*Reference Guide*

# *TMS320F280013x Flash API Version 2.00.01.00*

## Table of Contents



## List of Figures



## List of Tables

## Trademarks

C2000™ is a trademark of Texas Instruments.
Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All trademarks are the property of their respective owners.

# 1 Introduction

This reference guide provides a detailed description of Texas Instruments' TMS320F280013x Flash API Library (FAPI_F280013x _EABI_v2.00.01.lib or FAPI_F280013x _COFF_v2.00.01.lib) functions that can be used to erase, program and verify Flash on TMS320F280013x devices. Note that Flash API V2.00.XX.XX should be used only with TMS320F280013x devices. The Flash API Library is provided in C2000Ware at C2000Ware_x_xx_xx_xx\libraries\flash_api\f280013x.

## 1.1 Reference Material

Use this guide in conjunction with *TMS320F280013x Microcontrollers Data Manual* and *TMS320F280013x Microcontrollers Technical Reference Manual*.

## 1.2 Function Listing Format

This is the general format of an entry for a function, compiler intrinsic, or macro.

A short description of what **function_name()** does.

### Synopsis

Provides a prototype for **function_name()**.

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

### Parameters

| | |
|---|---|
| *parameter_1 [in]* | Type details of parameter_1 |
| *parameter_2 [out]* | Type details of parameter_2 |
| *parameter_n [in/out]* | Type details of parameter_3 |

Parameter passing is categorized as follows:

- *in* — Indicates the function uses one or more values in the parameter that you give it without storing any changes.
- *out* — Indicates the function saves one or more of the values in the parameter that you give it. You can examine the saved values to find out useful information about your application.
- *in/out* — Indicates the function changes one or more of the values in the parameter that you give it and saves the result. You can examine the saved values to find out useful information about your application.

### Description

Describes the function. This section also describes any special characteristics or restrictions that might apply:

- Function blocks or might block the requested operation under certain conditions
- Function has pre-conditions that might not be obvious
- Function has restrictions or special behavior

### Restrictions

Specifies any restrictions in using this function.

**Return Value**

Specifies any value or values returned by the function.

**See Also**

Lists other functions or data types related to the function.

**Sample Implementation**

Provides an example (or a reference to an example) that illustrates the use of the function. Along with the Flash API functions, these examples may use the functions from the device_support folder or driverlib folder provided in C2000Ware, to demonstrate the usage of a given Flash API function in an application context.

# 2 TMS320F280013x Flash API Overview

## 2.1 Introduction

The Flash API is a library of routines, that when called with the proper parameters in the proper sequence, erases, programs, or verifies Flash memory. The Flash API can be used to program the OTP memory as well.

---

**Note**

Read the data manual for the Flash and OTP memory map and Flash waitstate specifications. Note that this reference guide assumes that the user has already read the *Flash and OTP Memory* chapter in the *TMS320F280013x Microcontrollers Technical Reference Manual*.

Also, pay special attention to the functions Fapi_issueAsyncCommand(), Fapi_setupBankSectorEnable(), Fapi_issueBankEraseCommand() on this device. Usage of these functions is demonstrated in the flash API usage example provided in C2000Ware at "C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c"

---

## 2.2 API Overview

**Table 1-1. Summary of Initialization Functions**

| API Function | | Description |
|---|---|---|
| Fapi_initializeAPI() | | Initializes the API for first use or frequency change |

**Table 2-2. Summary of Flash State Machine (FSM) Functions**

| API Function | Description |
|---|---|
| Fapi_setActiveFlashBank() | Initializes Flash Wrapper and bank for an erase or program command |
| Fapi_setupBankSectorEnable() | Configures the Write/Erase protection for the sectors. |
| Fapi_issueBankEraseCommand() | Issues bank erase command to the Flash State Machine for the given bank address. |
| Fapi_issueAsyncCommandWithAddress() | Issues an erase sector command to FSM for the given address |
| Fapi_issueProgrammingCommand() | Sets up the required registers for programming and issues the command to the FSM |
| Fapi_issueProgrammingCommandForEccAddress() | Remaps an ECC address to the main data space and then call Fapi_issueProgrammingCommand() to program ECC |
| Fapi_issueAsyncCommand() | Issues a command (Clear Status) to FSM for operations that do not require an address |
| Fapi_checkFsmForReady() | Returns whether or not the Flash state machine (FSM) is ready or busy |

| | |
|---|---|
| Fapi_getFsmStatus() | Returns the STATCMD status register value from the Flash Wrapper |

**Table 2-3. Summary of Read Functions**

| API Function | Description |
|---|---|
| Fapi_doBlankCheck() | Verifies specified Flash memory range against erased state |
| Fapi_doVerify() | Verifies specified Flash memory range against supplied values |

Note: Fapi_calculatePsa() and Fapi_doPsaVerify() are deprecated.

**Table 2-4. Summary of Information Functions**

| API Function | Description |
|---|---|
| Fapi_getLibraryInfo() | Returns the information specific to the compiled version of the API library |

**Table 2-5. Summary of Utility Functions**

| API Function | Description |
|---|---|
| Fapi_flushPipeline() | Flushes the data cache in Flash Wrapper |
| Fapi_calculateEcc() | Calculates the ECC for the supplied address and 64-bit word |
| Fapi_isAddressEcc() | Determines if the address falls in ECC ranges |
| Fapi_remapEccAddress() | Remaps an ECC address to corresponding main address |
| Fapi_calculateFletcherChecksum() | Function calculates a Fletcher checksum for the memory range specified |

## 2.3 Using API

This section describes the flow for using various API functions.

### 2.3.1 Initialization Flow

#### *2.3.1.1 After Device Power Up*

After the device is first powered up, the *Fapi_initializeAPI()* function must be called before any other API function (except for the *Fapi_getLibraryInfo()* function) can be used. This procedure configures the Flash Wrapper based on the user specified operating system frequency.

#### *2.3.1.2 Flash Wrapper and Bank Setup*

Before performing a Flash operation for the first time, the *Fapi_setActiveFlashBank()* function must be called.

#### *2.3.1.3 On System Frequency Change*

If the System operating frequency is changed after the initial call to the *Fapi_initializeAPI()* function, this function must be called again before any other API function (except the *Fapi_getLibraryInfo()* function) can be used. This procedure will update the API internal state variables.

### 2.3.2 Building With the API

#### *2.3.2.1 Object Library Files*

The Flash API object file is distributed in the Arm® standard EABI elf and COFF object formats.

**Note**

Compilation requires the "Enable support for GCC extensions" option to be enabled. Compiler version 6.4.0 and onwards have this option enabled by default.

#### 2.3.2.2 Distribution Files

The following API files are distributed in the C2000Ware\libraries\flash_api\f280013x\ folder:

- Library Files
  - TMS320F280013x Flash API is NOT embedded into the Boot ROM of this device, it is wholly software. The software libraries provided are in EABI elf (FAPI_F280013x_EABI_v2.00.01.lib) and COFF (FAPI_F280013x_COFF_v2.00.01.lib) object formats. In order for the application to be able to erase or program the Flash/OTP, one of these two library files should be included in the application, depending on the output object format the application is using.
- FAPI_F280013x_EABI_v2.00.01.lib – This is the Flash API EABI elf object format library (FPU32 flag enabled for build) for TMS320F280013x devices.
- FAPI_F280013x_COFF_v2.00.01.lib – This is the Flash API COFF object format library (FPU32 flag enabled for build) for TMS320F280013x devices.
- Fixed point version of the API library is not provided.
- Include Files
  - FlashTech_F280013x_C28x.h – The master include file for TMS320F280013x devices. This file sets up compile specific defines and then includes the FlashTech.h master include file.
  - hw_flash.h – Definitions of the flash read path configuration registers
  - hw_flash_command.h – Definitions of the flash write/erase protection registers
  - hw_flash_memmap.h – Definitions of the flash register address map
- The following include files should not be included directly by the user's code, but are listed here for user reference:
  - FlashTech.h – This include file lists all public API functions and includes all other include files.
  - Init.h – Defines the API initialization structure.
  - Registers.h – Definitions common to all register implementations and includes the appropriate register include file for the selected device type.
  - Types.h – Contains all the enumerations and structures used by the API.
  - Constants/Constants.h – Constant definitions common to some C2000™ devices.
  - Constants/F280013x.h – Constant definitions for F280013x devices.

### 2.3.3 Key Facts For Flash API Usage

Here are some important facts about API usage:

- Names of the Flash API functions start with a prefix "Fapi_".
- Flash API does not configure PLL. The user application should configure the PLL as needed and pass the configured CPUCLK value to Fapi_initializeAPI() function (details of this function are given later in this document). Note that the flash API library does not support flash erase/program operations when the system frequency is less than or equal to 20MHz.
- Flash API does not check the PLL configuration to confirm the user input frequency. This is up to the system integrator - TI suggests to use the DCC module to check the system frequency. For example implementation, see the C2000Ware driverlib clock configuration function.
- Always configure waitstates as per the device-specific data manual before calling the Flash API functions. The Flash API will issue an error if the waitstate configured by the application is not appropriate for the operating frequency of the application.
- Flash API execution is interruptible. However, there should not be any read/fetch access from the Flash bank on which an erase/program operation is in progress. Therefore, the Flash API functions, the user application functions that call the Flash API functions, and any ISRs (Interrupt service routines,) must be executed from RAM. For example, the above mentioned conditions apply to the entire code-snippet shown below in addition to the Flash API functions. The reason for this is because the Fapi_issueAsyncCommandWithAddress() function issues the erase command to the FSM, but it does not wait until the erase operation is over. As long as the FSM is busy with the current operation, the Flash bank being erased should not be accessed.

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

- Flash API does not configure (enable/disable) watchdog. The user application can configure watchdog and service it as needed. Hence, the Fapi_ServiceWatchdogTimer() function is no longer provided.
- Flash API uses EALLOW and EDIS internally as needed to allow/disallow writes to protected registers.
- The Main Array flash programming must be aligned to 64-bit address boundaries (alignment on 128-bit address boundary is suggested) and each 64-bit word may only be programmed once per write/erase cycle.
- It is permissible to program the data and ECC separately. However, each 64-bit dataword and the corresponding ECC word may only be programmed once per write/erase cycle.
- There is no pump semaphore in TMS320F280013x devices.
- ECC should not be programmed for link-pointer locations. The API skips programming the ECC when the start address provided for the program operation is any of the three link-pointer addresses. API will use Fapi_DataOnly mode for programming these locations even if the user passes Fapi_AutoEccGeneration or Fapi_DataAndEcc mode as the programming mode parameter. The Fapi_EccOnly mode is not supported for programming these locations. The user application should exercise caution here. Care should be taken to maintain a separate structure/section for link-pointer locations in the application. Do not mix these fields with other DCSM OTP settings. If other fields are mixed with link-pointers, API will skip programming ECC for the non-link-pointer locations as well. This will cause ECC errors in the application.
- In order to avoid conflict between zone1 and zone2, a semaphore (FLSEM) is provided in the DCSM registers to configure Flash registers. The user application should configure this semaphore register before initializing the Flash and calling the Flash API functions. For more details on this register, see the *TMS320F280013x Microcontrollers Technical Reference Manual*.
- Note that the Flash API functions do not configure any of the DCSM registers. The user application should be sure to configure the required DCSM settings. For example, if a zone is secured, then Flash API should be executed from the same zone in order to be able to erase or program the Flash sectors of that zone. Or the zone should be unlocked. If not, Flash API's writes to Flash registers will not succeed. Flash API does not check whether the writes to the Flash registers are going through or not. It writes to them as required for the erase/program sequence and returns back assuming that the writes went through. This will cause the Flash API to return false success status. For example, Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, Address) when called, can return the success status but it does not mean that the sector erase is successful. Erase status should be checked using Fapi_getFSMStatus() and Fapi_doBlankCheck().
- Note that there should not be any access to the Flash bank/OTP on which the Flash erase/program operation is in progress.

# 3 API Functions

## 3.1 Initialization Functions

### 3.1.1 Fapi_initializeAPI()

Initializes the Flash API

#### Synopsis

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

#### Parameters

| | |
|---|---|
| *poFlashControlRegister [in]* | Pointer to the Flash Wrapper Registers' base address. Use FlashTech_CPU0_BASE_ADDRESS. |
| *u32HclkFrequency [in]* | System clock frequency in MHz |

#### Description

This function is required to initialize the Flash API before any other Flash API operation is performed. This function must also be called if the System frequency or RWAIT is changed.

**Note**

RWAIT register value must be set before calling this function.

#### Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_InvalidHclkValue** (failure: System clock does not match specified wait value)

**Sample Implementation** (Please refer to the flash programming example provided in C2000Ware at “C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c”)

## 3.2 Flash State Machine Functions

### 3.2.1 Fapi_setActiveFlashBank()

Initializes the Flash Wrapper for erase and program operations.

#### Synopsis

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

#### Parameters

| | |
|---|---|
| *oNewFlashBank [in]* | Bank number to set as active. Fapi_FlashBank0 should be used for this device since there is only one bank. |

#### Description

This function sets the Flash Wrapper for further operations to be performed on the bank. This function is required to be called after the *Fapi_initializeAPI()* function and before any other Flash API operation is performed.

**Note**

Application needs to call this only once and that can be with Fapi_FlashBank0.

---

**Return Value**

- **Fapi_Status_Success** (Success)
- **Fapi_Status_FsmBusy** (failure: FSM busy with another command)
- **Fapi_Error_InvalidBaseRegCntlAddress** (failure: Flash control register base address provided by user does not match the expected address)
- **Fapi_Error_InvalidBank** (failure: Bank specified does not exist on device)
- **Fapi_Error_InvalidHclkValue** (failure: System clock does not match specified wait value)

**Sample Implementation** (Please refer to the flash programming example provided in C2000Ware at "C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c")

### 3.2.2 Fapi_setupBankSectorEnable()

Configures Write(program)/Erase protection for the sectors.

**Synopsis**

```
Fapi_StatusType Fapi_setupBankSectorEnable(
                    uint32 WEPROT_register,
                     uint32 oSectorMask
                                        )
```

**Parameters**

| | |
|---|---|
| *pu32StartAddress [in]* | Register address for Write/Erase protection configuration.<br>Use<br>*FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA*<br>for the first 32 (0-31) sectors.<br>Use<br>*FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB*<br>for the remaining main-array (32-127) sectors.<br>Use<br>*FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROT_UO*<br>for the USER OTP. |
| *OSectorMask [in]* | 32-bit mask indicating which sectors to mask from the erase and program operations. |

**Description**

On this device, all of the flash main-array sectors and the USER OTP are protected from the erase and program operations by default. User application has to disable the protection for the sectors on which it wants to perform erase and/or program operations. This function can be used to enable/disable the protection. This function should be called before each erase and program command as shown in the flash API usage example provided in the C2000Ware.

First input parameter for this function can be the address of any of these three registers: *CMDWEPROTA, CMDWEPROTB, CMDWEPROT_UO*

*CMDWEPROTA* register is used to configure the protection for the first 32 sectors (0 to 31). Each bit in this register corresponds to each sector – Example: Bit 0 of this register is used to configure the protection for Sector 0 and Bit 31 of this register is used to configure the protection for Sector 31. A 32-bit user-provided sector mask (second parameter passed to this function) indicates which sectors the user wants to mask from the erase and program operations, that is, sectors that will not be erased and programmed. If a bit in the mask is 1, that particular sector will not be erased/programmed. If a bit in the mask is 0, that particular sector will be erased/programmed.

*CMDWEPROTB* register is used to configure the protection for the 32 – 127 sectors in the main-array flash bank. However, please note that each bit in this register is used to configure protection for 8 sectors together. This means, bit 0 is used to configure the protection for all of the sectors 32 to 39 together, bit 1 is used to configure the protection for all of the sectors 40 to 47 together, and so on. A 32-bit user-provided sector mask (second parameter passed to this function) indicates which sectors the user wants to mask from the erase and program operations, that is, sectors that will not be erased and programmed. If a bit in the mask is 1, that particular set of sectors will not be erased/programmed. If a bit in the mask is 0, that particular set of sectors will be erased/programmed.

*CMDWEPROT_UO* register is used to configure the protection for the USER OTP. Bit 0 in this register is used to configure the protection for the USER OTP. This means, if bit 0 is configured as 1, USER OTP will not be programmed. If bit 0 is configured as 0, USER OTP will be programmed. Other bits of this register can be configured as 1s. Since USER OTP is not erasable, the *CMDWEPROT_UO* register protection is not applicable for erase operations. This should be configured only for the program operation as needed.

**Return Value**

- **Fapi_Status_Success** (success)

**Sample Implementation** (Please refer to the flash programming example provided in C2000Ware at "C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c")

### 3.2.3 Fapi_issueAsyncCommandWithAddress()

Issues an erase command to the Flash State Machine along with a user-provided sector address.

**Synopsis**

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

**Parameters**

| | |
|---|---|
| *oCommand [in]* | Command to issue to the FSM. Use Fapi_EraseSector |
| *pu32StartAddress [in]* | Flash sector address for erase operation |

**Description**

This function issues an erase command to the Flash State Machine for the user-provided sector address. This function does not wait until the erase operation is over; it just issues the command and returns back. Hence, this function always returns success status when the Fapi_EraseSector command is used. The user application must wait for the Flash Wrapper to complete the erase operation before returning to any kind of Flash accesses. The Fapi_checkFsmForReady() function can be used to monitor the status of an issued command.

**Note**

This function does not check STATCMD after issuing the erase command. The user application must check the STATCMD value when FSM has completed the erase operation. STATCMD indicates if there is any failure occurrence during the erase operation. The user application can use the Fapi_getFSMStatus function to obtain the STATCMD value.

Also, the user application should use the Fapi_doBlankCheck() function to verify that the Flash is erased.

**Return Value**

- **Fapi_Status_Success** (success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_InvalidBaseRegCntlAddress** (failure: Flash control register base address provided by user does not match the expected address)
- **Fapi_Error_FeatureNotAvailable** (failure: User requested a command that is not supported).
- **Fapi_Error_FlashRegsNotWritable** (failure: Flash register write failed. The user should make sure that the API is executing from the same zone as that of the target address for flash operation OR the user should unlock before the flash operation).
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. For the valid address range), see the *TMS320F280013x Microcontrollers Data Manual*.

**Sample Implementation** (Please refer to the flash programming example provided in C2000Ware at "C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c")

### 3.2.4 Fapi_issueBankEraseCommand()

Issues a bank erase command to the Flash State Machine along with a user-provided sector mask.

**Synopsis**

```
Fapi_StatusType
Fapi_issueBankEraseCommand(
      uint32 *pu32StartAddress
                          )
```

**Parameters**

| | |
|---|---|
| *pu32StartAddress [in]* | Flash bank address for bank erase operation |

**Description**

This function issues a bank erase command to the Flash state machine for the user-provided bank address. If the FSM is busy with another operation, the function returns indicating the FSM is busy, otherwise it proceeds with the bank erase operation.

**Return Value**

- **Fapi_Status_Success** (success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_FlashRegsNotWritable** (Flash registers not writable)
- **Fapi_Error_InvalidBaseRegCntlAddress** (failure: Flash control register base address provided by user does not match the expected address)

**Sample Implementation** (Please refer to the flash programming example provided in C2000Ware at "C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c")

### 3.2.5 Fapi_issueProgrammingCommand()

Sets up data and issues program command to valid Flash or OTP memory addresses

**Synopsis**

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

### Parameters

| | |
|---|---|
| *pu32StartAddress [in]* | Start address in Flash for the data and ECC to be programmed. Also, the start address should always be even. |
| *pu16DataBuffer [in]* | Pointer to the Data buffer address. Data buffer should be 128-bit aligned. |
| *u16DataBufferSizeInWords [in]* | Number of 16-bit words in the Data buffer |
| *pu16EccBuffer [in]* | Pointer to the ECC buffer address |
| *u16EccBufferSizeInBytes [in]* | Number of 8-bit bytes in the ECC buffer |
| *oMode [in]* | Indicates the programming mode to use: |
| | Fapi_DataOnly — Programs only the data buffer |
| | Fapi_AutoEccGeneration — Programs the data buffer and auto generates and programs the ECC. |
| | Fapi_DataAndEcc — Programs both the data and ECC buffers |
| | Fapi_EccOnly — Programs only the ECC buffer |

**Note**

The pu16EccBuffer should contain ECC corresponding to the data at the 128-bit aligned main array/OTP address. The LSB of the pu16EccBuffer corresponds to the lower 64 bits of the main array and the MSB of the pu16EccBuffer corresponds to the upper 64 bits of the main array.

### Description

This function sets up the programming registers of the Flash State Machine based on the supplied parameters. It offers four different programming modes to the user for use in different scenarios as mentioned in Table 3-1.

**Table 3-1. Uses of Different Programming Modes**

| Programming Mode (oMode) | Arguments Used | Usage Purpose |
|---|---|---|
| Fapi_DataOnly | pu32StartAddress, pu16DataBuffer, u16DataBufferSizeInWords | Used when any custom programming utility or an user application (that embed/use Flash API) has to program data and corresponding ECC separately. Data is programmed using Fapi_DataOnly mode and then the ECC is programmed using Fapi_EccOnly mode. Generally most of the programming utilities do not calculate ECC separately and instead use Fapi_AutoEccGeneration mode. However, some Safety applications may require to insert intentional ECC errors in their Flash image (which is not possible when Fapi_AutoEccGeneration mode is used) to check the health of the SECDED (Single Error Correction and Double Error Detection) module at run time. In such case, ECC is calculated separately (using the Fapi_calculateEcc() function as applicable). Application may want to insert errors in either main array data or in the ECC as needed. In such scenarios, after the error insertion, Fapi_DataOnly mode and Fapi_EccOnly modes can be used to program the data and ECC respectively. |
| Fapi_AutoEccGeneration | pu32StartAddress, pu16DataBuffer, u16DataBufferSizeInWords | Used when any custom programming utility or user application (that embed/use Flash API to program Flash at run time to store data or to do a firmware update) has to program data and ECC together without inserting any intentional errors. This is the most prominently used mode. |

**Table 3-1. Uses of Different Programming Modes (continued)**

| Programming Mode (oMode) | Arguments Used | Usage Purpose |
|---|---|---|

| | | |
|---|---|---|
| Fapi_DataAndEcc | pu32StartAddress,<br>pu16DataBuffer,<br>u16DataBufferSizeInWords,<br>pu16EccBuffer,<br>u16EccBufferSizeInBytes | Purpose of this mode is not different than that of using Fapi_DataOnly and Fapi_EccOnly modes together. However, this mode is beneficial when both the data and the calculated ECC can be programmed at the same time. |
| Fapi_EccOnly | pu16EccBuffer,<br>u16EccBufferSizeInBytes | See the usage purpose given for Fapi_DataOnly mode. |

**Note**

Users must always program ECC for their flash image since ECC check is enabled at power up.

**Programming modes:**

**Fapi_DataOnly** – This mode will only program the data portion in Flash at the address specified. It can program from 1-bit up to 8 16-bit words. However, review the restrictions provided for this function to know the limitations of flash programming data size. The supplied starting address to program at plus the data buffer length cannot cross the 128-bit aligned address boundary. Arguments 4 and 5 are ignored when using this mode.

**Fapi_AutoEccGeneration** – This mode will program the supplied data in Flash along with automatically generated ECC. The ECC is calculated for every 64-bit data aligned on a 64-bit memory boundary. Hence, when using this mode, all the 64 bits of the data should be programmed at the same time for a given 64-bit aligned memory address. Data not supplied is treated as all 1s (0xFFFF). Once ECC is calculated and programmed for a 64-bit data, those 64 bits can not be reprogrammed (unless the sector is erased) even if it is programming a bit from 1 to 0 in that 64-bit data, since the new ECC value will collide with the previously programmed ECC value. When using this mode, if the start address is 128-bit aligned, then either 8 or 4 16-bit words can be programmed at the same time as needed. If the start address is 64-bit aligned but not 128-bit aligned, then only 4 16-bit words can be programmed at the same time. The data restrictions for Fapi_DataOnly also exist for this option. Arguments 4 and 5 are ignored

**Note**

Fapi_AutoEccGeneration mode will program the supplied data portion in Flash along with automatically generated ECC. The ECC is calculated for 64-bit aligned address and the corresponding 64-bit data. Any data not supplied is treated as 0xFFFF. Note that there are practical implications of this when writing a custom programming utility that streams in the output file of a code project and programs the individual sections one at a time into flash. If a 64-bit word spans more than one section (that is, contains the end of one section, and the start of another), values of 0xFFFF cannot be assumed for the missing data in the 64-bit word when programming the first section. When you go to program the second section, you will not be able to program the ECC for the first 64-bit word since it was already (incorrectly) computed and programmed using assumed 0xFFFF for the missing values. One way to avoid this problem is to align all sections linked to flash on a 64-bit boundary in the linker command file for your code project.

Here is an example:

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

If you do not align the sections in flash, you would need to track incomplete 64-bit words in a section and combine them with the words in other sections that complete the 64-bit word. This will be difficult to do. So it is recommended to align your sections on 64-bit boundaries.

Some 3rd party Flash programming tools or TI Flash programming kernel examples (*C2000Ware*) or any custom Flash programming solution may assume that the incoming data stream is all 128-bit aligned and may not expect that a section might start on an unaligned address. Thus it may try to program the maximum possible (128-bits) words at a time assuming that the address provided is 128-bit aligned. This can result in a failure when the address is not aligned. So, it is suggested to align all the sections (mapped to Flash) on a 128-bit boundary.

**Fapi_DataAndEcc** – This mode will program both the supplied data and ECC in Flash at the address specified. The data supplied must be aligned on a 64-bit memory boundary and the length of data must correlate to the supplied ECC. That means, if the data buffer length is 4 16-bit words, the ECC buffer must be 1 byte. If the data buffer length is 8 16-bit words, the ECC buffer must be 2 bytes in length. If the start address is 128-bit aligned, then either 8 or 4 16-bit words should be programmed at the same time as needed. If the start address is 64-bit aligned but not 128-bit aligned, then only 4 16-bit words should be programmed at the same time.

The LSB of pu16EccBuffer corresponds to the lower 64-bits of the main array and the MSB of pu16EccBuffer corresponds to the upper 64-bits of the main array.

The Fapi_calculateEcc() function can be used to calculate ECC for a given 64-bit aligned address and the corresponding data.

**Fapi_EccOnly** – This mode will only program the ECC portion in Flash ECC memory space at the address (Flash main array address should be provided for this function and not the corresponding ECC address) specified. It can program either 2 bytes (both LSB and MSB at a location in ECC memory) or 1 byte (LSB at a location in ECC memory).

The LSB of pu16EccBuffer corresponds to the lower 64-bits of the main array and the MSB of pu16EccBuffer corresponds to the upper 64-bits of the main array.

Arguments two and three are ignored when using this mode.

**Note**

The length of pu16DataBuffer and pu16EccBuffer cannot exceed 8 and 2, respectively.

**Note**

This function does not check STATCMD after issuing the program command. The user application must check the STATCMD value when FSM has completed the program operation. STATCMD indicates if there is any failure occurrence during the program operation. The user application can use the Fapi_getFsmStatus function to obtain the STATCMD value.

Also, the user application should use the Fapi_doVerify() function to verify that the Flash is programmed correctly.

This function does not wait until the program operation is over; it just issues the command and returns back. Hence, the user application must wait for the Flash Wrapper to complete the program operation before returning to any kind of Flash accesses. The Fapi_checkFsmForReady() function should be used to monitor the status of an issued command.

**Restrictions**

- As described above, this function can program only a max of 128-bits (given the address provided is 128-bit aligned) at a time. If the user wants to program more than that, this function should be called in a loop to program 128-bits (or 64-bits as needed by application) at a time.

- The Main Array flash programming must be aligned to 64-bit address boundaries and each 64-bit word may only be programmed once per write or erase cycle.
- It is alright to program the data and ECC separately. However, each 64-bit dataword and the corresponding ECC word may only be programmed once per write or erase cycle.
- ECC should not be programmed for linkpointer locations. The API will issue the Fapi_DataOnly command for these locations even if the user chooses Fapi_AutoEccGeneration mode or Fapi_DataAndEcc mode. Fapi_EccOnly mode is not supported for linkpointer locations.
- Fapi_EccOnly mode should not be used for Bank0 DCSM OTP space. If used, an error will be returned. For the DCSM OTP space, either Fapi_AutoEccGeneration or Fapi_DataAndEcc programming modes should be used.

**Return Value**

- **Fapi_Status_Success** (success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_InvalidBaseRegCntlAddress** (failure: Flash control register base address provided by user does not match the expected address)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (failure: Data buffer size specified is incorrect. Also, this error will be returned if Fapi_EccOnly mode is selected when programming the Bank0 DCSM OTP space)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: ECC buffer size specified is incorrect)
- **Fapi_Error_AsyncDataEccBufferLengthMismatch** (failure: Data buffer size either is not 64-bit aligned or data length crosses the 128-bit aligned memory boundary)
- **Fapi_Error_FlashRegsNotWritable** (failure: Flash register writes failed. The user should make sure that the API is executing from the same zone as that of the target address for flash operation OR the user should unlock before the flash operation.
- **Fapi_Error_FeatureNotAvailable** (failure: User passed a mode that is not supported)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. For the valid address range, see the *TMS320F280013x Microcontrollers Data Manual*.

**Sample Implementation** (Please refer to the flash programming example provided in C2000Ware at "C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c")

### 3.2.6 Fapi_issueProgrammingCommandForEccAddresses()

Remaps an ECC address to data address and calls Fapi_issueProgrammingCommand().

**Synopsis**

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

**Parameters**

| | |
|---|---|
| *pu32StartAddress [in]* | ECC start address in Flash for the ECC to be programmed |
| *pu16EccBuffer [in]* | pointer to the ECC buffer address |
| *u16EccBufferSizeInBytes [in]* | number of bytes in the ECC buffer<br>If the number of bytes is 1, LSB (ECC for lower 64 bits) gets programmed. MSB alone cannot be programmed using this function. If the number of bytes is 2, both LSB and MSB bytes of ECC get programmed. |

**Description**

This function will remap an address in the ECC memory space to the corresponding data address space and then call Fapi_issueProgrammingCommand() to program the supplied ECC data. The same limitations for Fapi_issueProgrammingCommand() using Fapi_EccOnly mode applies to this function. The LSB of pu16EccBuffer corresponds to the lower 64 bits of the main array and the MSB of pu16EccBuffer corresponds to the upper 64 bits of the main array.

**Note**

The length of the pu16EccBuffer cannot exceed 2.

**Note**

This function does not check STATCMD after issuing the program command. The user application must check the STATCMD value when FSM has completed the program operation. STATCMD indicates if there is any failure occurrence during the program operation. The user application can use the Fapi_getFSMStatus function to obtain the STATCMD value.

**Note**

Fapi_EccOnly mode should not be used for Bank0 DCSM OTP space. If used, an error will be returned. For the DCSM OTP space, either Fapi_AutoEccGeneration or Fapi_DataAndEcc programming modes should be used.

**Return Value**

- **Fapi_Status_Success** (success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_InvalidBaseRegCntlAddress** (failure: Flash control register base address provided by user does not match the expected address)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: Data buffer size specified is incorrect)
- **Fapi_Error_FlashRegsNotWritable** (failure: Flash register writes failed. The user should make sure that the API is executing from the same zone as that of the target address for flash operation OR the user should unlock before the flash operation.
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. For the valid address range, see the *TMS320F280013x Microcontrollers Data Manual*.

### 3.2.7 Fapi_issueAsyncCommand()

Issues a command to the Flash State Machine. See the description for the list of commands that can be issued by this function.

**Synopsis**

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

**Parameters**

| | |
|---|---|
| *oCommand [in]* | Command to issue to the FSM.<br>Use Fapi_ClearStatus command. |

**Description**

This function issues a command to the Flash State Machine for commands not requiring any additional information (such as address). On this device, Fapi_ClearStatus command should be issued to the Flash State Machine using this function. Note that Fapi_ClearStatus command should be issued before each program and erase command as shown in the flash programming example provided in C2000Ware. A new program or erase command can be given only when the STATCMD is zero (achieved by issuing the Fapi_ClearStatus command).

**Return Value**

- **Fapi_Status_Success** (success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_FeatureNotAvailable** (failure: User passed a command that is not supported)

**Sample Implementation** (Please refer to the flash programming example provided in C2000Ware at "C2000Ware_.....\driverlib\f280013x\examples\flash\flashapi_ex1_programming.c")

### 3.2.8 Fapi_checkFsmForReady()

Returns the status of the Flash State Machine

**Synopsis**

```
Fapi_StatusType Fapi_checkFsmForReady(void)
```

**Parameters**

None

**Description**

This function returns the status of the Flash State Machine indicating if it is ready to accept a new command or not. The primary use is to check if an Erase or Program operation has finished.

**Return Value**

- **Fapi_Status_FsmBusy** (FSM is busy and cannot accept new command except for suspend commands)
- **Fapi_Status_FsmReady** (FSM is ready to accept new command)

### 3.2.9 Fapi_getFsmStatus()

Returns the value of the STATCMD register

**Synopsis**

```
Fapi_FlashStatusType Fapi_getFsmStatus(void)
```

**Parameters**

None

**Description**

This function returns the value of the STATCMD register. This register allows the user application to determine whether an erase or program operation is successfully completed or in progress or suspended or failed. The user application should check the value of this register to determine if there is any failure after each erase and program operation.

### Return Value

**Table 3-2. STATCMD Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | FAILMISC | | | | FAILINVDATA | | FAILILLADDR | FAILVERIFY | FAILWEPROT | | CMDINPROGRESS | CMDPASS | CMDDONE |
| | | | RO - 0x0 | | | | RO - 0x0 | | RO - 0x0 | RO - 0x0 | RO - 0x0 | | RO - 0x0 | RO - 0x0 | RO - 0x0 |

RO – Read Only

**Table 3-3. STATCMD Register Field Descriptions**

| Bit | Name | Description | Reset value |
|---|---|---|---|
| 12 | FAILMISC | Command failed due to error other than write/erase protect violation or verify error.<br>0: No Fail<br>1: Fail | 0x0 |
| 8 | FAILINVDATA | Program command failed because an attempt was made to program a stored 0 value to a 1.<br>0: No Fail<br>1: Fail | 0x0 |
| 6 | FAILILLADDR | Command failed due to the use of an illegal address.<br>0: No Fail<br>1: Fail | 0x0 |
| 5 | FAILVERIFY | Command failed due to verify error.<br>0: No Fail<br>1: Fail | 0x0 |
| 4 | FAILWEPROT | Command failed due to Write/Erase Protect Sector violation.<br>0: No Fail<br>1: Fail | 0x0 |
| 2 | CMDINPROGRESS | Command in Progress<br>0: Command complete<br>1: Command is in progress | 0x0 |
| 1 | CMDPASS | Command Pass - valid when CMD_DONE field is 1<br>0: Fail<br>1: Pass | 0x0 |
| 0 | CMDDONE | Command Done<br>0: Command not Done<br>1: Command Done | 0x0 |

## 3.3 Read Functions

### 3.3.1 Fapi_doBlankCheck()

Verifies region specified is erased value

#### Synopsis

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

#### Parameters

| | | |
|---|---|---|
| *pu32StartAddress [in]* | | Start address for region to blank check |
| *u32Length [in]* | | Length of region in 32-bit words to blank check |
| *poFlashStatusWord [out]* | | Returns the status of the operation if result is not Fapi_Status_Success |
| | ->au32StatusWord[0] | Address of first non-blank location |
| | ->au32StatusWord[1] | Data read at first non-blank location |
| | ->au32StatusWord[2] | Value of compare data (always 0xFFFFFFFF) |
| | ->au32StatusWord[3] | N/A |

#### Description

This function checks if the flash is blank (erased state) starting at the specified address for the length of 32-bit words specified. If a non-blank location is found, corresponding address and data will be returned in the poFlashStatusWord parameter.

#### Restrictions

None

#### Return Value

- **Fapi_Status_Success (success)** - specified Flash locations are found to be in erased state
- **Fapi_Error_Fail** (failure: region specified is not blank)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. For the valid address range), see the *TMS320F280013x Microcontrollers Data Manual*.

### 3.3.2 Fapi_doVerify()

Verifies region specified against supplied data

#### Synopsis

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

#### Parameters

| | | |
|---|---|---|
| *pu32StartAddress [in]* | | start address for region to verify |
| *u32Length [in]* | | length of region in 32-bit words to verify |
| *pu32CheckValueBuffer [in]* | | address of buffer to verify region against. Data buffer should be 128-bit aligned. |
| *poFlashStatusWord [out]* | | returns the status of the operation if result is not Fapi_Status_Success |
| | ->au32StatusWord[0] | address of first verify failure location |
| | ->au32StatusWord[1] | data read at first verify failure location |
| | ->au32StatusWord[2] | value of compare data |
| | ->au32StatusWord[3] | N/A |

### Description

This function verifies the device against the supplied data starting at the specified address for the length of 32-bit words specified. If a location fails to compare, these results will be returned in the poFlashStatusWord parameter.

### Restrictions

None

### Return Value

- **Fapi_Status_Success** (success: region specified matches supplied data) )
- **Fapi_Error_Fail** (failure: region specified does not match supplied data)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. For the valid address range, see the *TMS320F280013x Microcontrollers Data Manual*.

## 3.4 Informational Functions

### 3.4.1 Fapi_getLibraryInfo()

Returns information about this compile of the Flash API

#### Synopsis

```
Fapi_LibraryInfoType Fapi_getLibraryInfo(void)
```

#### Parameters

*None*

#### Description

This function returns information specific to the compile of the Flash API library. The information is returned in a struct Fapi_LibraryInfoType. The members are as follows:

- u8ApiMajorVersion – Major version number of this compile of the API. This value is 2.
- u8ApiMinorVersion – Minor version number of this compile of the API. Minor version is 00 for F280013x devices.
- u8ApiRevision – Revision version number of this compile of the API.

  **Revision number is 01 for this release.**

  **Note**: The beta flash API libraries (FAPI_F280013x_EABI_v2.00.00.lib and FAPI_F280013x_COFF_v2.00.00.lib) should not be used by the users for their production application. Only the FAPI_F280013x_EABI_v2.00.01.lib and FAPI_F280013x_COFF_v2.00.01.lib libraries should be used for the production application.

- oApiProductionStatus – Production status of this compile *(Alpha_Internal, Alpha, Beta_Internal, Beta, Production).*

  **Production status is Production for this release.**

- u32ApiBuildNumber – Build number of this compile.
- u8ApiTechnologyType – Indicates the Flash technology supported by the API. This field returns a value of 0x5.
- u8ApiTechnologyRevision – Indicates the revision of the technology supported by the API
- u8ApiEndianness – This field always returns as 1 (Little Endian) for F280013x devices.
- u32ApiCompilerVersion – Version number of the Code Composer Studio code generation tools used to compile the API

#### Return Value

- **Fapi_LibraryInfoType** (gives the information retrieved about this compile of the API)

## 3.5 Utility Functions

### 3.5.1 Fapi_flushPipeline()

Flushes the Flash Wrapper pipeline buffers

#### Synopsis

```
void Fapi_flushPipeline(void)
```

#### Parameters

None

### Description

This function flushes the Flash Wrapper data cache. The data cache must be flushed before the first non-API Flash read after an erase or program operation.

### Return Value

None

## 3.5.2 Fapi_calculateEcc()

Calculates the ECC for the supplied address and 64-bit value

### Synopsis

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

### Parameters

| | |
|---|---|
| *u32Address [in]* | Address of the 64-bit value to calculate the ECC |
| *u64Data [in]* | 64-bit value to calculate ECC on (should be in little endian order) |

### Description

This function will calculate the ECC for a 64-bit aligned word including address. There is no need to provide a left-shifted address to this function anymore. TMS320F280013x Flash API takes care of it.

### Return Value

- 8-bit calculated ECC (upper 8 bits of the 16-bit return value should be ignored)
- If an error occurs, the 16-bit return value is 0xDEAD

## 3.5.3 Fapi_isAddressEcc()

Indicates is an address is in the Flash Wrapper ECC space

### Synopsis

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

### Parameters

| | |
|---|---|
| *u32Address [in]* | Address to determine if it lies in ECC address space |

### Description

This function returns True if address is in ECC address space or False if it is not.

### Return Value

- **FALSE** (Address is not in ECC address space)
- **TRUE** (Address is in ECC address space)

## 3.5.4 Fapi_remapEccAddress()

Takes ECC address and remaps it to main address space

### Synopsis

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

### Parameters

| | |
|---|---|
| *u32EccAddress [in]* | ECC address to remap |

### Description

This function returns the main array Flash address for the given Flash ECC address. When the user wants to program ECC data at a known ECC address, this function can be used to obtain the corresponding main array address. Note that the Fapi_issueProgrammingCommand() function needs a main array address and not the ECC address (even for the Fapi_EccOnly mode).

### Return Value

- **32-bit Main Flash Address**

### 3.5.5 Fapi_calculateFletcherChecksum()

Calculates the Fletcher checksum from the given address and length.

### Synopsis

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

### Parameters

| | |
|---|---|
| *pu16Data [in]* | Address to start calculating the checksum from |
| *u16Length [in]* | Number of 16-bit words to use in calculation |

### Description

This function generates a 32-bit Fletcher checksum starting at the supplied address for the number of 16-bit words specified. Note that only the flash main-array address range can be used for this function. DCSM OTP address range should not be provided.

### Return Value

- 32-bit Fletcher Checksum value

# 4 Recommended FSM Flows

## 4.1 New Devices From Factory

Devices are shipped erased from the factory. It is recommended, but not required, to do a blank check on devices received to verify that they are erased.

## 4.2 Recommended Erase Flow

Figure 4-1 describes the flow for erasing a sector(s) on a device. For further information, see Section 3.2.7 , 3.2.2 , 3.2.3 .

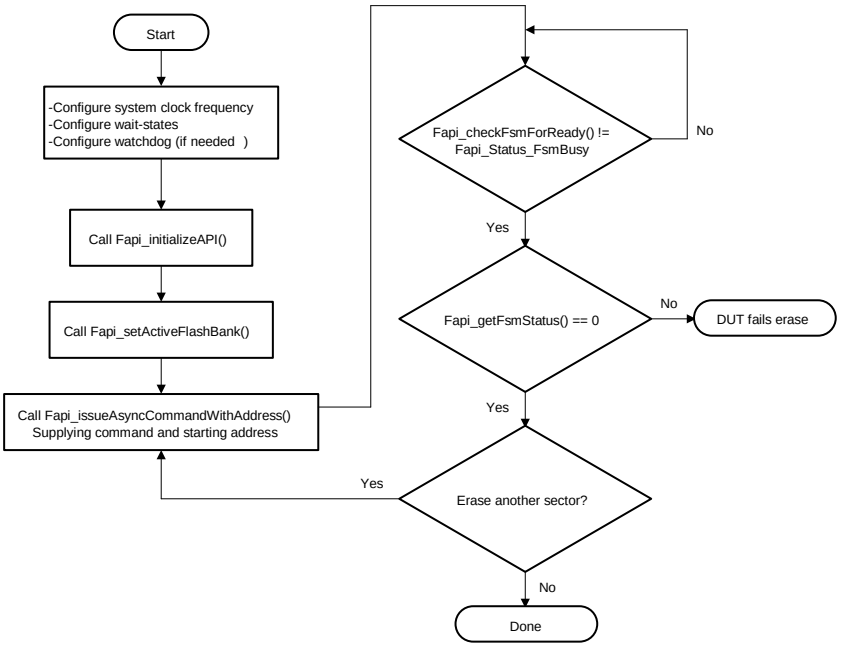


**Figure 4-1. Recommended Erase Flow**

Note: Before issuing the sector erase command, Fapi_ClearStatus command should be issued and the Write/Erase protections should be configured for the sectors as needed by the application.

## 4.3 Recommended Bank Erase Flow

Figure 4-2 describes the flow for erasing a Flash bank. For further information, see Section 3.2.7 , 3.2.2 , 3.2.4 .

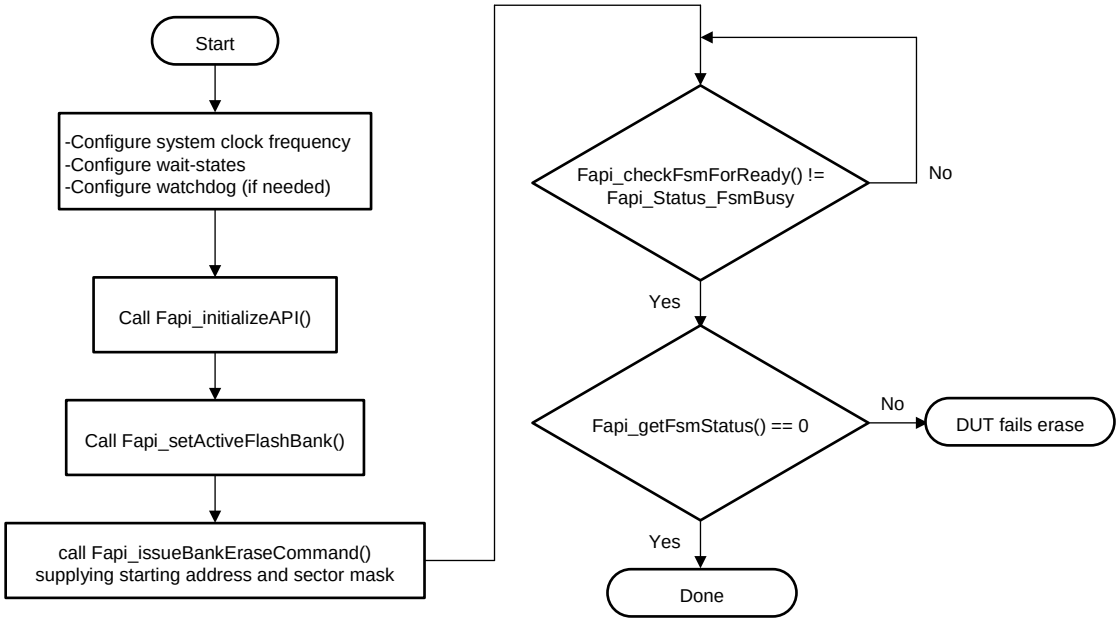


**Figure 4-2. Recommended Bank Erase Program Flow**

Note: Before issuing the bank erase command, Fapi_ClearStatus command should be issued and the Write/Erase protections should be configured for the sectors as needed by the application.

## 4.4 Recommended Program Flow

Figure 4-3 describes the flow for programming a device. This flow assumes the user has already erased all affected sectors or bank following the Recommended Erase Flow. For further information, see Section 3.2.7 , 3.2.2 , 3.2.5 .

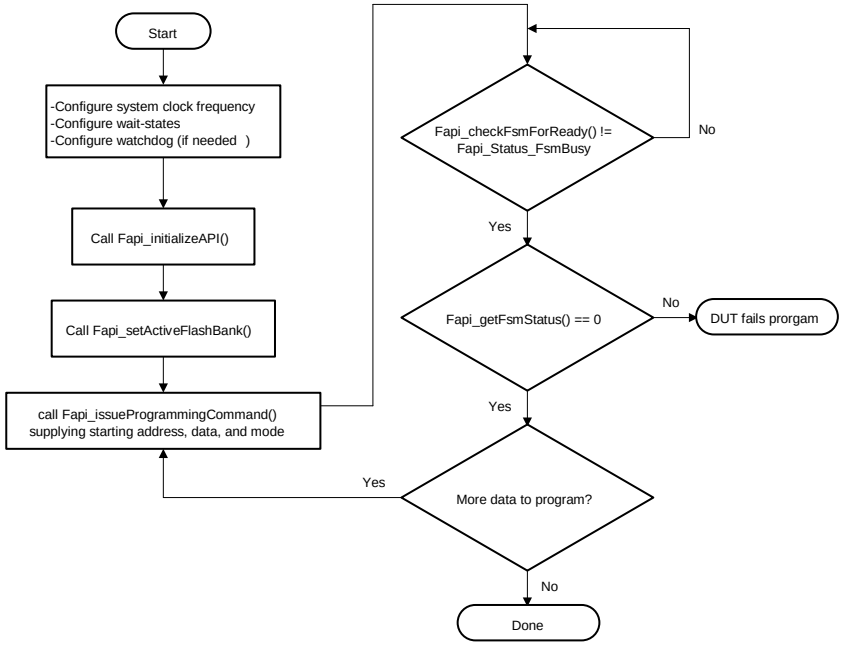

**Figure 4-3. Recommended Program Flow**

Note: Before issuing the program command, Fapi_ClearStatus command should be issued and the Write/Erase protections should be configured for the sectors as needed by the application.

# A Flash State Machine Commands

## A.1 Flash State Machine Commands

**Table A-1. Flash State Machine Commands**

| Command | Description | Enumeration Type | API Call(s) |
|---|---|---|---|
| Program Data | Used to program data to any valid Flash address | Fapi_ProgramData | Fapi_issueProgrammingCommand()<br>Fapi_issueProgrammingCommandForEccAddresses() |
| Erase Sector | Used to erase a Flash sector located by the specified address | Fapi_EraseSector | Fapi_issueAsyncCommandWithAddress() |
| Erase Bank | Used to erase a Flash bank | Fapi_EraseBank | Fapi_issueBankEraseCommand() |
| Clear Status | Clears the status register | Fapi_ClearStatus | Fapi_issueAsyncCommand() |

# B Typedefs, Defines, Enumerations and Structures

## B.1 Type Definitions

```
#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
```

## B.2 Defines

```
#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
```

## B.3 Enumerations

### B.3.1 Fapi_FlashProgrammingCommandsType

This contains all the possible modes used in the Fapi_IssueProgrammingCommand().

```
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;
```

### B.3.2 Fapi_FlashBankType

This is used to indicate which Flash bank is being used.

```
typedef enum
{
   Fapi_FlashBank0
}  ATTRIBUTE_PACKED Fapi_FlashBankType;
```

### B.3.3 Fapi_FlashStateCommandsType

This contains all the possible Flash State Machine commands.

```
typedef enum
{
   Fapi_ProgramData    = 0x0002,
   Fapi_EraseSector    = 0x0006,
   Fapi_EraseBank      = 0x0008,
   Fapi_ClearStatus    = 0x0010,
}  ATTRIBUTE_PACKED Fapi_FlashStateCommandsType;
```

### B.3.4 Fapi_StatusType

This is the master type containing all possible returned status codes.

```
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,  /* Flash Wrapper 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 */
}  ATTRIBUTE_PACKED
Fapi_StatusType;
```

### B.3.5 Fapi_ApiProductionStatusType

This lists the different production status values possible for the API.

```
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;
```



## B.4 Structures

### B.4.1 Fapi_FlashStatusWordType

This structure is used to return status values in functions that need more flexibility

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

### B.4.2 Fapi_LibraryInfoType

This is the structure used to return API information:

```
typedef struct
{
   uint8  u8ApiMajorVersion;
   uint8  u8ApiMinorVersion;
   uint8  u8ApiRevision;
   Fapi_ApiProductionStatusType oApiProductionStatus;
```

```
    uint32 u32ApiBuildNumber;
    uint8  u8ApiTechnologyType;
    uint8  u8ApiTechnologyRevision;
    uint8  u8ApiEndianness;
    uint32 u32ApiCompilerVersion;
} Fapi_LibraryInfoType;
```

## Revision History

February 2023

- Added a note in section 2.3.3 to mention that the flash API library should not be used when the system frequency is less than or equal to 10MHz
- Updated the description of the 3.2.2 Fapi_setupBankSectorEnable() to mention that CMDWEPROT_UO register should configured only for program operation
- Clarified that the flash API version 2.00.01 should be used for the user production application

June 2023

- Added a note in section 2.3.3 to mention that the flash API library should not be used when the system frequency is less than or equal to 20MHz
- Updated section B.3 Enumerations to show only the enumerations that are applicable for this device