闪存 API 对闪存寄存器的写入将不会成功。闪存 API 不会检查对闪存寄存器的写入是否正在进行。其按照擦除/编程序列的要求向它们写入数据，并在假设写入完成后就返回。这将导致闪存 API 返回错误的成功状态。例如，调用 Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, Address) 时，可以返回成功状态，但这并不意味着扇区擦除成功。应使用 Fapi_getFSMStatus() 和 Fapi_doBlankCheck() 来检查擦除状态。
- 请注意，不应对正在进行闪存擦除/编程操作的闪存存储体/OTP 进行任何访问。

3 API 函数

3.1 初始化函数

3.1.1 Fapi_initializeAPI()

对闪存 API 进行初始化

概要

```
Fapi_StatusType Fapi_initializeAPI(
        Fapi_FmcRegistersType *poFlashControlRegister,
        uint32 u32HclkFrequency)
```

参数

poFlashControlRegister [in]	指向闪存控制器寄存器基地址的指针。使用 F021_CPU0_BASE_ADDRESS。
u32HclkFrequency [in]	以 MHz 为单位的系统时钟频率

说明

在执行任何其他闪存 API 操作之前，需要使用该函数来对闪存 API 进行初始化。如果更改系统频率或 RWAIT，也必须调用该函数。

备注

调用该函数前必须设置 RWAIT 寄存器值。

对闪存控制寄存器基地址在该函数内部进行了硬编码，不使用用户提供的值 (传递给该函数的第一个参数) 。但是，如果内部硬编码值与用户提供的值不匹配，即使仍然正常执行初始化步骤，也会向用户返回警告。

返回值

- **Fapi_Status_Success** (成功)
- **Fapi_Warning_BaseRegCntlAddressMismatch** (警告)

实现示例

```
#include “F021_F28003x_C28x.h”
#define CPUCLK_FREQUENCY   120   /* 120 MHz System frequency */
int main(void)
{
     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
     //
     Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);

     //
     // Jump to RAM and call the Flash API functions
     //
     Example_CallFlashAPI();
}
#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
     Fapi_StatusType oReturnCheck;

     //
     // This function is required to initialize the Flash API based on
     // System frequency before any other Flash API operation can be performed
     // Note that the FMC register base address and system frequency are passed as the parameters
     //
     // This function must also be called whenever System frequency or RWAIT is changed.
```

```
    //
    oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, CPUCLK_FREQUENCY);
    if(oReturnCheck != Fapi_Status_Success)
    {
         Example_Error(oReturnCheck);
    }

    //
    // Fapi_setActiveFlashBank function initializes Flash bank
    // and FMC for erase and program operations.
    //
    oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);

    if(oReturnCheck != Fapi_Status_Success)
    {
         Example_Error(oReturnCheck);
    }
    //
    // Erase Program
    //
    /* User code for further Bank flash operations */
    .
    .
    .
    .

    //
    // Example is done here
    //

    Example_Done();

}
```

3.2 闪存状态机函数

3.2.1 Fapi_setActiveFlashBank()

对 FMC 进行初始化从而进行擦除和编程操作。

概要

```
Fapi_StatusType Fapi_setActiveFlashBank(
          Fapi_FlashBankType oNewFlashBank)
```

参数

oNewFlashBank [in]	设置为有效的存储体编号。

说明

该函数用于设置闪存控制器，以便在存储体上进行进一步的操作。在使用 *Fapi_initializeAPI()* 函数之后以及在执行任何其他闪存 API 操作之前需要调用该函数。

备注

闪存存储体编号在该函数内部进行了硬编码，不使用用户提供的值。

即使该器件上最多有 3 个存储体，也无需在切换存储体时调用该函数。应用程序只需要调用该函数一次，其可以与 Fapi_FlashBank0 一起使用。

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Status_FsmBusy**（失败：FSM 正忙于执行另一个命令）
- **Fapi_Error_InvalidBaseRegCntlAddress**（失败：用户提供的闪存控制寄存器基地址与预期地址不匹配）
- **Fapi_Error_InvalidBank**（失败：器件上不存在指定的存储体）
- **Fapi_Error_InvalidHclkValue**（失败：系统时钟与指定的等待值不匹配）
- **Fapi_Error_OtpChecksumMismatch**（失败：计算出的 TI OTP 校验和与 TI OTP 中的值不匹配）

实现示例

请参阅 节 3.1.1 中的示例。

3.2.2 Fapi_issueAsyncCommandWithAddress()

向闪存状态机发出擦除命令以及用户提供的扇区地址。

概要

```
Fapi_StatusType Fapi_issueAsyncCommandWithAddress(
        Fapi_FlashStateCommandsType oCommand,
        uint32 *pu32StartAddress)
```

参数

oCommand [in]	向 FSM 发出的命令。使用 Fapi_EraseSector
pu32StartAddress [in]	用于擦除操作的闪存扇区地址

说明

该函数针对用户提供的扇区地址向闪存状态机发出擦除命令。该函数不会等到擦除操作结束；它只是发出命令并返回。因此，当使用 Fapi_EraseSector 命令时，该函数始终返回成功状态。用户应用程序必须等待 FMC 完成擦除操作，然后才能返回到任何类型的闪存访问。Fapi_checkFsmForReady() 函数可用于监测已发出命令的状态。

此外，如果应用程序同时进行存储体擦除和扇区擦除操作，则应用程序必须在调用该函数进行扇区擦除操作之前向 FSM 发出 Fapi_ClearMore 命令（使用 Fapi_issueAsyncCommand）。在执行存储体擦除命令之后，需要执行 Fapi_ClearMore 命令将 FSM 初始化为干净状态从而进行扇区擦除操作。如果应用程序中仅使用其中一个擦除操作（扇区擦除或存储体擦除），则无需在扇区擦除操作之前发出 Fapi_ClearMore 命令。

备注

该函数在发出擦除命令后不检查 FMSTAT。当 FSM 完成擦除操作时，用户应用程序必须检查 FMSTAT 值。FMSTAT 指示擦除操作期间是否有任何故障发生。用户应用程序可以使用 Fapi_getFsmStatus 函数来获取 FMSTAT 值。

此外，用户应用程序应使用 Fapi_doBlankCheck() 函数来验证闪存是否被擦除。

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Error_InvalidBaseRegCntlAddress**（失败：用户提供的闪存控制寄存器基地址与预期地址不匹配）
- **Fapi_Error_FeatureNotAvailable**（失败：用户请求的命令不受支持。）
- **Fapi_Error_FlashRegsNotWritable**（失败：闪存寄存器写入失败。用户应确保 API 从与闪存操作的目标地址相同的区域执行，或者用户应在闪存操作之前解锁。）
- **Fapi_Error_InvalidAddress**（失败：用户提供的地址无效。有关有效地址范围的信息，请参阅 *TMS320F28003x 微控制器数据手册*。）

实现示例

```
#include “F021_F28003x_C28x.h”
#define CPUCLK_FREQUENCY   120   /* 120 MHz System frequency */
int main(void)
{
     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
     //
     Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);
```

```
    //
    // Jump to RAM and call the Flash API functions
    //
    Example_CallFlashAPI();
}
#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
    Fapi_StatusType oReturnCheck;
    Fapi_FlashStatusType oFlashStatus;
    //
    // This function is required to initialize the Flash API based on
    // System frequency before any other Flash API operation can be performed
    // Note that the FMC register base address and system frequency are passed as the parameters
    //
    oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, CPUCLK_FREQUENCY);
    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error(oReturnCheck);
    }

    //
    // Fapi_setActiveFlashBank function initializes Flash banks
    // and FMC for erase and program operations.
    //
    oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error(oReturnCheck);
    }

    //
    // Code for bank erase (not shown here)
    //

    //
    // Code for Bank 0 sector 4 program (not shown here)
    //

    //
    // Issue ClearMore command - Required prior to Sector Erase
    //
    oReturnCheck = Fapi_issueAsyncCommand(Fapi_ClearMore);

    //
    // Wait until FSM is done with clear more operation
    //
    while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

    if(oReturnCheck != Fapi_Status_Success)
    {
        //
        // Check Flash API documentation for possible errors
        //
        Example_Error(oReturnCheck);
    }

    //
    // Bank0 Flash operations
    //
    //
    // Erase Bank0 Sector4
    //
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, (uint32 *)0x84000);
    //
    // Wait until FSM is done with erase sector operation
    //
    while(Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}
    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error (oReturnCheck);
    }
    //
    // Read FMSTAT contents to know the status of FSM
    // after erase command to see if there are any erase operation
    // related errors
    //
    oFlashStatus = Fapi_getFsmStatus();
```

```
    if (oFlashStatus!=0)
    {
         FMSTAT_Fail();
    }
    //
    // Do blank check.
    // Verify that the sector is erased.
    //
    oReturnCheck = Fapi_doBlankCheck((uint32 *)0x84000,
0x800,&oFlashStatusWord);
    if(oReturnCheck != Fapi_Status_Success)
    {
         Example_Error(oReturnCheck);
    }
    //
    // * User code for further Bank0 flash operations *
    //
    .
    .
    .
    //
    // Example is done here
    //
    Example_Done();
}
```

3.2.3 Fapi_issueBankEraseCommand()

向闪存状态机发出存储体擦除命令以及用户提供的扇区掩码。

概要

```
Fapi_StatusType Fapi_issueBankEraseCommand(
        uint32 *pu32StartAddress,
        uint16 oSectorMask)
```

参数

pu32StartAddress [in]	用于进行存储体擦除操作的闪存存储体地址
OSectorMask [in]	16 位掩码，指示存储体擦除操作中要屏蔽的扇区

说明

该函数针对用户提供的存储体地址，向闪存状态机发出存储体擦除命令。如果 FSM 正忙于进行另一个操作，该函数将返回，指示 FSM 处于繁忙状态，否则将继续进行存储体擦除操作。用户提供的 16 位扇区掩码指示存储体擦除操作中用户想要屏蔽的扇区，即不会被擦除的扇区。每个位表示一个扇区，位 0 表示扇区 0，位 1 表示扇区 1，依此类推，直到位 15 代表扇区 15。如果掩码中的某个位为 1，则不会擦除该特定扇区。

无法暂停存储体擦除操作。如果用户应用程序在存储体擦除操作有效期间发出暂停命令（使用 Fapi_issueFsmSuspendCommand()），暂停函数将返回错误。

备注

为存储体擦除命令提供正确的扇区掩码非常重要。如果错误地选择掩码来擦除无法访问的扇区（属于其他安全区域），存储体擦除命令将继续尝试不停地擦除扇区并且 FSM 永远不会退出（因为擦除不会成功）。为避免这种情况，用户必须注意提供正确的掩码。但是，鉴于有可能选择不正确的掩码，TI 建议在 FSM 为进行存储体擦除操作发出最大脉冲数后，将允许的最大擦除脉冲数初始化为零。这将确保 FSM 在尝试擦除不可访问的扇区到允许的最大擦除脉冲后结束存储体擦除命令。

C2000Ware 的闪存 API 使用示例中的 Example_EraseBanks() 函数描述了该序列的实现，如下面的*实现示例*部分（等待 FSM 完成命令的 while 循环的内容）所示。无论应用程序是否使用安全性，用户都必须按原样使用该代码。在擦除失败的情况下，FSM 需要退出存储体擦除操作。

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Status_FsmBusy**（FSM 处于繁忙状态）
- **Fapi_Error_FlashRegsNotWritable**（闪存寄存器不可写）
- **Fapi_Error_InvalidBaseRegCntlAddress**（失败：用户提供的闪存控制寄存器基地址与预期地址不匹配）

实现示例

有关更多详细信息，请参阅 C2000Ware 中闪存 API 使用示例中的 Example_EraseBanks()，其位于 C:\ti\c2000\C2000Ware_x_xx_xx_xx\driverlib\f28003x\examples\flash\flashapi_ex1_programming.c。下面显示了示例的一部分，说明如何在发出最大脉冲后将擦除脉冲初始化为零。

```
u32CurrentAddress = Bzero_Sector8_start;
oReturnCheck = Fapi_issueBankEraseCommand((uint32 *)u32CurrentAddress,     0x001F);

// Wait until FSM is done with bank erase operation
while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady)
{
     //
     // Initialize the Erase Pulses to zero after issuing max pulses
     //
     if(HWREG(FLASH0CTRL_BASE + FLASH_O_ACC_EP) > MAX_ERASE_PULSE)
     {
         EALLOW;

         //
         // Enable Flash Optimization
         //
         HWREG(FLASH0CTRL_BASE + FLASH_O_OPT) = OPT_ENABLE;

         HWREG(FLASH0CTRL_BASE + FLASH_O_ERA_PUL) =
                 HWREG(FLASH0CTRL_BASE + FLASH_O_ERA_PUL) &
                 ~(uint32_t)FLASH_ERA_PUL_MAX_ERA_PUL_M;

         //
         // Disable Flash Optimization
         //
         HWREG(FLASH0CTRL_BASE + FLASH_O_OPT) = OPT_DISABLE;

         EDIS;
     }
}
```

3.2.4 Fapi_issueProgrammingCommand()

设置数据并向有效的闪存或 OTP 存储器地址发出编程命令

概要

```
Fapi_StatusType Fapi_issueProgrammingCommand(
        uint32 *pu32StartAddress,
        uint16 *pu16DataBuffer,
        uint16  u16DataBufferSizeInWords,
        uint16 *pu16EccBuffer,
        uint16  u16EccBufferSizeInBytes,
        Fapi_FlashProgrammingCommandType oMode)
```

参数

pu32StartAddress [in]	闪存中的起始地址，用于对数据和 ECC 进行编程	
pu16DataBuffer [in]	指向数据缓冲区地址的指针。数据缓冲区应为 128 位对齐。	
u16DataBufferSizeInWords [in]	数据缓冲区中 16 位字的数量	
pu16EccBuffer [in]	指向 ECC 缓冲区地址的指针	
u16EccBufferSizeInBytes [in]	ECC 缓冲区中的 8 位字节数	
oMode [in]	表示要使用的编程模式：	
	Fapi_DataOnly	仅对数据缓冲区进行编程
	Fapi_AutoEccGeneration	对数据缓冲区进行编程，并自动生成 ECC 并对其进行编程。
	Fapi_DataAndEcc	对数据和 ECC 缓冲区进行编程
	Fapi_EccOnly	仅对 ECC 缓冲区进行编程

备注

pu16EccBuffer 应包含与 128 位对齐主阵列/OTP 地址处的数据对应的 ECC。pu16EccBuffer 的 LSB 与主阵列的低 64 位相对应，pu16EccBuffer 的 MSB 与主阵列的高 64 位相对应。

说明

该函数根据提供的参数设置闪存状态机的编程寄存器，为用户提供了适用于不同场景的四种不同编程模式，如表 3-1 中所述。

表 3-1. 不同编程模式的使用

编程模式 (oMode)	使用的参数	用途
Fapi_DataOnly	pu32StartAddress、pu16DataBuffer、u16DataBufferSizeInWords	当任何自定义编程实用程序或用户应用（嵌入/使用闪存 API）必须单独对数据和相应 ECC 进行编程时使用。使用 Fapi_DataOnly 模式对数据进行编程，然后使用 Fapi_EccOnly 模式对 ECC 进行编程。通常大多数编程实用程序不会单独计算 ECC，而是使用 Fapi_AutoEccGeneration 模式。但是，某些安全应用程序可能需要在其闪存映像中插入有意的 ECC 错误（使用 Fapi_AutoEccGeneration 模式时无法实现），从而在运行时检查单错校正和双错检测 (SECDED) 模块的运行状况。在这种情况下，ECC 是单独计算的（使用附录 F 中提供的 ECC 计算算法或使用 Fapi_calculateEcc() 函数（如适用））。应用程序可能希望根据需要在主阵列数据或 ECC 中插入错误。在这种情况下，在错误插入之后，可以使用 Fapi_DataOnly 模式和 Fapi_EccOnly 模式分别对数据和 ECC 进行编程。
Fapi_AutoEccGeneration	pu32StartAddress、pu16DataBuffer、u16DataBufferSizeInWords	当任何自定义编程实用程序或用户应用（其在运行时嵌入/使用闪存 API 对闪存进行编程从而存储数据或进行固件更新）必须同时对数据和 ECC 进行编程而不插入任何有意错误时使用。该模式是常用的模式。
Fapi_DataAndEcc	pu32StartAddress、pu16DataBuffer、u16DataBufferSizeInWords、pu16EccBuffer、u16EccBufferSizeInBytes	该模式的用途与同时使用 Fapi_DataOnly 和 Fapi_EccOnly 模式的用途没有什么不同。但是，当可以同时对数据和计算出的 ECC 进行编程时，该模式会很有用。

表 3-1. 不同编程模式的使用（续）

编程模式 (oMode)	使用的参数	用途
Fapi_EccOnly	pu16EccBuffer、u16EccBufferSizeInBytes	请参阅 Fapi_DataOnly 模式的用途描述。

备注

由于 ECC 检查在上电时启用，用户必须始终为其闪存映像对 ECC 进行编程。

编程模式：

Fapi_DataOnly – 该模式将只对闪存中指定地址的数据部分进行编程。其编程范围从 1 位至 8 个 16 位字。但是，请注意该函数的限制条件中对闪存编程数据大小的限制。提供的编程起始地址加上数据缓冲区长度无法跨越 128 位对齐地址边界。使用该模式时将忽略参数 4 和 5。

Fapi_AutoEccGeneration – 该模式将对闪存中提供的数据以及自动生成的 ECC 进行编程。针对在 64 位存储器边界上对齐的每项 64 位数据计算 ECC。因此，在使用该模式时，应针对给定的 64 位对齐存储器地址，同时对所有 64 位数据进行编程。未提供的数据全部视作 1 (0xFFFF)。针对 64 位数据计算 ECC 并对其进行编程后，即使在该 64 位数据中将位从 1 编程为 0，也无法对此类 64 位数据进行重新编程（除非扇区被擦除），因为新的 ECC 值将与先前编程的 ECC 值相冲突。使用该模式时，如果起始地址进行了 128 位对齐，则可以根据需要同时对 8 个或 4 个 16 位字进行编程。如果起始地址进行了 64 位对齐而不是 128 位对齐，则只能同时对 4 个 16 位字进行编程。该选项还存在 Fapi_DataOnly 的数据限制。忽略参数 4 和 5

备注

Fapi_AutoEccGeneration 模式将对闪存中提供的数据部分以及自动生成的 ECC 进行编程。针对 64 位对齐地址和相应的 64 位数据计算 ECC。未提供的任何数据将视作 0xFFFF。请注意，在编写自定义编程实用程序时，该实用程序在代码项目的输出文件中流式传输，并将各段一次编程到闪存中，这会产生实际影响。如果一个 64 位的字跨越多个段（即包含一段的末尾和另一段的开头），在对第一段进行编程时，无法针对 64 位字中缺失数据假设值为 0xFFFF。当您对第二段进行编程时，您将无法对第一个 64 位字的 ECC 进行编程，因为它已经（错误地）使用假定的 0xFFFF 对缺失值进行了计算和编程。避免该问题的一种方法是在代码项目的链接器命令文件中的 64 位边界上对齐链接到闪存的所有段。

下面我们举例说明：

```
SECTIONS
   {
     .text        : > FLASH, ALIGN(4)
     .cinit       : > FLASH, ALIGN(4)
     .const       : > FLASH, ALIGN(4)
     .init_array  : > FLASH, ALIGN(4)
     .switch      : > FLASH, ALIGN(4)
   }
```

如果不在闪存中对齐这些段，则需要跟踪段中不完整的 64 位字，并将这些字与其他段中的字组合在一起，从而使 64 位字变得完整。这不容易做到。因此，建议将段在 64 位边界上对齐。

某些第三方闪存编程工具或 TI 闪存编程内核示例 (*C2000Ware*) 或任何自定义闪存编程解决方案可能假定传入数据流全部为 128 位对齐，并且可能没有预想到某段的起始地址未对齐。因此，假设提供的地址为 128 位对齐，则可能会尝试对最大可能的（128 位）字进行一次编程。当地址未对齐时，这可能会导致出现故障。因此，建议使用 ALIGN(8) 将所有段（已映射到闪存）在 128 位边界上对齐。

Fapi_DataAndEcc – 该模式将在指定地址的闪存中对提供的数据和 ECC 进行编程。提供的数据必须在 64 位存储器边界上对齐，并且数据长度必须与提供的 ECC 相关联。这意味着，如果数据缓冲区长度为 4 个 16 位字，则 ECC 缓冲区长度必须为 1 字节。如果数据缓冲区长度为 8 个 16 位字，则 ECC 缓冲区的长度必须为 2 字节。如果起始地址进行了 128 位对齐，则应根据需要同时对 8 或 4 个 16 位字进行编程。如果起始地址进行了 64 位对齐而不是 128 位对齐，则应同时对 4 个 16 位字进行编程。

pu16EccBuffer 的 LSB 与主阵列的低 64 位相对应，pu16EccBuffer 的 MSB 与主阵列的高 64 位相对应。

Fapi_calculateEcc() 函数可用于计算给定 64 位对齐地址和相应数据的 ECC。

Fapi_EccOnly – 该模式将仅在指定的地址处（应为该函数提供闪存主阵列地址，而不是相应的 ECC 地址）对闪存 ECC 存储空间中的 ECC 部分进行编程。它可以对 2 字节（ECC 存储器中某一位置的 LSB 和 MSB）或 1 字节（ECC 存储器中某一位置的 LSB）编程。

pu16EccBuffer 的 LSB 与主阵列的低 64 位相对应，pu16EccBuffer 的 MSB 与主阵列的高 64 位相对应。

使用该模式时将忽略参数二和三。

备注

pu16DataBuffer 和 pu16EccBuffer 的长度分别不可超过 8 和 2。

备注

该函数在发出编程命令后不检查 FMSTAT。当 FSM 完成编程操作时，用户应用程序必须检查 FMSTAT 值。FMSTAT 指示编程操作期间是否有任何故障发生。用户应用程序可以使用 Fapi_getFsmStatus 函数来获取 FMSTAT 值。

此外，用户应用程序应使用 Fapi_doVerify() 函数来验证闪存是否已正确编程。

该函数不会等到编程操作结束；它只是发出命令并返回。因此，用户应用程序必须等待 FMC 完成编程操作，然后才能返回到任何类型的闪存访问。应将 Fapi_checkFsmForReady() 函数用于监测已发出命令的状态。

限制

- 如上所述，该函数一次最多只能对 128 位进行编程（鉴于提供的地址进行了 128 位对齐）。如果用户想对更多位进行编程，则应循环调用该函数，从而一次对 128 位（或应用程序所需的 64 位）进行编程。
- 主阵列闪存编程必须与 64 位地址边界对齐，并且每个 64 位字在每个写入或擦除周期只能编程一次。
- 可以单独对数据和 ECC 进行编程。但是，每个 64 位数据字和相应的 ECC 字在每个写入或擦除周期中只能编程一次。
- DCSM OTP 编程必须与 128 位地址边界对齐，并且每个 128 位字只能编程一次。例外包括：
  - DCSM OTP 中的 DCSM Zx-LINKPOINTER1 和 Zx-LINKPOINTER2 值应一起编程，并且可以按照 DCSM 操作的要求一次编程 1 位。
  - DCSM OTP 中的 DCSM Zx-LINKPOINTER3 值可按照 DCSM 操作的要求，一次编程 1 位。
- 不应针对链接指针位置对 ECC 进行编程。即使用户选择 Fapi_AutoEccGeneration 模式或 Fapi_DataAndEcc 模式，API 也会为此类位置发出 Fapi_DataOnly 命令。链路指针位置不支持 Fapi_EccOnly 模式。
- Fapi_EccOnly 模式不应该用于存储体 0 DCSM OTP 空间。如果使用，将返回错误。对于 DCSM OTP 空间，应使用 Fapi_AutoEccGeneration 或 Fapi_DataAndEcc 编程模式。

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Error_InvalidBaseRegCntlAddress**（失败：用户提供的闪存控制寄存器基地址与预期地址不匹配）
- **Fapi_Error_AsyncIncorrectDataBufferLength**（失败：指定的数据缓冲区大小不正确。此外，如果在对存储体 0 DCSM OTP 空间进行编程时选择了 Fapi_EccOnly 模式，则会返回该错误）
- **Fapi_Error_AsyncIncorrectEccBufferLength**（失败：指定的 ECC 缓冲区大小不正确）
- **Fapi_Error_AsyncDataEccBufferLengthMismatch**（失败：数据缓冲区大小不是 64 位对齐的，或者数据长度跨越了 128 位对齐的存储器边界）
- **Fapi_Error_FlashRegsNotWritable**（失败：闪存寄存器写入失败。用户应确保 API 从与闪存操作的目标地址相同的区域执行，或者用户应在闪存操作之前解锁。）
- **Fapi_Error_FeatureNotAvailable**（失败：用户传递了不受支持的模式）
- **Fapi_Error_InvalidAddress**（失败：用户提供的地址无效。有关有效地址范围的信息，请参阅 *TMS320F28003x 微控制器数据手册*。）

实现示例

该示例未显示擦除操作。请注意，扇区在重新编程之前应被擦除。

```
#include “F021_F28003x_C28x.h”
#define CPUCLK_FREQUENCY   120   /* 120 MHz System frequency */
int main(void)
{
     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
     //
     Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);

     //
     // Jump to RAM and call the Flash API functions
     //
     Example_CallFlashAPI();
}
#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
     Fapi_StatusType oReturnCheck;
     Fapi_FlashStatusType oFlashStatus;
     uint16 au16DataBuffer[8] = {0x0001, 0x0203, 0x0405, 0x0607, 0x0809, 0x0A0B, 0x0C0D, 0x0E0F};
     uint32 *DataBuffer32 = (uint32 *)au16DataBuffer;
     uint32 u32Index = 0;
     EALLOW;
     //
     // This function is required to initialize the Flash API based on
     // System frequency before any other Flash API operation can be performed
     // Note that the FMC register base address and system frequency are passed as the parameters
     //
     oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, CPUCLK_FREQUENCY);
     if(oReturnCheck != Fapi_Status_Success)
     {
          Example_Error(oReturnCheck);
     }

     //
     // Fapi_setActiveFlashBank function initializes Flash banks
     // and FMC for erase and program operations.
     //
     oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);

     if(oReturnCheck != Fapi_Status_Success)
     {
          Example_Error(oReturnCheck);
     }
     //
     // Bank0 Program
     //

     //
     // Program 0x200 16-bit words in Bank0 Sector 4
     //

     for(u32Index = 0x84000; (u32Index < 0x84200) &&
                             (oReturnCheck == Fapi_Status_Success); u32Index+=8)
     {
          //
          // Issue program command
          //
          oReturnCheck = Fapi_issueProgrammingCommand((uint32 *)u32Index, au16DataBuffer, 8,
                                                      0, 0, Fapi_AutoEccGeneration);
          //
          // Wait until the Flash program operation is over
          //
          while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}
          if(oReturnCheck != Fapi_Status_Success)
          {
               Example_Error (oReturnCheck);
          }
          //
```

```
            // 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(oFlashStatus != 0)
            {
                 //
                 //Check FMSTAT and debug accordingly
                 //
                 FMSTAT_Fail();
            }

            //
            // Verify the programmed values
            //
            oReturnCheck = Fapi_doVerify((uint32 *)u32Index, 4, DataBuffer32, &oFlashStatusWord);
            if(oReturnCheck != Fapi_Status_Success)
            {
                 //
                 // Check Flash API documentation for possible errors
                 //
                 Example_Error(oReturnCheck);
            }
       }
       //
       // * User code for further Bank0 flash operations *
       //
       .
       .
       .
       .

       //
       // Example is done here
       //
       Example_Done();

}
```

3.2.5 Fapi_issueProgrammingCommandForEccAddresses()

将 ECC 地址重新映射到数据地址并调用 Fapi_issueProgrammingCommand()。

概要

```
Fapi_StatusType Fapi_issueProgrammingCommandForEccAddress(
        uint32 *pu32StartAddress,
        uint16 *pu16EccBuffer,
        uint16  u16EccBufferSizeInBytes)
```

参数

pu32StartAddress [in]	闪存中 ECC 的起始地址，用于对 ECC 进行编程
pu16EccBuffer [in]	指向 ECC 缓冲区地址的指针
u16EccBufferSizeInBytes [in]	ECC 缓冲区中的字节数 如果字节数为 1，则对 LSB（针对低 64 位的 ECC）进行编程。无法使用该函数单独对 MSB 进行编程。如果字节数为 2，则将对 ECC 的 LSB 和 MSB 字节进行编程。

说明

该函数会将 ECC 存储空间中的地址重新映射到对应的数据地址空间，然后调用 Fapi_issueProgrammingCommand() 对提供的 ECC 数据进行编程。使用 Fapi_EccOnly 模式的 Fapi_issueProgrammingCommand() 的限制同样也适用于该函数。pu16EccBuffer 的 LSB 与主阵列的低 64 位相对应，pu16EccBuffer 的 MSB 与主阵列的高 64 位相对应。

备注

pu16EccBuffer 的长度不可超过 2。

备注

该函数在发出编程命令后不检查 FMSTAT。当 FSM 完成编程操作时，用户应用程序必须检查 FMSTAT 值。FMSTAT 指示编程操作期间是否有任何故障发生。用户应用程序可以使用 Fapi_getFsmStatus 函数来获取 FMSTAT 值。

备注

Fapi_EccOnly 模式不应该用于存储体 0 DCSM OTP 空间。如果使用，将返回错误。对于 DCSM OTP 空间，应使用 Fapi_AutoEccGeneration 或 Fapi_DataAndEcc 编程模式。

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Error_InvalidBaseRegCntlAddress**（失败：用户提供的闪存控制寄存器基地址与预期地址不匹配）
- **Fapi_Error_AsyncIncorrectEccBufferLength**（失败：指定的数据缓冲区大小不正确）
- **Fapi_Error_FlashRegsNotWritable**（失败：闪存寄存器写入失败。用户应确保 API 从与闪存操作的目标地址相同的区域执行，或者用户应在闪存操作之前解锁。）
- **Fapi_Error_InvalidAddress**（失败：用户提供的地址无效。有关有效地址范围的信息，请参阅 *TMS320F28003x 微控制器数据手册*。）

3.2.6 Fapi_issueFsmSuspendCommand()

发出闪存状态机暂停命令

概要

```
Fapi_StatusType Fapi_issueFsmSuspendCommand(void)
```

参数

无

说明

该函数发出立即暂停命令，会暂停当前有效的 FSM 命令（Program 和 Erase Sector）。使用 Fapi_getFsmStatus() 检查操作是否成功。无法暂停存储体擦除命令，并且将返回指示失败的错误。

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Error_Fail**（失败，如果存储体擦除是当前有效命令）

3.2.7 Fapi_issueAsyncCommand()

向闪存状态机发出命令。有关该函数可发出的命令列表，请参阅说明。

概要

```
Fapi_StatusType Fapi_issueAsyncCommand(
        Fapi_FlashStateCommandsType oCommand)
```

参数

oCommand [in]	向 FSM 发出的命令

说明

对于不需要任何附加信息（如地址）的命令，该函数向闪存状态机发出命令。典型命令为 Clear Status、Program Resume、Erase Resume 和 Clear_More。该函数不会等到命令结束；它只是发出命令并返回。因此，用户应用程序必须等待 FMC 完成给定的命令，然后才能返回到任何类型的闪存访问。Fapi_checkFsmForReady() 函数可用于监测已发出命令的状态。

以下是此类命令的详细信息：

- Fapi_ClearStatus：执行该命令会清除 FMSTAT 寄存器中的 ILA、PGV、EV、CSTAT 和 INVDAT 位。闪存 API 在发出编程或擦除命令之前发出该命令。
- Fapi_ClearMore：执行此命令会清除 Clear Status 命令可清除的所有内容，此外还会将 FMSTAT 寄存器中的 ESUSP 位和 PSUSP 位清零。
- Fapi_ProgramResume：执行该命令将恢复先前暂停的编程操作。在暂停命令未处于有效状态时发出恢复命令会不起作用。请注意，如果之前的编程操作处于暂停状态，则无法启动新的编程操作。编程恢复 (Fapi_ProgramResume) 命令完成后，应使用 Fapi_doVerify() 来确保按照要求对数据进行编程。如果未对数据进行编程，用户应用程序应通过调用编程函数重新对相应的数据进行编程。
- Fapi_EraseResume：执行该命令将恢复先前暂停的擦除操作。在暂停命令未处于有效状态时发出恢复命令会不起作用。请注意，如果之前的擦除操作处于暂停状态，则无法启动新的擦除操作。

备注

该函数在发出命令后不检查 FMSTAT。当 FSM 完成操作时，用户应用程序必须检查 FMSTAT 值。FMSTAT 指示操作期间是否有任何故障发生。用户应用程序可以使用 Fapi_getFsmStatus 函数来获取 FMSTAT 值。

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Error_FeatureNotAvailable**（失败：用户传递了不受支持的命令）

实现示例

```
#include “F021_F28003x_C28x.h”
#define CPUCLK_FREQUENCY   120   /* 120 MHz System frequency */
int main(void)
{
     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
     //
     Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);

     //
     // Jump to RAM and call the Flash API functions
     //
     Example_CallFlashAPI();
}
#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
     Fapi_StatusType oReturnCheck;
     Fapi_FlashStatusType oFlashStatus;
     uint16 au16DataBuffer[8] = {0x0001, 0x0203, 0x0405, 0x0607, 0x0809, 0x0A0B, 0x0C0D, 0x0E0F};
     uint32 *DataBuffer32 = (uint32 *)au16DataBuffer;
     uint32 u32Index = 0;
     //
     // Bank0 operations
     //
     EALLOW;
     //
     // This function is required to initialize the Flash API based on
     // System frequency before any other Flash API operation can be performed
     // Note that the FMC register base address and system frequency are passed as the parameters
     //
     oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, CPUCLK_FREQUENCY);
     if(oReturnCheck != Fapi_Status_Success)
     {
          Example_Error(oReturnCheck);
     }

     //
     // Fapi_setActiveFlashBank function initializes Flash banks
     // and FMC for erase and program operations.
     //
     oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);

     if(oReturnCheck != Fapi_Status_Success)
     {
          Example_Error(oReturnCheck);
     }
     //
     // Issue an async command
     //
     oReturnCheck = Fapi_issueAsyncCommand(Fapi_ClearMore);
     //
     // Wait until the Fapi_ClearMore operation is over
     //
     while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}
     if(oReturnCheck != Fapi_Status_Success)
     {
         Example_Error (oReturnCheck);
     }
     //
     // 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(oFlashStatus != 0)
     {
```

```
            //
            //Check FMSTAT and debug accordingly
            //
            FMSTAT_Fail();
        }

        //
        // * User code for further Bank0 flash operations *
        //
        .
        .
        .
        .


        EDIS;
        //
        // Example is done here
        //
        Example_Done();

}
```

3.2.8 Fapi_checkFsmForReady()

返回闪存状态机的状态

概要

```
Fapi_StatusType Fapi_checkFsmForReady(void)
```

参数

无

说明

该函数返回闪存状态机的状态，指示其是否准备好接受新命令。主要用途是检查擦除或编程操作是否已完成。

返回值

- **Fapi_Status_FsmBusy**（FSM 处于繁忙状态，除暂停命令外，无法接受新命令）
- **Fapi_Status_FsmReady**（FSM 已准备好接受新命令）

3.2.9 Fapi_getFsmStatus()

返回 FMSTAT 寄存器的值

概要

```
Fapi_FlashStatusType Fapi_getFsmStatus(void)
```

参数

无

说明

该函数返回 FMSTAT 寄存器的值。该寄存器允许用户应用程序确定擦除或编程操作是成功完成、正在进行、暂停还是失败。用户应用程序应检查该寄存器的值从而确定每次擦除和编程操作后是否有任何故障。

返回值

表 3-2. FMSTAT 寄存器

位 31	...	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留					PGV	保留	EV	保留	Busy	ERS	PGM	INV DAT	CSTAT	保留	ESUSP	PSUSP	保留

表 3-3. FMSTAT 寄存器字段说明

位	字段	说明
31 - 13	RSVD	保留
12	PGV	编程验证。设置后，表示在为编程操作提供允许的最大编程脉冲数后，未成功对字进行编程。
11	RSVD	保留
10	EV	擦除验证。设置后，表示在为擦除操作指定了允许的最大擦除脉冲数后，未成功擦除扇区。在执行擦除验证命令期间，如果发现某个位为 0，则立即设置该标志。
9	RSVD	保留
8	Busy	设置后，该位表示正在处理编程、擦除或暂停操作。
7	ERS	擦除有效。设置后，该位表示闪存模块正在主动进行擦除操作。该位在擦除开始时设置，并在擦除完成时清除。当擦除暂停时，该位也会被清除。当擦除恢复时，该位也会被设置。
6	PGM	编程有效。设置后，该位表示闪存模块当前正在进行编程操作。该位在编程开始时设置，并在编程完成时清除。当编程暂停时，该位也会被清零。当编程恢复时，该位也会被设置。
5	INVDAT	无效数据。设置后，该位表示用户尝试在已存在 "0" 的情况下对 "1" 进行编程。该位由 Clear Status 命令清零。
4	CSTAT	命令状态。一旦 FSM 启动，出现任何故障都会设置该位。设置后，该位通知主机编程或擦除命令失败并且命令已停止。该位由 Clear Status 命令清零。对于某些错误，这将是 FSM 错误的唯一指示，因为其原因不属于其他错误位类型。
3	RSVD	RSVD
2	ESUSP	擦除暂停。设置后，该位表示闪存模块已接收并处理擦除暂停操作。在发出擦除恢复命令或执行 Clear_More 命令之前，该位保持设置状态。
1	PSUSP	编程暂停。设置后，该位表示闪存模块已接收并处理编程暂停操作。在发出编程恢复命令或执行 Clear_More 命令之前，该位保持设置状态。
0	RSVD	RSVD

3.3 读取函数

3.3.1 Fapi_doBlankCheck()

验证指定区域是否为擦除值

概要

```
Fapi_StatusType Fapi_doBlankCheck(
        uint32 *pu32StartAddress,
        uint32  u32Length,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

参数

pu32StartAddress [in]		要进行空白检查的区域的起始地址
u32Length [in]		要进行空白检查的区域长度（以 32 位字为单位）
poFlashStatusWord [out]		如果结果不是 Fapi_Status_Success，则返回操作状态
	->au32StatusWord[0]	第一个非空白位置的地址
	->au32StatusWord[1]	在第一个非空白位置读取的数据
	->au32StatusWord[2]	比较数据的值（始终为 0xFFFFFFFF）
	->au32StatusWord[3]	不适用

说明

该函数在从指定地址开始的指定长度（以 32 位字为单位）的区域内，检查闪存是否为空白（擦除状态）。如果发现非空白位置，则在 poFlashStatusWord 参数中返回相应的地址和数据。

限制

无

返回值

- **Fapi_Status_Success（成功）**：发现指定的闪存位置处于已擦除状态
- **Fapi_Error_Fail**（失败：指定区域非空白）
- **Fapi_Error_InvalidAddress**（失败：用户提供的地址无效。有关有效地址范围的信息，请参阅 *TMS320F28003x 微控制器数据手册*。）

3.3.2 Fapi_doVerify()

根据提供的数据验证指定区域

概要

```
Fapi_StatusType Fapi_doVerify(
        uint32 *pu32StartAddress,
        uint32  u32Length,
        uint32 *pu32CheckValueBuffer,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

参数

pu32StartAddress [in]		要验证区域的起始地址
u32Length [in]		要验证的区域长度（以 32 位字为单位）
pu32CheckValueBuffer [in]		用于验证区域的缓冲区地址。数据缓冲区应为 128 位对齐。
poFlashStatusWord [out]		如果结果不是 Fapi_Status_Success，则返回操作状态
	->au32StatusWord[0]	首次验证失败位置的地址
	->au32StatusWord[1]	在首次验证失败的位置读取的数据
	->au32StatusWord[2]	比较数据的值
	->au32StatusWord[3]	不适用

说明

该函数在从指定地址开始的指定长度（以 32 位字为单位）的区域内，根据提供的数据验证器件。如果位置比较失败，这些结果将在 poFlashStatusWord 参数中返回。

限制

无

返回值

- **Fapi_Status_Success**（成功：指定的区域与提供的数据匹配）
- **Fapi_Error_FAIL**（失败：指定区域与提供的数据不匹配）
- **Fapi_Error_InvalidAddress**（失败：用户提供的地址无效。有关有效地址范围的信息，请参阅 *TMS320F28003x 微控制器数据手册*。）

备注

在 F28003x 器件上，当读取存储体 0 用户可配置 DCSM OTP 中相应的新编程密码位置时，这些区域将被锁定。这可能会导致进一步的擦除/编程操作失败。因此，不建议对用户可配置的 DCSM OTP 存储体 0 位置进行验证 (Fapi_doVerify())。

3.3.3 Fapi_calculatePsa()

计算指定区域的 PSA

概要

```
uint32 Fapi_calculatePsa(
        uint32 *pu32StartAddress,
        uint32  u32Length,
        uint32  u32PsaSeed,
        Fapi_FlashReadMarginModeType oReadMode)
```

参数

pu32StartAddress [in]	用于计算 PSA 值的区域起始地址
u32Length [in]	用于计算 PSA 值的区域长度（以 32 位字为单位）
u32PsaSeed [in]	PSA 计算的种子值
oReadMode [in]	仅适用于正常模式。使用 Fapi_NormalRead。

说明

该函数使用 u32PsaSeed 值计算从 pu32StartAddress 开始的 u32Length 32 位字的指定区域的 PSA 值。附录 E 给出了 PSA 算法。

限制

无

返回值

- **PSA 值**（成功）
- **0xA5A5A5A5U**（失败：用户提供的地址无效。有关有效地址范围的信息，请参阅 *TMS320F28003x 微控制器数据手册*。）

3.3.4 Fapi_doPsaVerify()

根据指定的 PSA 值验证指定区域

概要

```
Fapi_StatusType Fapi_doPsaVerify(
        uint32 *pu32StartAddress,
        uint32  u32Length,
        uint32  u32PsaValue,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

参数

pu32StartAddress [in]		用于验证 PSA 值的区域起始地址
u32Length [in]		用于验证 PSA 值的区域长度（以 32 位字为单位）
u32PsaValue [in]		用于比较区域的 PSA 值
poFlashStatusWord [out]		如果结果不是 Fapi_Status_Success，则返回操作状态
	->au32StatusWord[3]	实际 PSA

说明

该函数在从指定地址开始的指定长度（以 32 位字为单位）的区域内，根据提供的 PSA 值验证器件。计算出的 PSA 值在 poFlashStatusWord 参数中返回。

限制

无

返回值

- **Fapi_Status_Success**（成功）
- **Fapi_Error_Fail**（失败：指定区域与提供的数据不匹配）
- **Fapi_Error_InvalidAddress**（失败：用户提供的地址无效。有关有效地址范围的信息，请参阅 *TMS320F28003x 微控制器数据手册*。）

3.4 信息函数

3.4.1 Fapi_getLibraryInfo()

返回有关该闪存 API 编译的信息

概要

```
Fapi_LibraryInfoType Fapi_getLibraryInfo(void)
```

参数

无

说明

该函数返回特定于闪存 API 库编译的信息。该信息在 Fapi_LibraryInfoType 结构中返回。该结构的成员如下：

- u8ApiMajorVersion – 该 API 编译的主要版本号。该值为 1。
- u8ApiMinorVersion – 该 API 编译的次要版本号。F28003x 器件的次要版本为 58。
- u8ApiRevision – 该 API 编译的修订版本号。

  FAPI_F280033_34_EABI_v1.58.11.lib 和 FAPI_F280033_34_COFF_v1.58.11.lib 的修订版号为 11。

  FAPI_F28003x_EABI_v1.58.10.lib 和 FAPI_F28003x_COFF_v1.58.10.lib 的修订版号为 01。但是，FAPI_F28003x_EABI_v1.58.10.lib 和 FAPI_F28003x_COFF_v1.58.10.lib 的 Fapi_getLibraryInfo() 会返回修订版号 01，而不是 10。这是一个已知问题，如勘误表（本文档的附录 G）所述。
- oApiProductionStatus – 该编译的生产状态 *（Alpha_Internal、Alpha、Beta_Internal、Beta、生产）*

  该版本的生产状态为“生产”。
- u32ApiBuildNumber – 该编译的构建版本号。用于区分不同的 α 和 β 构建。

  该版本的构建版本号为 0。
- u8ApiTechnologyType – 表示 API 支持的闪存技术。该字段返回值为 0x4。
- u8ApitechnologyRevision – 表示 API 支持的技术的修订版
- u8ApiEndianness – 对于 F28003x 器件，此字段始终返回 1（小端字节序）。
- u32ApiCompilerVersion – 用于编译 API 的 Code Composer Studio 代码生成工具的版本号

返回值

- **Fapi_LibraryInfoType**（提供有关该 API 编译的检索信息）

3.5 实用功能

3.5.1 Fapi_flushPipeline()

刷新 FMC 流水线缓冲区

概要

```
void Fapi_flushPipeline(void)
```

参数

无

说明

该函数可刷新 FMC 数据缓存。在进行擦除或编程操作后，必须在读取第一个非 API 闪存之前刷新数据缓存。

返回值

无

3.5.2 Fapi_calculateEcc()

计算所提供地址和 64 位值的 ECC

概要

```
uint8 Fapi_calculateEcc(              uint32 u32Address,
              uint64 u64Data)
```

参数

u32Address [in]	用于计算 ECC 的 64 位值的地址
u64Data [in]	用于计算 ECC 的 64 位值（应采用小端字节序的顺序）

说明

该函数将计算包括地址在内的 64 位对齐字的 ECC。不再需要为该函数提供左移地址。TMS320F28003x 闪存 API 负责处理。

返回值

- 计算出的 8 位 ECC（应忽略 16 位返回值的高 8 位）
- 如果发生错误，16 位返回值为 0xDEAD

3.5.3 Fapi_isAddressEcc()

表示地址位于闪存控制器 ECC 空间

概要

```
boolean Fapi_isAddressEcc(
          uint32 u32Address)
```

参数

u32Address [in]	用于确定是否位于 ECC 地址空间的地址

说明

如果地址位于 ECC 地址空间，则该函数返回 TRUE，否则返回 FALSE。

返回值

- **FALSE**（地址不在 ECC 地址空间内）

- **TRUE**（地址位于 ECC 地址空间）

3.5.4 Fapi_remapEccAddress()

获取 ECC 地址并将其重新映射到主地址空间

概要

```
uint32 Fapi_remapEccAddress(
         uint32 u32EccAddress)
```

参数

u32EccAddress [in]	要重新映射的 ECC 地址

说明

该函数返回给定闪存 ECC 地址的主阵列闪存地址。当用户想要在已知 ECC 地址处对 ECC 数据进行编程时，可以使用该函数获取对应的主阵列地址。请注意，Fapi_issueProgrammingCommand() 函数需要一个主阵列地址而不是 ECC 地址（即使是在 Fapi_EccOnly 模式下）。

返回值

- **32 位主闪存地址**
- **0xFFFFFFFF（如果提供的闪存 ECC 地址无效）**

3.5.5 Fapi_calculateFletcherChecksum()

根据给定的地址和长度计算 Fletcher 校验和。

概要

```
uint32 Fapi_calculateFletcherChecksum(
              uint16 *pu16Data,
              uint16 u16Length)
```

参数

pu16Data [in]	计算校验和的起始地址
u16Length [in]	计算中要使用的 16 位字的数量

说明

该函数为指定数量的 16 位字从提供的地址开始生成 32 位 Fletcher 校验和。

返回值

- 32 位 Fletcher 校验和值

4 推荐的 FSM 流程

4.1 新出厂器件

器件出厂时已擦除。建议对收到的器件进行空白检查从而验证其是否已被擦除，但这不是必需的。

4.2 推荐的擦除流程

图 4-1 描述了器件上扇区的擦除流程。有关更多信息，请参阅节 3.2.2。

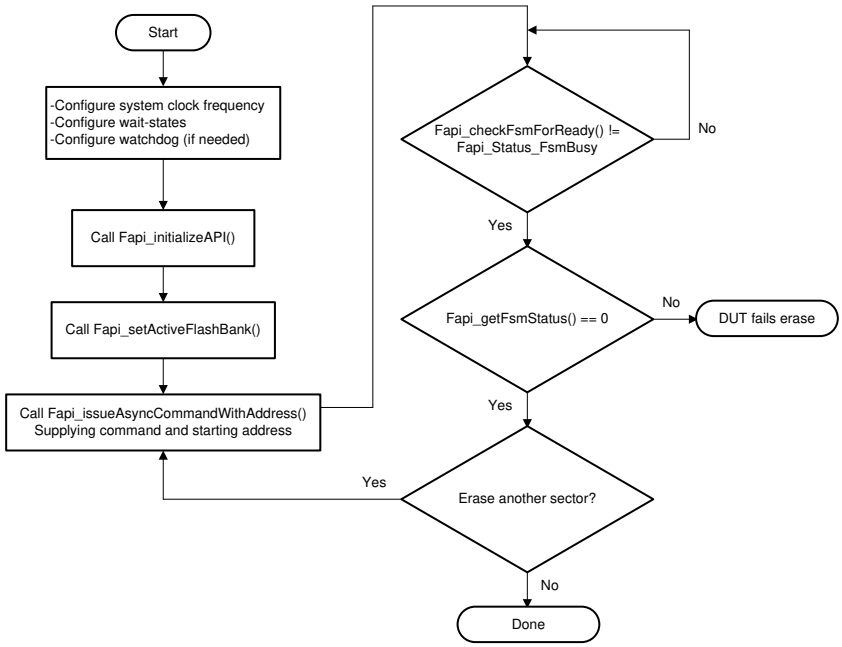

图 4-1. 推荐的擦除流程

4.3 推荐的存储体擦除流程

图 4-2 描述了闪存存储体的擦除流程。有关更多信息，请参阅节 3.2.3。

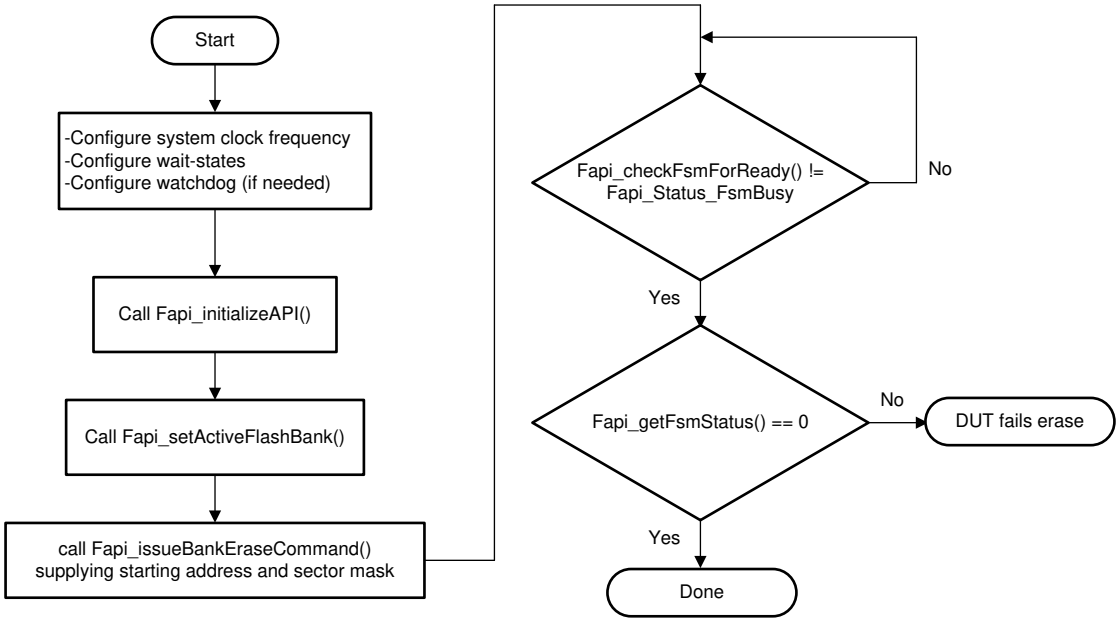

图 4-2. 推荐的存储体擦除流程

4.4 推荐的编程流程

图 4-3 描述了对器件进行编程的流程。该流程假定用户已经按照推荐的擦除流程擦除了所有受影响的扇区或存储体。有关更多信息，请参阅节 4.2。

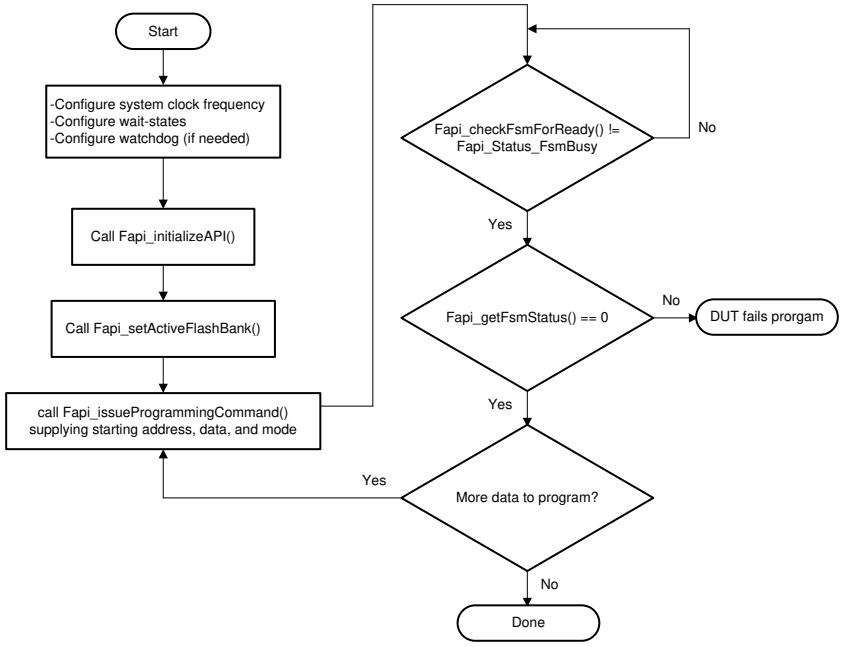

图 4-3. 推荐的编程流程

5 与安全相关的软件应用程序使用假设

1. 只有在执行 Fapi_initializeAPI() 时，才会对闪存 API 的全局变量进行初始化。因此，在使用任何其他闪存 API 函数之前，必须执行该函数一次。无需多次调用该函数。但是，只要用户应用程序在运行时更新系统频率或闪存等待状态配置（这种情况很少见），就必须调用该函数。
2. 在执行 Fapi_initializeAPI() 之前，用户应用程序必须确保为所需的输出频率正确配置 PLL。请注意，driverlib 中提供的 SysCtl_setClock() 通过使用 DCC 模块实现了这一点。SysCtl_setClock() 调用的 SysCtl_isPLLValid() 使用 DCC 模块来确保 PLL 输出处于预期范围内。有关 DCC 模块的更多详细信息，请参阅器件特定技术参考手册。
3. 如果用户应用程序需要保护闪存寄存器空间免受因代码失控等原因造成的任何损坏，则可以通过使用 ERAD 模块来实现。有关 ERAD 使用的详细信息，请参阅器件特定技术参考手册。
4. 当将 Fapi_BlankCheck() 用于闪存主阵列地址范围时，建议同时检查相应的 ECC 阵列地址范围。每个扇区的 ECC 地址范围在器件专用数据表的*存储器映射*表中给出。
5. 用户应用程序软件必须根据需要使用 ECC_ENABLE 寄存器为闪存读取/获取路径启用 ECC 评估功能。闪存 API 不启用/禁用该寄存器。请注意，ECC_ENABLE 在上电时默认启用。因此，用户应用程序软件定义 NMI ISR 和闪存 single-bit 错误 ISR，以便能够响应单个和不可纠正的闪存错误。有关 FLASH_ECC_REGS 寄存器的更多详细信息，请参阅器件特定技术参考手册。
6. 如果用户应用程序的安全标准要求其定期读取整个闪存范围，从而确保数据的完整性，用户应用程序可以这样做，并计算整个闪存范围的校验和/CRC。VCU CRC 模块可用于实现这一点。有关 VCU CRC 模块的更多详细信息，请参阅器件特定技术参考手册。
7. 如果用户应用程序的安全标准要求其定期读取闪存寄存器空间，从而确保寄存器配置的完整性，应用程序可以这样做，并计算闪存寄存器存储范围的校验和/CRC。
8. 如果用户应用程序的安全标准要求其在进行编程和擦除操作后验证（使用 CPU 读取）闪存内容，它可以分别使用 Fapi_doVerify() 和 Fapi_doBlankCheck() 来验证闪存内容。
9. 闪存 API 不对看门狗进行配置/维护。用户应用程序可以配置看门狗并根据需要对其进行维护（在闪存 API 函数调用之间进行串行或使用 CPU 计时器 ISR）。
10. 如果 RAM 或闪存中的闪存 API 代码由于应用程序设计不当（例如，堆栈溢出问题）而被覆盖，则当 CPU 尝试执行闪存 API 函数时，可能会出现非法指令陷阱 (ITRAP)。因此，用户应用程序定义 ITRAP ISR 来响应此类事件。

A 闪存状态机命令

A.1 闪存状态机命令

表 A-1. 闪存状态机命令

命令	说明	枚举类型	API 调用
Program Data	用于将数据编程到任何有效的闪存地址	Fapi_ProgramData	Fapi_issueProgrammingCommand() Fapi_issueProgrammingCommandForEccAddress()
Erase Sector	用于擦除指定地址处的闪存扇区	Fapi_EraseSector	Fapi_issueAsyncCommandWithAddress()
Erase Bank	用于擦除闪存存储体，可选择使用提供的扇区掩码	Fapi_EraseBank	Fapi_issueBankEraseCommand()
Clear Status	清除状态寄存器	Fapi_ClearStatus	Fapi_issueAsyncCommand()
Program Resume	恢复暂停的编程操作	Fapi_ProgramResume	Fapi_issueAsyncCommand()
Erase Resume	恢复暂停的擦除操作	Fapi_EraseResume	Fapi_issueAsyncCommand()
Clear More	清除状态寄存器	Fapi_ClearMore	Fapi_issueAsyncCommand()

B 编译器版本和构建设置

表 B-1. 编译器版本和构建设置

工具或字段	设置或值
TI C2000 编译器	TI v21.6.0.LTS 或更高版本
--float_support	fpu32
--opt_level	关闭
--opt_for_speed	2
--fp_mode	严格

C 目标库函数信息

C.1 TMS320F28003x 闪存 API 库

表 C-1. C28x 函数大小

函数名称	大小（以字为单位）
Fapi_calculateEcc	47
Fapi_calculateFletcherChecksum	44
Fapi_calculatePsa *包括对以下函数的引用：* • Fapi_isAddressEcc	25
Fapi_checkFsmForReady	14
Fapi_doBlankCheck *包括对以下函数的引用：* • Fapi_flushPipeline • Fapi_isAddressEcc	135
Fapi_doVerify *包括对以下函数的引用：* • Fapi_flushPipeline • Fapi_isAddressEcc	15
Fapi_flushPipeline	21
Fapi_getFsmStatus	7
Fapi_getLibraryInfo	31
Fapi_initializeAPI	46
Fapi_isAddressEcc	35
Fapi_issueAsyncCommand	32
Fapi_issueAsyncCommandWithAddress *包括对以下函数的引用：* • Fapi_setupBankSectorEnable	127
Fapi_issueBankEraseCommand	
Fapi_issueFsmSuspendCommand	51
Fapi_issueProgrammingCommand *包含对以下函数的引用：* • Fapi_calculateEcc • Fapi_setupBankSectorEnable	650
Fapi_issueProgrammingCommandForEccAddresses *包括对以下函数的引用：* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_remapEccAddress	21
Fapi_remapEccAddress	62
Fapi_setActiveFlashBank *包括对以下函数的引用：* • Fapi_calculateFletcherChecksum	61
Fapi_issueBankEraseCommand	170

D typedef、定义、枚举和结构

D.1 类型定义

```
#if defined(__TMS320C28XX__)
typedef unsigned char       boolean;
typedef unsigned int            uint8; /*This is 16 bits in C28x*/
typedef unsigned int            uint16;
typedef unsigned long int       uint32;
typedef unsigned long long int  uint64;
#endif
```

D.2 定义

```
#if (defined(__TMS320C28xx__) && __TI_COMPILER_VERSION__ < 6004000)
#if !defined(__GNUC__)
#error “F021 Flash API requires GCC language extensions.Use the -gcc option.”
#endif
#endif
#ifndef TRUE
   #define TRUE             1
#endif
#ifndef FALSE
   #define FALSE            0
#endif
```

D.3 枚举

D.3.1 Fapi_FlashProgrammingCommandsType

其中包含 Fapi_IssueProgrammingCommand() 中使用的所有可能模式。

```
typedef enum
{
Fapi_AutoEccGeneration,   /* This is the default mode for the command and will
                             auto generate the ecc for the provided data buffer */
   Fapi_DataOnly,         /* Command will only process the data buffer */
   Fapi_EccOnly,          /* Command will only process the ecc buffer */
   Fapi_DataAndEcc        /* Command will process data and ecc buffers */
}  ATTRIBUTE_PACKED Fapi_FlashProgrammingCommandsType;
```

D.3.2 Fapi_FlashBankType

用于指示正在使用哪个闪存存储体。

```
typedef enum
{
   Fapi_FlashBank0,
   Fapi_FlashBank1,
   Fapi_FlashBank2
}  ATTRIBUTE_PACKED Fapi_FlashBankType;
```

D.3.3 Fapi_FlashStateCommandsType

其中包含所有可能的闪存状态机命令。

```
typedef enum
{
   Fapi_ProgramData    = 0x0002,
   Fapi_EraseSector    = 0x0006,
   Fapi_EraseBank      = 0x0008,
   Fapi_ClearStatus    = 0x0010,
   Fapi_ProgramResume  = 0x0014,
   Fapi_EraseResume    = 0x0016,
   Fapi_ClearMore      = 0x0018
}  ATTRIBUTE_PACKED Fapi_FlashStateCommandsType;
```

D.3.4 Fapi_FlashReadMarginModeType

其中包含所有可能的闪存状态机命令。

```
typedef enum
{
   Fapi_NormalRead = 0x0,
}  ATTRIBUTE_PACKED Fapi_FlashReadMarginModeType;
```

D.3.5 Fapi_StatusType

其为包含所有可能返回的状态代码的主类型。

```
typedef enum
{
   Fapi_Status_Success=0,           /* Function completed successfully */
   Fapi_Status_FsmBusy,             /* FSM is Busy */
   Fapi_Status_FsmReady,            /* FSM is Ready */
   Fapi_Status_AsyncBusy,           /* Async function operation is Busy */
   Fapi_Status_AsyncComplete,       /* Async function operation is Complete */
   Fapi_Error_Fail=500,             /* Generic Function Fail code */
   Fapi_Error_OtpChecksumMismatch,  /* Returned if OTP checksum does not match expected value */
   Fapi_Error_InvalidDelayValue,    /* Returned if the Calculated RWAIT value exceeds 15  -
                                       Legacy Error */
   Fapi_Error_InvalidHclkValue,     /* Returned if FClk is above max FClk value -
                                       FClk is a calculated from SYSCLK and RWAIT */
   Fapi_Error_InvalidCpu,           /* Returned if the specified Cpu does not exist */
   Fapi_Error_InvalidBank,          /* Returned if the specified bank does not exist */
   Fapi_Error_InvalidAddress,       /* Returned if the specified Address does not exist in Flash
                                       or OTP */
   Fapi_Error_InvalidReadMode,      /* Returned if the specified read mode does not exist */
   Fapi_Error_AsyncIncorrectDataBufferLength,
   Fapi_Error_AsyncIncorrectEccBufferLength,
   Fapi_Error_AsyncDataEccBufferLengthMismatch,
   Fapi_Error_FeatureNotAvailable,  /* FMC feature is not available on this device */
   Fapi_Error_FlashRegsNotWritable, /* Returned if Flash registers are not writable due to
                                       security */
   Fapi_Error_InvalidCPUID,          /* Returned if OTP has an invalid CPUID */
   Fapi_Error_InvalidBaseRegCntlAddress, /* Returned if base address of register control is
incorrect */
   Fapi_Warning_BaseRegCntlAddressMismatch /* Returned if base address of register control is
incorrect */
}  ATTRIBUTE_PACKED Fapi_StatusType;
```

D.3.6 Fapi_ApiProductionStatusType

此处列出了 API 可能的不同生产状态值。

```
typedef enum
{
   Alpha_Internal,          /* For internal TI use only.  Not intended to be used by customers */
   Alpha,                   /* Early Engineering release.  May not be functionally complete */
   Beta_Internal,           /* For internal TI use only.  Not intended to be used by customers */
   Beta,                    /* Functionally complete, to be used for testing and validation */
   Production               /* Fully validated, functionally complete, ready for production use */
}  ATTRIBUTE_PACKED Fapi_ApiProductionStatusType;
```

D.4 结构

D.4.1 Fapi_FlashStatusWordType

该结构用于在需要更大灵活性的函数中返回状态值

```
typedef struct
{
   uint32 au32StatusWord[4];
}  ATTRIBUTE_PACKED Fapi_FlashStatusWordType;
```

D.4.2 Fapi_LibraryInfoType

用于返回 API 信息的结构如下：

```
typedef struct
{
   uint8  u8ApiMajorVersion;
   uint8  u8ApiMinorVersion;
   uint8  u8ApiRevision;
   Fapi_ApiProductionStatusType oApiProductionStatus;
   uint32 u32ApiBuildNumber;
   uint8  u8ApiTechnologyType;
   uint8  u8ApiTechnologyRevision;
   uint8  u8ApiEndianness;
   uint32 u32ApiCompilerVersion;
} Fapi_LibraryInfoType;
```

E 并行签名分析 (PSA) 算法

E.1 函数详细信息

Fapi_doPsaVerify() 和 Fapi_calculatePsa() 函数使用并行签名分析 (PSA) 算法。这些函数通常用于验证闪存中是否对特定模式进行编程，而无需传输完整的数据模式。PSA 签名基于如下基元多项式：

```
f(X) = 1 + X + X^2 + X^22 + X^31
uint32 calculatePSA (uint32* pu32StartAddress,
                     uint32 u32Length, /* Number of 32-bit words */
                     uint32 u32InitialSeed)
{
     uint32 u32Seed, u32SeedTemp;
     u32Seed = u32InitialSeed;
     while(u32Length--)
     {
        u32SeedTemp = (u32Seed << 1)^*(pu32StartAddress++);
        if(u32Seed & 0x80000000)
        {
            u32SeedTemp ^= 0x00400007; /* XOR the seed value with mask */
        }
        u32Seed = u32SeedTemp;
     }
     return u32Seed;
}
```

F ECC 计算算法

F.1 函数详细信息

下面的函数可用于计算给定的 64 位对齐地址（无需左移地址）和相应的 64 位数据的 ECC。

```
//
//Calculate the ECC for an address/data pair
//
uint16 CalcEcc(uint32 address, uint64 data)
{
                const uint32 addrSyndrome[8] =  {0x554ea, 0x0bad1, 0x2a9b5, 0x6a78d,
                                                 0x19f83, 0x07f80, 0x7ff80, 0x0007f};
                const uint64 dataSyndrome[8] =  {0xb4d1b4d14b2e4b2e, 0x1557155715571557,
                                                 0xa699a699a699a699, 0x38e338e338e338e3,
                                                 0xc0fcc0fcc0fcc0fc, 0xff00ff00ff00ff00,
                                                 0xff0000ffff0000ff, 0x00ffff00ff0000ff};
                const uint16 parity = 0xfc;
                uint64 xorData;
                uint32 xorAddr;
                uint16 bit, eccBit, eccVal;

                //
                //Extract bits "20:2" of the address
                //
                address = (address >> 2) & 0x7ffff;
                //
                //Compute the ECC one bit at a time.
                //
                eccVal = 0;
                for (bit = 0; bit < 8; bit++)
                {
                      //
                      //Apply the encoding masks to the address and data
                      //
                      xorAddr = address & addrSyndrome[bit];
                      xorData = data & dataSyndrome[bit];
                      //
                      //Fold the masked address into a single bit for parity calculation.
                      //The result will be in the LSB.
                      //
                      xorAddr = xorAddr ^ (xorAddr >> 16);
                      xorAddr = xorAddr ^ (xorAddr >> 8);
                      xorAddr = xorAddr ^ (xorAddr >> 4);
                      xorAddr = xorAddr ^ (xorAddr >> 2);
                      xorAddr = xorAddr ^ (xorAddr >> 1);
                      //
                      //Fold the masked data into a single bit for parity calculation.
                      //The result will be in the LSB.
                      //
                      xorData = xorData ^ (xorData >> 32);
                      xorData = xorData ^ (xorData >> 16);
                      xorData = xorData ^ (xorData >> 8);
                      xorData = xorData ^ (xorData >> 4);
                      xorData = xorData ^ (xorData >> 2);
                      xorData = xorData ^ (xorData >> 1);
                      //
                      //Merge the address and data, extract the ECC bit, and add it in
                      //
                      eccBit = ((uint16)xorData ^ (uint16)xorAddr) & 0x0001;
                      eccVal |= eccBit << bit;
                }

                //
                //Handle the bit parity.For odd parity, XOR the bit with 1
                //
                eccVal ^= parity;
                return eccVal;
}
```

G 勘误

公告 1：

FAPI_F28003x_EABI_v1.58.10.lib 的 Fapi_getLibraryInfo() 返回的修订版号不正确

受影响的闪存 API 版本：

FAPI_F28003x_EABI_v1.58.10.lib。

详细信息：

Fapi_getLibraryInfo() 的编码错误，因此将修订版号返回为 01 而不是 10。如果应用程序不使用此函数，则此公告不适用。

权变措施：

用户应用程序应将 01 视为 10。

公告 2：

对于 F280033/34 器件，FAPI_F28003x_EABI_v1.58.10.lib 不支持 DCSM OTP 编程。此公告不适用于非 F280033/34 器件。

受影响的闪存 API 版本：

FAPI_F28003x_EABI_v1.58.10.lib。

详细信息：

闪存 API 编程命令将阻止 DCSM OTP 范围的编程。如果 F280033/34 用户应用程序不对 DCSM OTP 进行编程，则此公告不适用。

权变措施：

如果 F280033/34 用户应用程序需要对 DCSM OTP 进行编程，则应使用 FAPI_F280033_34_EABI_v1.58.11.lib。非 F280033/34 用户应继续使用 FAPI_F28003x_EABI_v1.58.10.lib。

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 B (November 2022) to Revision C (November 2023)	Page
• 更新了 节 2.1	4
• 更新了 节 2.3.3	6
• 更新了节 3.4.1。	27
• 添加了附录 G	41

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