User's Guide

TMS320F28P551x Flash API Version 6.00.00.00

ABSTRACT

Flash is an electrically erasable and programmable non-volatile memory that can be programmed and erased many times to ease code development. Flash memory can be used primarily as a program memory for the core, and secondarily as static data memory. This guide describes the usage of the flash API library to perform erase, program, and verify operations for the on-chip flash memory of TMS320F28P551x devices.

Table of Contents

List of Figures

List of Tables

Trademarks

Code Composer Studio™ 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 user's guide provides a detailed description of the Texas Instruments' TMS320F28P551x flash application programming interface (API) library (FAPI_F28P551x_EABI_v6.00.00.00 and FAPI_F28P551x_COFF_v6.00.00.00) functions that can be used to erase, program, and verify flash on TMS320F28P551x devices. Even though the library names for TMS320F28P551x are different, the flash API libraries are the same.

Note

Only use the flash API V6.00.XX.XX with TMS320F28P551x devices. The flash API library for TMS320F28P551x is provided in C2000Ware at C2000Ware_x_xx_xx_xx\libraries\flash_api\f28p551x.

1.1 Reference Material

Use this guide in conjunction with *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* and *TMS320F28P551/2/8 Real-Time Microcontrollers Technical Reference Manual* for the TMS320F28P551x device.

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 the function without storing any changes.
- out — Indicates the function saves one or more of the values in the parameter that you give the function. 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 the function 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 potentially apply:

- The function blocks or potentially block the requested operation under certain conditions.
- The function has pre-conditions that are potentially not obvious.
- The 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 can 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 TMS320F28P551x 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for the TMS320F28P551x flash and one-time programmable (OTP) memory map and flash waitstate specifications. This user's guide assumes that the user has already read the *Flash and OTP Memory* chapter in the *TMS320F28P551/2/8 Real-Time Microcontrollers Technical Reference Manual* for TMS320F28P551x.

Also, pay special attention to the functions Fapi_issueAsyncCommand(), Fapi_setupBankSectorEnable(), and Fapi_issueBankEraseCommand() on this device. Usage of these functions for TMS320F28P551x is demonstrated in the flash API usage example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c*. Usage of these functions for 512-bit programming is demonstrated in the flash API usage example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c*.

2.2 API Overview

Table 2-1. 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	Description
Fapi_setActiveFlashBank()	Initializes flash wrapper and bank for an erase or program command.
Fapi_setupBankSectorEnable()	Configures the write and erase protection for the sectors.
Fapi_issueBankEraseCommand()	Issues bank erase command to the flash state machine (FSM) 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. **Note** A maximum of 128-bits (eight 16-bit words) can be programed at once.
Fapi_issueProgrammingCommandForEccAddress()	Remaps an error correction code (ECC) address to the main data space and then calls the Fapi_issueProgrammingCommand() function to program ECC.
Fapi_issueAutoEcc512ProgrammingCommand()	Sets up the required registers for 512-bit (thirty-two 16-bit words) programming with *AutoECC* generation mode and issues the command to the FSM.

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

API Function Description	Description
Fapi_issueDataAndEcc512ProgrammingCommand()	Sets up the required registers for 512-bit (thirty-two 16-bit words) programming with user provided flash data and ECC, and issues the command to the FSM.
Fapi_issueDataOnly512ProgrammingCommand()	Sets up the required registers for 512-bit (thirty-two 16-bit words) programming with user provided flash data and issues the command to the FSM.
Fapi_issueEccOnly64ProgrammingCommand()	Sets up the required registers for 64-bit (four 16-bit words) ECC programming with user provided ECC data and issues the command to the FSM.
Fapi_issueAsyncCommand()	Issues a command (clear status) to the FSM for operations that do not require an address.
Fapi_checkFsmForReady()	Returns whether or not the flash state machine 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 the specified flash memory range against the erased state.
Fapi_doVerify()	Verifies the specified flash memory range against the supplied values.

Note

Fapi_calculatePsa() and Fapi_doPsaVerify() functions 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 the flash wrapper.
Fapi_calculateEcc()	Calculates the ECC for the supplied address and 64-bit word.
Fapi_isAddressEcc()	Determines if the address falls in the ECC ranges.
Fapi_remapEccAddress()	Remaps an ECC address to the corresponding main address.
Fapi_calculateFletcherChecksum()	Calculates a Fletcher checksum for the memory range specified. The Fapi_calculateFletcherChecksum() function is deprecated in future devices.

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 updates the API internal state variables.

2.3.2 Building With the API

2.3.2.1 Objects Library Files

The flash API object file is distributed in the Arm® standard embedded application binary interface (EABI) executable and linking format (ELF) and common object file formats (COFF).

Note

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

2.3.2.2 Distribution Files

The following API files are distributed in the C2000Ware\libraries\flash_api\f28p551x\folder:

- Library files
  - TMS320F28P551x flash API is NOT embedded into the boot ROM of this device, flash API is wholly software. The software libraries provided are in EABI ELF (FAPI_F28P551x_EABI_v6.00.00.lib) and COFF (FAPI_F28P551x_COFF_v6.00.00.lib) object formats. For the application to be able to erase or program the flash or OTP memory, one of these two library files must be included in the application, depending on the output object format the application is using.
- FAPI_F28P551x_EABI_v6.00.00.lib – This is the flash API EABI ELF object format library (FPU32 flag enabled for build) for TMS320F28P551x devices.
- FAPI_F28P551x_COFF_v6.00.00.lib – This is the flash API COFF object format library (FPU32 flag enabled for build) for TMS320F28P551x devices.
- A fixed point version of the API library is not provided.
- Include Files:
  - FlashTech_f28p551x_C28x.h – The main *include* file for TMS320F28P551x devices. This file sets up compile specific defines and then includes the FlashTech.h file.
  - hw_flash_command.h – Definitions of the flash write and erase protection registers.
- The following *include* files must 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.
  - Registers.h – Definitions common to all register implementations and includes the appropriate *register include* file for the selected device type.
  - Registers_C28x.h – Flash configuration register.
  - Types.h – Contains all the enumerations and structures used by the API.
  - Constants/F28P551x.h – Constant definitions for F28P551x devices.

2.3.2.3 Key Facts For Flash API Usage

Here are some important facts about API usage:

- Names of the flash API functions start with the prefix: Fapi_.
- Flash API does not configure PLL. The user application must configure the phase-locked loop (PLL) as needed and pass the configured central processing unit clock (CPUCLK) value to the Fapi_initializeAPI() function (details of this function are given later in this document).

Note

The flash API library does not support flash erase and 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 check is up to the system integrator—TI suggests using the dual-clock comparator (DCC) module to check the system frequency. For an example implementation, see the C2000Ware driverlib clock configuration function.
- Always configure waitstates according to the device-specific data manual before calling the flash API functions. The flash API issues 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, read and fetch access from the flash bank is not permissible when an erase or program operation is in progress. Therefore, the flash API functions, the user application functions that call the flash API functions, and any interrupt service routines (ISRs) 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 the conditions apply to the entire code-snippet is because the Fapi_issueAsyncCommandWithAddress() function issues the erase command to the FSM, but the FSM 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 is not 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 or disable) the watchdog timer. The user application can configure the watchdog timer and service the watchdog timer as necessary. Hence, the Fapi_ServiceWatchdogTimer() function is no longer provided.
- Flash API uses EALLOW and EDIS internally as needed to allow or disallow writes to protected registers.
- The main array flash programming must be aligned to 64-bit address boundaries (alignment on a 128-bit address boundary is suggested) and each 64-bit word must only be programmed once per write or erase cycle.
- The data and ECC can be programed separately. However, each 64-bit data word and the corresponding ECC word must only be programmed once per write or erase cycle.
- There is no pump semaphore in TMS320F28P551x devices.
- ECC must 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 uses 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 is expected to exercise caution in these instances. Care must be taken to maintain a separate structure or section for link-pointer locations in the application. Do not mix these fields with other dual code security module (DCSM) OTP settings. If other fields are mixed with link-pointers, API skips programming ECC for the non-link-pointer locations as well. This causes ECC errors in the application.
- When using 128-bit DCSM OTP programming, the DCSM OTP address must be aligned to 128-bit address boundaries and each 128-bit word must only be programmed once. The exceptions are:
  - The DCSM Zx-LINKPOINTER1 and Zx-LINKPOINTER2 values in the DCSM OTP must be programmed together, and can be programmed 1 bit at a time as required by the DCSM operation.
  - The DCSM Zx-LINKPOINTER3 values in the DCSM OTP must be programmed 1 bit at a time as required by the DCSM operation.
  - Users can program JLM_Enable separately if needed. If users want to program all the 64-bits together, users must read the link-pointer (ZxOTP_LINKPOINTER1, ZxOTP_LINKPOINTER2, and ZxOTP_LINKPOINTER3) and program that value along with the JLM_Enable value.
  - 64-bits can be programmed in a given aligned 128-bit word when the other aligned 64-bits are reserved.

Note

Programing DCSM OTP is allowed in some 512-bit programming modes except for link-pointers range (0x00078000 to 0x00078005 and 0x00078200 to 0x00078205). For programing link pointer locations, use the Fapi_issueProgrammingCommand() function with AutoECCGeneration mode. For more details, see Table 2-6.

Table 2-6. Programming Function Capabilities: DCSM OTP and Link Pointer Restrictions

512-Bit Programming Functions	Programing of DCSM OTP and Link Pointer Range
Fapi_issueAutoEcc512ProgrammingCommand()	DCSM OTP allowed, link pointer not allowed.
Fapi_issueDataAndEcc512ProgrammingCommand()	DCSM OTP allowed, link pointer not allowed.
Fapi_issueDataOnly512ProgrammingCommand()	DCSM OTP and link pointer not allowed.
Fapi_issueEccOnly64ProgrammingCommand()	ECC DCSM OTP allowed, ECC link pointers not allowed.

- To avoid conflict between zone1 and zone2, a flash semaphore (FLSEM) is provided in the DCSM registers to configure flash registers. The user application must configure this semaphore register before initializing the flash and calling the flash API functions. For more details on this register, see the *TMS320F28P551/2/8 Real-Time Microcontrollers Technical Reference Manual* for TMS320F28P551x.
- The flash API functions do not configure any of the DCSM registers. The user application must configure the required DCSM settings. For example, if a zone is secured, then flash API must be executed from the same zone to be able to erase or program the flash sectors of that zone. Or the zone must be unlocked. If not, flash APIs written to flash registers do not succeed. Flash API does not check whether the writes to the flash registers are going through or not. Flash API writes to the registers as required for the erase or program sequence and returns the status back assuming that the writes went through. This causes the flash API to return false success status. For example, Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, Address) when called, can return the success status but that does not mean that the sector erase is successful. The erase status must be checked using the Fapi_getFSMStatus() and Fapi_doBlankCheck() functions.
- There must not be any access to the flash bank or OTP memory on which the flash erase or program operations are 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

Table 3-1. Fapi_initializeAPI Parameters

Parameters	Description
poFlashControlRegister [in]	Pointer to the base address of the flash wrapper registers. 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 the RWAIT register is changed.

Note

The value of the RWAIT register must be set before calling this function.

Return Value

- **Fapi_Status_Success** (Success)
- **Fapi_Error_InvalidHclkValue** (Failure: System clock does not match the specified wait value)
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register write failed. The user must verify that the API is executing from the same zone as that of the target address for the flash operation OR the user must unlock before the flash operation.)

Sample Implementation

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c* for TMS320F28P551x.

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c* for TMS320F28P551x.

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

Table 3-2. Fapi_setActiveFlashBank Parameters

Parameters	Description
oNewFlashBank [in]	Bank number to set as active. The Fapi_FlashBank0 is used for this device even though there are multiple banks.

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

The application must call this only once, and that call can be with the Fapi_FlashBank0.

Return Value

- **Fapi_Status_Success** (Success)

Sample Implementation

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c* for TMS320F28P551x.

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c* for TMS320F28P551x.

3.2.2 Fapi_setupBankSectorEnable()

Configures write (program) and erase protection for the sectors.

Synopsis

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

Parameters

Table 3-3. Fapi_setupBankSectorEnable Parameters

Parameters	Description
pu32StartAddress [in]	Register address for write and erase protection configuration. Use *FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTA* for the first 32 (0-31) sectors. Use *FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROTB* for the remaining main-array (32–127) sectors. Use *FLASH_WRAPPER_PROGRAM_BASE+FLASH_O_CMDWEPROT_UO* 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. The application of the user must disable the protection for the sectors that the application

wants to perform erase or program (or both) operations. This function can be used to enable or disable the protection. This function is called before each erase and program command, as shown in the flash API usage example provided in the C2000Ware.

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

The *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 are not meant to be erased and programmed. If a bit in the mask is 1, that particular sector is not erased or programmed.

The *CMDWEPROTB* register is used to configure the protection for the 32–127 sectors in the main-array flash bank. However, note that each bit in this register is used to configure protection for eight 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 are not meant to be erased and programmed. If a bit in the mask is 1, that particular set of sectors is not erased or programmed. If a bit in the mask is 0, that particular set of sectors is erased or programmed.

The *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 is not erased or programmed. Other bits can be configured as 1s. Since USER OTP is not erasable, the CMDWEPROT_UO register protection is not applicable for erase operations. This must be configured only for the program operation as needed.

Return Value

- **Fapi_Status_Success** (Success)
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register write failed. The user must verify that the API is executing from the same zone as that of the target address for the flash operation OR the user must unlock before the flash operation.)
- **Fapi_Error_FeatureNotAvailable** (Failure: User requested a write to non *CMDWEPROT* protected registers.)

Sample Implementation

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c* for TMS320F28P551x.

See the flash programming example provided in *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c* for TMS320F28P551x.

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

Table 3-4. Fapi_issueAsyncCommandWithAddress Parameters

Parameters	Description
oCommand [in]	Command to issue to the FSM. Use Fapi_EraseSector.
pu32StartAddress [in]	Flash sector address for the 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; the function just issues the command and returns back the status. Hence, this function always returns a 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 the STATCMD value after issuing the erase command. The user application must check the value of the STATCMD value when the FSM has completed the erase operation. The STATCMD value 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 must 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_FeatureNotAvailable** (Failure: User requested a command that is not supported).
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register write failed. The user must verify that the API is executing from the same zone as that of the target address for the flash operation OR the user must unlock before the flash operation.)
- **Fapi_Error_InvalidAddress** (Failure: User provided an invalid address. For the valid address range, see the *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

Sample Implementation

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c* for TMS320F28P551x.

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c* for TMS320F28P551x.

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

Table 3-5. Fapi_issueBankEraseCommand Parameters

Parameters	Description
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 the function 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_InvalidAddress** (Failure: User provided an invalid address. For the valid address range, see the *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

Sample Implementation

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c* for TMS320F28P551x .

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c* for TMS320F28P551x .

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

Table 3-6. Fapi_issueProgrammingCommand Parameters

Parameters	Description
pu32StartAddress [in]	Start address in flash for the data and ECC to be programmed. Also, the start address must always be even.
pu16DataBuffer [in]	Pointer to the data buffer address. Data buffer must 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 automatically generates and programs the ECC. Fapi_DataAndEcc programs both the data and ECC buffers. Fapi_EccOnly programs only the ECC buffer.

Note

The pu16EccBuffer must contain ECC corresponding to the data at the 128-bit aligned main array or OTP address. The least significant bit (LSB) of the pu16EccBuffer corresponds to the lower 64 bits of the main array and the most significant bit (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. The function offers four different programming modes to the user for use in different scenarios as mentioned in Uses of Different Programming Modes.

For the allowed programming range of the function, see Table 3-7.

Table 3-7. Uses of Different Programming Modes

Programming Mode (oMode)	Arguments Used	Usage Purpose
Fapi_DataOnly	pu32StartAddress, pu16DataBuffer, u16DataBufferSizeInWords	Used when any custom programming utility or user application (that embeds or uses the flash API) must program data and the corresponding ECC separately. Data is programmed using Fapi_DataOnly mode and then the ECC is programmed using Fapi_EccOnly mode. Typically, most of the programming utilities do not calculate ECC separately and instead use Fapi_AutoEccGeneration mode. However, some safety applications can require the insertion of intentional ECC errors in the 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). The application can insert errors in either the main array data or in the ECC as needed. In such scenarios, after the error insertion, the Fapi_DataOnly 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 embeds or uses flash API to program flash at run time to store data or to do a firmware update) must program data and ECC together without inserting any intentional errors. This mode is the most prominently used mode.
Fapi_DataAndEcc	pu32StartAddress, pu16DataBuffer, u16DataBufferSizeInWords, pu16EccBuffer, u16EccBufferSizeInBytes	The purpose of this mode is not different than that of using the 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 u16EccBufferSizeInBytes	See the usage purpose given for the Fapi_DataOnly mode.

Table 3-8. Permitted Programming Range for Fapi_issueProgrammingCommand()

Flash API	Main Array	DCSM OTP	ECC	Link Pointer
Fapi_issueProgrammingCommand() 128-bit, Fapi_AutoEccGeneration mode	Allowed	Allowed	Allowed	Allowed (Flash API programs the module as *Fapi_DataOnly)*
Fapi_issueProgrammingCommand() 128-bit, Fapi_DataOnly mode	Allowed	Allowed (Flash API programs the module as *Fapi_AutoEccGeneration)*	Not allowed	Allowed
Fapi_issueProgrammingCommand() 128-bit, Fapi_DataAndEcc mode	Allowed	Allowed	Allowed	Allowed (Flash API programs the module as *Fapi_DataOnly)*
Fapi_issueProgrammingCommand() 128-bit, Fapi_EccOnly mode	Not allowed	Not allowed	Allowed	Not allowed

Note

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

Programming Modes

Fapi_DataOnly – This mode only programs the data portion in Flash at the address specified. The mode 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 four and five are ignored when using this mode.

Fapi_AutoEccGeneration – This mode programs 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 must 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 cannot be reprogrammed (unless the sector is erased) even if ECC is programming a bit from 1 to 0 in that 64-bit data, since the new ECC value collides with the previously programmed ECC value. When using this mode, if the start address is 128-bit aligned, then either eight or four 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 four 16-bit words can be programmed at the same time. The data restrictions for Fapi_DataOnly mode also exist for this option. Arguments four and five are ignored.

Note

Fapi_AutoEccGeneration mode programs 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 are not be able to program the ECC for the first 64-bit word since ECC was already (incorrectly) computed and programmed using the 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 must track incomplete 64-bit words in a section and combine the words with the words in other sections that complete the 64-bit word. This is a difficult task. Therefore, TI recommends aligning your sections on 64-bit boundaries.

Some third-party Flash programming tools or TI Flash programming kernel examples (*C2000Ware*) or any custom Flash programming assumes that the incoming data stream is all 128-bit aligned and does not necessarily expect that a section potentially starts on an unaligned address. Thus, the select programming can 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, TI suggests aligning all the sections (mapped to Flash) on a 128-bit boundary.

Fapi_DataAndEcc – This mode programs 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 four 16-bit words, the ECC buffer must be one byte. If the data buffer length is eight 16-bit words, the ECC buffer must be two bytes in length. If the start address is 128-bit aligned, then either eight or four 16-bit words must be programmed at the same time as needed. If the start address is 64-bit aligned but not 128-bit aligned, then only four 16-bit words must 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 only programs the ECC portion in the Flash ECC memory space at the address (the Flash main array address must be provided for this function and not the corresponding ECC address) specified. The mode can program either two bytes (both LSB and MSB at a location in ECC memory) or one 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 eight and two, respectively.

Note

This function does not check STATCMD after issuing the program command. The user application must check the STATCMD value when the 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 must use the Fapi_doVerify() function to verify that the Flash is programmed correctly.

This function does not wait until the program operation is over; the function just issues the command and returns the command 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 must be used to monitor the status of an issued command.

Restrictions

- As described above, this function can program only a maximum 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 must 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 can only be programmed once per write or erase cycle.
- Programming the data and ECC separately is acceptable. However, each 64-bit data word and the corresponding ECC word can only be programmed once per write or erase cycle.
- ECC must not be programmed for *linkpointer* locations. The API issues 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 must not be used for the Bank0 DCSM OTP space. If used, an error is returned. For the DCSM OTP space, either the Fapi_AutoEccGeneration or Fapi_DataAndEcc programming modes must be used.

Return Value

- **Fapi_Status_Success** (Success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (Failure: Data buffer size specified is incorrect. Also, this error is returned if Fapi_EccOnly mode is selected when programming the Bank0 DCSM OTP space.)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (Failure: The ECC buffer size specified is incorrect.)
- **Fapi_Error_AsyncDataEccBufferLengthMismatch** (Failure: Data buffer size is either not 64-bit aligned or data length crosses the 128-bit aligned memory boundary.)
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register writes failed. The user must verify that the API is executing from the same zone as that of the target address for flash operation OR the user must 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

Sample Implementation

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c* for TMS320F28P551x.

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c* for TMS320F28P551x.

3.2.6 Fapi_issueProgrammingCommandForEccAddresses()

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

Synopsis

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

Parameters

Table 3-9. Fapi_issueProgrammingCommandForEccAddresses Parameters

Parameters	Description
pu32StartAddress [in]	The 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. 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 remaps an address in the ECC memory space to the corresponding data address space and then calls the Fapi_issueProgrammingCommand() function to program the supplied ECC data. The same limitations for the Fapi_issueProgrammingCommand() function 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 two.

Note

This function does not check STATCMD after issuing the program command. The user application must check the STATCMD value when the 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 must not be used for the Bank0 DCSM OTP space. If used, an error is returned. For the DCSM OTP space, either Fapi_AutoEccGeneration or Fapi_DataAndEcc programming modes must be used.

Return Value

- **Fapi_Status_Success** (Success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (Failure: Data buffer size specified is incorrect.)
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register writes failed. The user must verify that the API is executing from the same zone as that of the target address for flash operation OR the user must unlock before the flash operation.)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. For the valid address range, see the *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

3.2.7 Fapi_issueAutoEcc512ProgrammingCommand()

Sets up data and issues 512-bit (32 16-bit words) AutoEcc generation mode program command to valid flash or OTP memory addresses.

Synopsis

```
Fapi_StatusType
Fapi_issueAutoEcc512ProgrammingCommand (
                                uint32 *pu32StartAddress,uint16*pu16DataBuffer,
                               uint16u16DataBufferSizeInWords
                                          )
```

Parameters

Table 3-10. Fapi_issueAutoEcc512ProgrammingCommand Parameters

Parameters	Description
pu32StartAddress [in]	Start address in flash for the data and ECC to be programmed.
pu16DataBuffer [in]	Pointer to the data buffer address. Address of the data buffer must be 512-bit aligned.
u16DataBufferSizeInWords [in]	Number of 16-bit words in the data buffer. Maximum data buffer size in words must not exceed 32.

Description

This function automatically generates eight bytes of ECC data for the user-provided 512-bit data (second parameter) and programs the data and ECC together at the user-provided 512-bit aligned flash address (first parameter). When this command is issued, the flash state machine programs all of the 512 bits along with ECC. Hence, when using this mode, data not supplied is treated as all 1s (0xFFFF). Once ECC is calculated and programmed for a 512-bit data, those 512 bits cannot be reprogrammed (unless the sector is erased) even if the ECC is programming a bit from 1 to 0 in that 512-bit data, since the new ECC value collides with the previously programmed ECC value.

Note

The Fapi_issueAutoEcc512ProgrammingCommand() function programs the supplied data portion in flash along with automatically generated ECC. The ECC is calculated for a 512-bit aligned address and the corresponding 512-bit data. Any data not supplied is treated as 0xFFFF. 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 512-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 are not able to program the ECC for the first 512-bit word since the second section was already (incorrectly) computed and programmed using the assumed 0xFFFF for the missing values. One way to avoid this problem is to align all sections linked to flash on a 512-bit boundary in the linker command file for your code project.

```
SECTIONS
  {
    .text       : > FLASH, ALIGN(32)
    .cinit      : > FLASH, ALIGN(32)
    .const      : > FLASH, ALIGN(32)
    .init_array : > FLASH, ALIGN(32)
    .switch     : > FLASH, ALIGN(32)
  }
```

If you do not align the sections in flash, you need to track incomplete 512-bit words in a section and combine these words with the words in other sections that complete the 512-bit word. This is difficult to do. Hence, TI recommends aligning your sections on 512-bit boundaries.

Some 3rd party flash programming tools or TI flash programming kernel examples (*C2000Ware*) or any custom flash programming option can assume that the incoming data stream is all 512-bit aligned and does not necessarily expect that a section starts on an unaligned address. Thus, the tool can try to program the maximum possible (512-bits) words at a time assuming that the address provided is a 512-bit aligned address. This can result in a failure when the address is not aligned. So, TI suggests aligning all the sections (mapped to flash) on a 512-bit boundary.

For allowed programing range for the function, see Table 3-11.

Table 3-11. Permitted Programming Range for Fapi_issueAutoEcc512ProgrammingCommand()

FlashAPI	Main Array	DCSM OTP	ECC	Link pointer
Fapi_issueAutoEcc512ProgrammingCommand()	Allowed	Allowed	Allowed	Not allowed

Restriction

- As described above, this function can program only a maximum of 512-bits (given the address provided is 512-bit aligned) at a time. If the user wants to program more than that, this function must be called in a loop to program 512-bits at a time.
- The main array flash programming must be aligned to 512-bit address boundaries and thirty-two 16-bit words can only be programmed once per write or erase cycle.
- 512-bit address range starting with link pointer address must always be programmed using the 128-bit Fapi_issueProgrammingCommand() function.

Return Value

- **Fapi_Status_Success** (Success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (Failure: Data buffer size specified is incorrect. Also, this error is returned if Fapi_EccOnly mode is selected when programming the Bank0 DCSM OTP space.)
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register writes failed. The user must verify that the API is executing from the same zone as that of the target address for flash operation OR the user must 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

Sample Implementation

Example implementation for this function will be released in the next C2000ware release.

3.2.8 Fapi_issueDataAndEcc512ProgrammingCommand()

Sets up the flash state machine registers for the 512-bit (32 16-bit words) programming with user-provided flash data and ECC data and issues the programing command to valid flash and OTP memory.

Synopsis

```
Fapi_StatusType Fapi_issueDataAndEcc512ProgrammingCommand(
                                uint32 *pu32StartAddress,
                                uint16 *pu16DataBuffer,
                                uint16 u16DataBufferSizeInWords,
                                uint16 *pu16EccBuffer,
                                uint16 u16EccBufferSizeInBytes
                                                        )
```

Parameter

Table 3-12. Fapi_issueDataAndEcc512ProgrammingCommand Parameters

Parameters	Description
pu32StartAddress [in]	512-bit aligned flash address to program the provided data and ECC.
pu16DataBuffer [in]	Pointer to the data buffer address. Address of the data buffer must be 512-bit aligned.
u16DataBufferSizeInWords [in]	Number of 16-bit words in the data buffer. Maximum data buffer size in words must not exceed 32.
pu16EccBuffer [in]	Pointer to the ECC buffer address
u16EccBufferSizeInBytes [in]	Number of 8-bit bytes in the ECC buffer. Maximum ECC buffer size in words must not exceed 8.

Description

This function programs both the user provided 512-bit data (second parameter) and eight bytes of ECC data (fourth parameter) together at the user-provided 512-bit aligned flash address. The address of data provided must be aligned on a 512-bit memory boundary and the length of data must correlate to the supplied ECC. That means, if the data buffer length is thirty-two 16-bit words, the ECC buffer must be eight bytes (one ECC bytes corresponding to 64-bit data).

Each byte of pu16EccBuffer corresponds to each 64-bit of the main array data provided in the *pu16DataBuffer*.

For more details, see Table 3-17.

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

For allowed programing range for the function, see Table 3-13.

Table 3-13. Permitted Programming Range for Fapi_issueDataAndEcc512ProgrammingCommand()

Flash API	Main Array	DCSM OTP	ECC	Link Pointer
Fapi_issueDataAndEcc512ProgrammingCommand()	Allowed	Allowed	Allowed	Not Allowed

Restrictions

- As described above, this function can program only a maximum of 512-bits (given the address provided is 512-bit aligned) at a time. If the user wants to program more than that, this function must be called in a loop to program 512-bits at a time.
- The main array flash programming must be aligned to 512-bit address boundaries and thirty-two 16-bit words can only be programmed once per write or erase cycle.
- 512-bit address range starting with link pointer address must always be programmed using the 128-bit Fapi_issueProgrammingCommand() function.

Return Value

- **Fapi_Status_Success** (Success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (Failure: Data buffer size specified is incorrect. Also, this error is 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 must verify that the API is executing from the same zone as that of the target address for flash operation OR the user must 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

Sample Implementation

Example implementation for this function will be released in next C2000ware release.

3.2.9 Fapi_issueDataOnly512ProgrammingCommand()

Sets up the flash state machine registers for the 512-bit (32 16-bit words) programming with user-provided flash data and issues the programing command to valid flash.

Synopsis

```
Fapi_StatusType
Fapi_issueDataOnly512ProgrammingCommand(
                        uint32 *pu32StartAddress,
                        uint16 *pu16DataBuffer,
                        uint16 u16DataBufferSizeInWords
                                              )
```

Parameters

Table 3-14. Fapi_issueDataOnly512ProgrammingCommand Parameters

Parameters	Description
pu32StartAddress [in]	512-bit aligned flash address to program the provided data
pu16DataBuffer [in]	Pointer to the Data buffer address. Data buffer must be 512-bit aligned.
u16DataBufferSizeInWords [in]	Number of 16-bit words in the Data buffer. Max Data buffer size in words must not exceed 32.

Description

This function only programs the data portion in flash at the address specified. The function can program 512-bit data (second parameter) at the user-provided 512-bit aligned flash address.

This function is used when a user application (that embeds or uses the flash API) must program 512-bit of data and the corresponding 64-bit of ECC data separately. 512-bit data is programmed using the Fapi_issueDataOnly512ProgrammingCommand () function and then the 64-bit ECC is programmed using the Fapi_issueEccOnly64ProgrammingCommand() function. Generally, most of the programming utilities do not calculate ECC separately and instead use the Fapi_issueAutoEcc512ProgrammingCommand() function. However, some safety applications potentially require the insertion of intentional ECC errors in the 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). The application potentially requires the insertion of errors in either the main array data or in the ECC. In such scenarios, after the error insertion,

the Fapi_issueDataOnly512ProgrammingCommand() API and then the 64-bit ECC is programmed using the Fapi_issueEccOnly64ProgrammingCommand() API and can be used to program the data and ECC respectively.

For allowed programming range for the function, see Table 3-15.

Table 3-15. Permitted Programming Range for Fapi_issueDataOnly512ProgrammingCommand()

Flash API	Main Array	DCSM OTP	ECC	Link Pointer
Fapi_issueDataOnly512ProgrammingCommand()	Allowed	Not allowed	Not allowed	Not allowed

Restrictions

- As described above, this function can program only a maximum of 512-bits (given the address provided is 512-bit aligned) at a time. If the user wants to program more than that, this function must be called in a loop to program 512-bits at a time.
- The main array flash programming must be aligned to 512-bit address boundaries and thirty-two 16-bit words can only be programmed once per write or erase cycle.
- 512-bit address range starting with link pointer address must always be programmed using the 128-bit Fapi_issueProgrammingCommand() function.

Return Value

- **Fapi_Status_Success** (Success)
- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (Failure: Data buffer size specified is incorrect. Also, this error is returned if Fapi_EccOnly mode is selected when programming the Bank0 DCSM OTP space.)
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register writes failed. The user must verify that the API is executing from the same zone as that of the target address for flash operation OR the user must 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

Sample Implementation

Example implementation for this function will be released in next C2000ware release.

3.2.10 Fapi_issueEccOnly64ProgrammingCommand()

Sets up the flash state machine registers for the 64-bit (four 16-bit words) programming with user-provided ECC data and issues the programing command to valid Flash and OTP memory.

Synopsis

```
Fapi_StatusType
Fapi_issueEccOnly64ProgrammingCommand(
                    uint32 *pu32StartAddress,
                    uint16 *pu16EccBuffer,
                    uint16 u16EccBufferSizeInBytes
                                          )
```

Parameters

Table 3-16. Fapi_issueEccOnly64ProgrammingCommand Parameters

Parameters	Description
pu32StartAddress [in]	512-bit aligned flash address to program the provided ECC data.
pu16EccBuffer [in]	Pointer to the ECC buffer address.
u16EccBufferSizeInBytes[in]	Number of 8-bit bytes in the ECC buffer. Maximum ECC buffer size in words must not exceed eight.

Description

This function only programs the ECC portion in the flash ECC memory space at the address (the flash main array address must be provided for this function and not the corresponding ECC address) specified. The function can program 64-bit of ECC data (second parameter) at the ECC address corresponding to the user-provided 512-bit aligned flash address.

64-bit ECC data can be split as eight bytes ECC data correlated to 512-bit aligned data (4 × 128, each two bytes corresponding to each 128 data). For more information, see Table 3-17.

Table 3-17. 64-Bit ECC Data Interpretation

512 Bits Data (4 × 128bits)			
1st 128-Bit Data	2nd 128-Bit Data	3rd 128-Bit Data	4th 128-Bit Data
LSB of pu16EccBuffer[0]	LSB of pu16EccBuffer[1]	LSB of pu16EccBuffer[2]	LSB of pu16EccBuffer[3]
MSB of pu16EccBuffer[0]	MSB of pu16EccBuffer[1]	MSB of pu16EccBuffer[2]	MSB of pu16EccBuffer[3]

For allowed programing range for the function, see Table 3-18.

Table 3-18. Permitted Programming Range for Fapi_issueEccOnly64ProgrammingCommand()

Flash API	Main Array	DCSM OTP	ECC	Link Pointer
Fapi_issueEccOnly64ProgrammingCommand()	Not Allowed	Not Allowed	Allowed	Not Allowed

Restrictions

- As described above, this function can program only a maximum of 64-bits ECC at a time. If the user wants to program more than that, this function must be called in a loop to program 64-bits at a time.
- The main array flash programming must be aligned to 512-bit address boundaries and the 64-bit ECC word can only be programmed once per write or erase cycle.
- ECC must not be programmed for link pointer locations. 512-bit address range starting with link pointer address must always be programmed using the 128-bit Fapi_issueProgrammingCommand() function.

Return Value

- **Fapi_Status_FsmBusy** (FSM busy)
- **Fapi_Status_Success** (Success)
- **Fapi_Error_AsyncDataEccBufferLengthMismatch** (Failure: Data buffer size is either not 64-bit aligned or data length crosses the 128-bit aligned memory boundary.)
- **Fapi_Error_FlashRegsNotWritable** (Failure: Flash register writes failed. The user must verify the API is executing from the same zone as that of the target address for flash operation OR the user must 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

Sample Implementation

Example implementation for this function will be released in next C2000ware release.

3.2.11 Fapi_issueAsyncCommand()

Issues a command to the flash state machine. For the list of commands that can be issued by this function, see the description.

Synopsis

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

Parameters

Table 3-19. Fapi_issueAsyncCommand Parameters

Parameters	Description
oCommand [in]	Command to issue to the FSM. Use the 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, the Fapi_ClearStatus command is issued to the flash state machine using this function. The Fapi_ClearStatus command must 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

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_128bit_programming.c* for TMS320F28P551x.

See the flash programming example provided in C2000Ware at *C2000Ware_.....\driverlib\f28p551x\examples\flash\flashapi_512bit_programming.c* for TMS320F28P551x.

3.2.12 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 the FSM 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.)

3.2.13 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 must check the value of this register to determine if there is any failure after each erase and program operation.

Table 3-20. 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[1] - 0x0				RO - 0x0		RO - 0x0	RO - 0x0	RO - 0x0		RO - 0x0	RO - 0x0	RO - 0x0

(1) RO – Read only.

Table 3-21. STATCMD Register Field Descriptions

Bit	Name	Description	Reset Value
12	FAILMISC	Command failed due to error other than write or erase protect violation or verify error. 0: No fail 1: Fail	0x0
8	FAILINVDATA	Program command failed because an attempt was made to program a stored 0 value to a 1. 0: No fail 1: Fail	0x0
6	FAILILLADDR	Command failed due to the use of an illegal address. 0: No fail 1: Fail	0x0
5	FAILVERIFY	Command failed due to verify error. 0: No fail 1: Fail	0x0
4	FAILWEPROT	Command failed due to write or erase protect sector violation. 0: No fail 1: Fail	0x0
2	CMDINPROGRESS	Command in progress 0: Complete 1: In progress	0x0
1	CMDPASS	Command Pass – valid when CMD_DONE field is 1 0: Fail 1: Pass	0x0
0	CMDDONE	Command done 0: Not done 1: 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

Table 3-22. Fapi_doBlankCheck Parameters

Parameters	Description
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, the corresponding address and data are 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

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

Table 3-23. Fapi_doVerify Parameters

Parameters	Description
pu32StartAddress [in]	Start address for region to verify.
u32Length [in]	Length of region in 32-bit words to verify.
u32Length [in]	Address of buffer to verify region against. Data buffer must be 128-bit aligned.
poFlashStatusWord [out]	Returns the status of the operation if the result is not Fapi_Status_Success.
→au32StatusWord[0]	Address of first verify failure location.

Table 3-23. Fapi_doVerify Parameters (continued)

Parameters	Description
→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 are 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 *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* for TMS320F28P551x.)

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 six.
- u8ApiMinorVersion – Minor version number of this compile of the API. Minor version is zero for F28P551x devices.
- u8ApiRevision – Revision version number of this compile of the API.

Revision number is 10 for this release.

- oApiProductionStatus – Production status of this compile (Alpha_Internal, Alpha, Beta_Internal, Beta, and 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 F28P551x devices.
- u32ApiCompilerVersion – Version number of the Code Composer Studio™ software 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

Table 3-24. Fapi_calculateEcc Parameters

Parameters	Description
u32Address [in]	Address of the 64-bit value to calculate the ECC.
u64Data [in]	64-bit value to calculate ECC on (must be in little endian order)

Description

This function calculates the ECC for a 64-bit aligned word including address. There is no need to provide a left-shifted address to this function anymore. The flash API takes care of this function.

Return Value

- 8-bit calculated ECC (upper eight bits of the 16-bit return value must 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

Table 3-25. Fapi_isAddressEcc Parameters

Parameters	Description
u32Address [in]	Address to determine if value lies in ECC address space.

Description

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

Return Value

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

3.5.4 Fapi_remapEccAddress()

Takes the ECC address and remaps the address to the 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. 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**
- **0xFFFFFFFF** – If the flash ECC address provided is invalid.

3.5.5 Fapi_calculateFletcherChecksum()

Calculates the Fletcher checksum from the given address and length.

Synopsis

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

Parameters

Table 3-26. Fapi_calculateFletcherChecksum Parameters

Parameters	Description
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. Only the flash main array address range can be used for this function. The DCSM OTP address range must 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. TI recommends, but does not require, that users complete a blank check on devices received to verify that the devices are erased.

4.2 Recommended Erase Flow

Figure 4-1 describes the flow for erasing a sector (or sectors) on a device. For further information, see Section 3.2.11, Section 3.2.2, and Section 3.2.3.

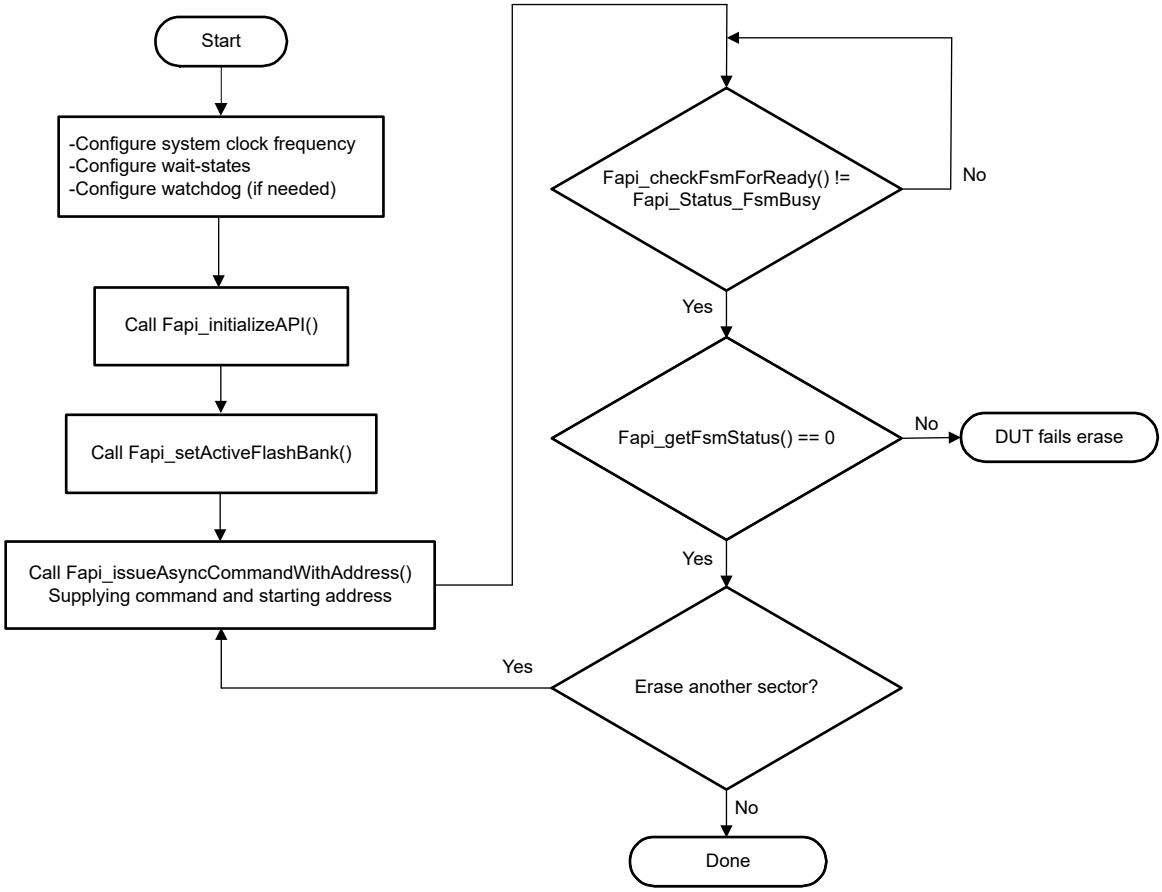

Figure 4-1. Recommended Erase Flow

Note

Before issuing the sector erase command, the Fapi_ClearStatus command must be issued and the write and erase protections must be configured for the sectors as required 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 3.2.11, Section 3.2.2, and Section 3.2.3.

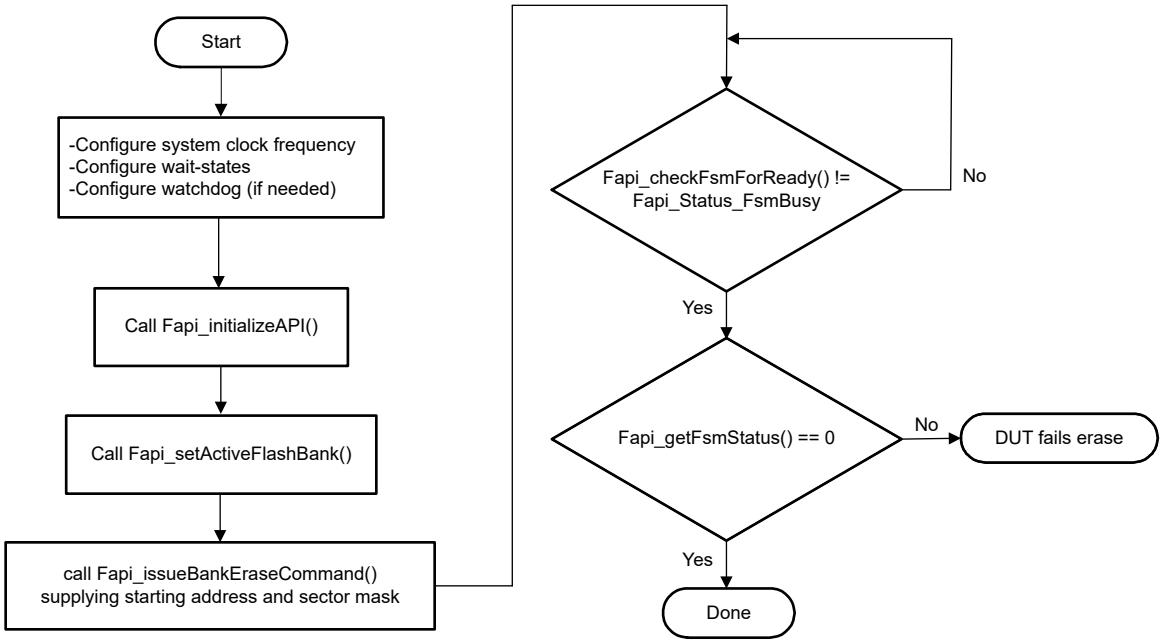

Figure 4-2. Recommended Bank Erase Program Flow

Note

Before issuing the program command, the Fapi_ClearStatus command must be issued and the write and erase protections must be configured for the sectors as required 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 3.2.11, Section 3.2.2, and Section 3.2.3.

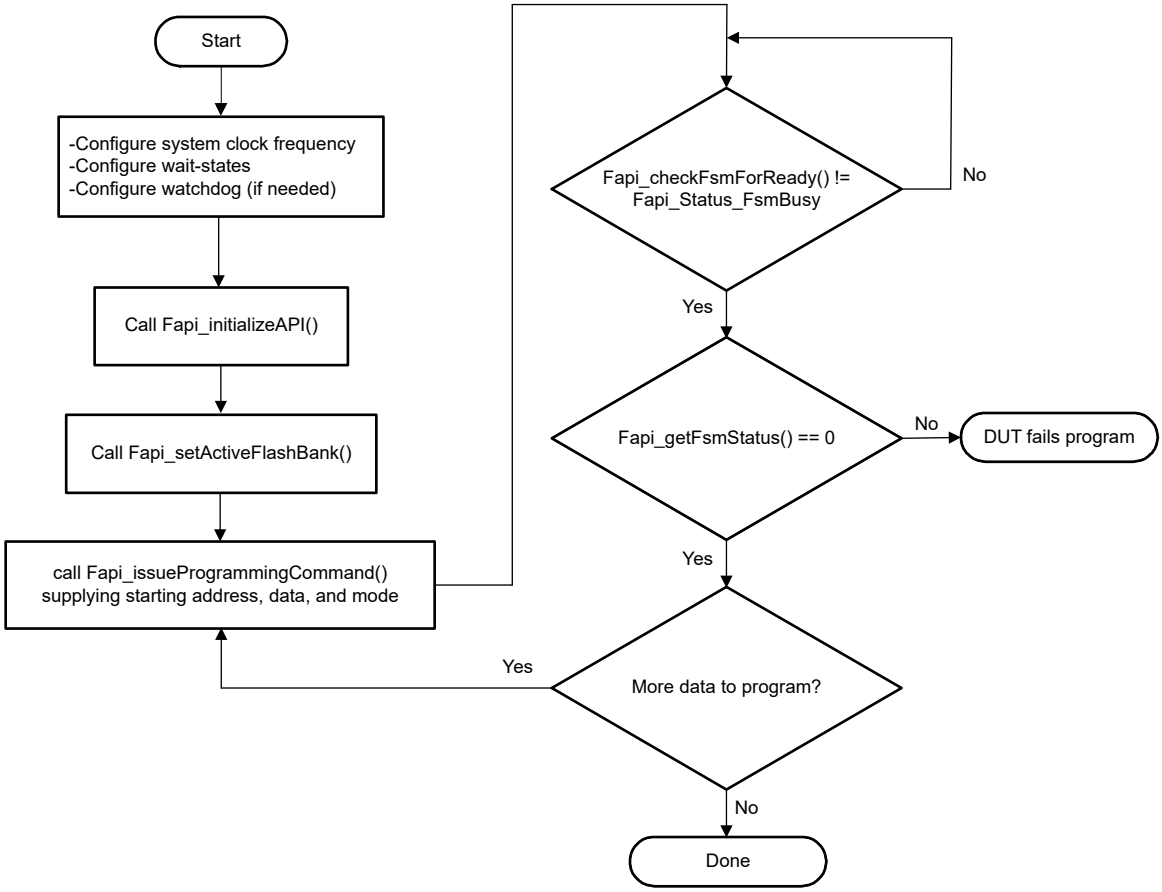

Figure 4-3. Recommended Program Flow

Note

Before issuing the program command, the Fapi_ClearStatus command must be issued and the write and erase protections must be configured for the sectors as required by the application.

5 Software Application Assumptions of Use Related to Safety

1. The Fapi_initializeAPI() function must be executed once before using any of the other flash API functions. There is no need to call this function more than once. However, this function must be called whenever the user application updates the system frequency or the flash wait state configuration at runtime (which is rare).
2. Before executing the Fapi_initializeAPI() function, the user application must verify that the PLL is configured correctly for the desired output frequency. Note that the SysCtl_setClock() function provided in the driverlib achieves this assumption by using the DCC module. The SysCtl_isPLLValid() function called by the SysCtl_setClock() function uses the DCC module to verify that the PLL output is in the expected range. For more details of the DCC module, see the device-specific technical reference manual.
3. If the user application requires protection of the flash register space from any corruption due to reasons like runaway code, the assumption can be accomplished by using the ERAD module. For ERAD usage details, see the device-specific technical reference manual.
4. When using the Fapi_BlankCheck() function for the flash main array address range, TI suggests checking the corresponding ECC array address range as well. The ECC address range for each sector is given in the *Memory Map* table in the device-specific datasheet.
5. User application software must enable the ECC evaluation for the flash read and fetch path using the ECC_ENABLE register as needed. Flash API does not enable or disable this register. Note that the ECC_ENABLE is enabled by default at power-up. Hence, the user application software defines the NMI ISR and flash single-bit error ISR to be able to respond for the single and uncorrectable flash errors. For more details on the FLASH_ECC_REGS registers, see the device-specific technical reference manual.
6. If the safety standards of the user application requires the application to read the entire flash memory range periodically to verify that the data is intact, the user application does so and calculates the checksum or CRC for the entire flash memory range. The VCU CRC module can be used to accomplish this assumption. For more details on the VCU CRC module, see the device-specific technical reference manual.
7. If the safety standards of the user application requires the application to read the flash register space periodically to verify that the register configuration is intact, the user application does so and calculates the checksum or CRC for the flash register memory range.
8. If the safety standards of the user application requires the application to verify (using CPU read) the flash content after program and erase operations, the user application can do so by using the Fapi_doVerify() and Fapi_doBlankCheck() functions, respectively.
9. Flash API does not configure or service the watchdog function. The user application can configure and service the watchdog function as needed (either serially in between the flash API function calls or using a CPU timer ISR).
10. If the flash API code in RAM or flash gets overwritten due to improper application design (for example, stack overflow), ITRAPs (illegal instruction trap) can occur when the CPU tries to execute the flash API functions. Hence, the user application defines the ITRAP ISR to respond such events.

A Flash State Machine Commands

Table A-1. Flash State Machine Commands

Command	Description	Enumeration Type	API Calls
Program Data	Used to program data to any valid flash address	Fapi_ProgramData	Fapi_issueProgrammingCommand() Fapi_issueProgrammingCommandForEccAddresses() Fapi_issueAutoEcc512ProgrammingCommand() Fapi_issueDataOnly512ProgrammingCommand() Fapi_issueDataAndEcc512ProgrammingCommand() Fapi_issueEccOnly64ProgrammingCommand()
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 “F65 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() function.

```
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 main 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_StateMachineTimeout,  /* State machine polling never returned ready and timed out */
   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 structure is used to return status values in functions that need more flexibility.

```
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;
```

References

- Texas Instruments: *TMS320F28P551, TMS320F28P552, and TMS320F28P558 Real-Time Microcontrollers Data Manual* , datasheet
- Texas Instruments: *TMS320F28P551/2/8 Real-Time Microcontrollers Technical Reference Manual* , technical reference manual.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025