TMS320F28M35x and TMS320F28M36x Flash API

Version 1.53

Reference Guide

Literature Number: SPNU595A
January 2014–Revised July 2017

Contents

1 Introduction 4
1.1 Reference Material 4
1.2 Function Listing Format 4
2 TMS320F28M35x/36x Flash API Overview 6
2.1 Introduction 6
2.2 API Overview 6
2.3 Using API 7
3 API Functions 10
3.1 Initialization Functions 10
3.2 Flash State Machine Functions 11
3.3 Read Functions 26
3.4 Informational Functions 34
3.5 Utility Functions 37
3.6 User Definable Functions 39
4 Recommended FSM Flows 43
4.1 New devices from Factory 43
4.2 Recommended Erase Flow 44
4.3 Recommended Program Flow 45
Appendix A Flash State Machine Commands 46
A.1 Flash State Machine Commands 46
Appendix B Object Library Function Information 47
B.1 ARM CortexM3 Library 47
B.2 C28x Library 48
Appendix C Typedefs, defines, enumerations and structures 51
C.1 Type Definitions 51
C.2 Enumerations 52
C.3 Structures 55
Appendix D Parallel Signature Analysis (PSA) Algorithm 57
D.1 Function Details 57
Appendix E ECC Calculation Algorithm 58
E.1 Function Details 58
Revision History 61

List of Figures

1 Recommended Erase Flow 44
2 Recommended Program Flow 45

List of Tables

1 Summary of Initialization Functions 6
2 Summary of Flash State Machine (FSM) Functions 6
3 Summary of Read Functions 6
4 Summary of Information Functions 7
5 Summary of Utility Functions 7
6 Summary of User-Defined Functions 7
7 Uses of Different Programming Modes 14
8 Uses of Different Programming Modes 18
9 FMSTAT Register 24
10 FMSTAT Register Field Descriptions 25
11 Flash State Machine Commands 46
12 ARM CortexM3 Function Sizes and Stack Usage 47
13 C28x Function Sizes and Stack Usage 48

1 Introduction

NOTE: This document is applicable only for TMS320F28M35x/6x devices.

This reference guide provides a detailed description of Texas Instruments' TMS320F28M35x/6x Flash API library functions that can be used to erase, program and verify Flash on TMS320F28M35x/6x devices.

The Flash API libraries are provided in controlSUITE™ at the locations below:

- TMS320F28M35x devices:
  - C28x libraries: F021_API_C28x.lib and F021_API_C28x_FPU32.lib are available at controlsuite/device_support/f28M35x/vx/F28M35x/lib folder.
  - ARM library: F021_API_CortexM3_LE.lib is available at controlsuite/device_support/f28M35x/Vx/MWare/lib.
- TMS320F28M36x devices:
  - C28x libraries: F021_API_C28x.lib and F021_API_C28x_FPU32.lib are available at controlsuite/device_support/F28M36x/vx/F28M36x/lib folder.
  - ARM library: F021_API_CortexM3_LE.lib is available at controlsuite/device_support/f28M36x/Vx/MWare/lib.

1.1 Reference Material

Use this guide in conjunction with:

- *F28M35x Concerto Microcontrollers Data Manual*
- *F28M36x Concerto Microcontrollers Data Manual*
- *Concerto F28M35x Technical Reference Manual*
- *Concerto F28M36x Technical Reference Manual*

1.2 Function Listing Format

This is the general format of an entry for a function.

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

Synopsis

Provides a prototype for function **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_n

Parameter passing is categorized as follows:

- *In* — Means the function uses one or more values in the parameter that you give it without storing any changes.
- *Out* — Means 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* — Means 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 this function.

See Also

Lists other functions or data types related to this 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 provided in controlSUITE, to demontrate the usage of a given Flash API function in an application context.

2 TMS320F28M35x/36x 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. Flash API can be used to program and verify the OTP memory as well.

NOTE: Please refer to the data manual for Flash and OTP's memory map and Flash waitstate specifications. Also, note that this reference guide assumes that the user has already read the *Flash Memory Controller Module* chapter in the TRM. See *Concerto F28M36x Technical Reference Manual* and *Concerto F28M35x Technical Reference Manual*.

2.2 API Overview

Table 1. Summary of Initialization Functions

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

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

API Function	Description
Fapi_setActiveFlashBank()	Initializes the Flash Memo Controller (FMC) and bank for an erase, program, or other command
Fapi_issueAsyncCommandWithAddress()	Issues an erase sector command to FSM for the given sector address
Fapi_issueProgrammingCommand()	Sets up the required registers for programming and issues the program command to the FSM
Fapi_issueProgrammingCommandForEccAddresses()	Remaps an Error Correction Code (ECC) address to the main data space and then calls Fapi_issueProgrammingCommand() to program the supplied ECC data
Fapi_issueFsmSuspendCommand()	Suspends program data and erase sector FSM commands
Fapi_issueAsyncCommand()	Issues a command (clear status, program resume, erase resume, clear more) to 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 FMSTAT status register value from the Flash memory controller

Table 3. Summary of Read Functions

API Function	Description
Fapi_doBlankCheck()	Verifies specified Flash memory range for the erased state
Fapi_doBlankCheckByByte()(1)	Verifies specified Flash memory range for the erased state by byte
Fapi_doVerify()	Verifies specified Flash memory range against supplied values
Fapi_doVerifyByByte()(1)	Verifies specified Flash memory range against supplied values by byte
Fapi_doMarginRead()	Reads the specified Flash memory range using the specified read-margin mode and returns the data
Fapi_doMarginReadByByte()(1)	Reads the specified Flash memory range using the specified read-margin mode by byte and returns the data
Fapi_calculatePsa()	Calculates a Parallel Signature Analysis (PSA) value for the specified Flash memory range
Fapi_doPsaVerify()	Verifies a specified Flash memory range against the supplied PSA value

(1) Not applicable for C28x cores.

Table 4. Summary of Information Functions

API Function	Description
Fapi_getLibraryInfo()	Returns the information specific to the compiled version of the API library
Fapi_getDeviceInfo()[1]	Returns the information specific to the device on which the API library is being executed
Fapi_getBankSectors()[2][1]	Returns the sector information for a bank

(1) These functions are not supported in future devices. Therefore, TI suggests not to use this function.
(2) This function returns the sector information for the maximum flash bank size configuration in any given family. For example, in F28M36x devices, some PART numbers will have 1MB Flash in M3 subsystem and some PART numbers will have 512KB Flash in M3 subsystem. However, this function is hardcoded to return the sector information, assuming the Flash size as 1MB (max Flash size in F28M36x M3 subsystem).

Table 5. Summary of Utility Functions

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

Table 6. Summary of User-Defined Functions

API Function	Description
Fapi_serviceWatchdogTimer()[1]	User-modifiable function to service watchdog timer
Fapi_setupEepromSectorEnable()[2]	Users should not modify this function. This function should be used as provided by TI.
Fapi_setupBankSectorEnable()[2]	User should not modify this function. This function should be used as provided by TI.

(1) This function is not supported in future devices. Therefore, TI suggests not to use this function.
(2) Users should not modify these functions, even though these functions are provided in the Fapi_User Defined Functions.c file. These functions are not merged into the library and are provided in the User-Defined section to maintain the same code across TI devices that share common code. These functions are merged into the library in subsequent 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 functions *Fapi_getLibraryInfo()* and *Fapi_getDeviceInfo())* can be used. This procedure initializes the API internal structures.

2.3.1.2 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 Fapi_getLibraryInfo() and Fapi_getDeviceInfo()) can be used. This will update the API internal state variables.

2.3.2 Building With the API

2.3.2.1 Object Library Files

The ARM Cortex Flash API object file is distributed in the ARM standard EABI elf object format. C28x Flash API object files are distributed in the standard Common Object File Format (COFF).

NOTE: Compilation with the TI ARM and C28x codegen tools requires "Enable support for GCC extensions" option to be enabled.

ARM Compiler version 5.2.0 and onwards have this option enabled by default.

C2000 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 controlSUITE:

- Library Files
  - F021_API_CortexM3_LE.lib – This is the Flash API object file for the Cortex M3 master subsystem in F28M35x/F28M36x.
  - F021_API_C28x.lib – This is the Flash API object file for the C28x control subsystem in F28M35x/F28M36x.
  - F021_API_C28x_FPU32.lib – This is the Flash API object file for the C28x control subsystem applications in F28M35x/F28M36x that are using floating point unit.
- Source Files
  - Fapi_UserDefinedFunctions.c – This file contains the user-definable functions and must be compiled with the user's code. This file is provided in controlSUITE at controlsuite\device_support\F28m3xx\VX\F28m3xx_examples_Dual\flash_prog\m3\C28x
- Include Files
  - These files set up compile-specific defines and then includes the F021.h master include file.
    - F021_Concerto_C28x.h – The master include file for Concerto C28x application.
    - F021_Concerto_Cortex.h – The master include file for Concerto Cortex M3 application.
- The following include files should not be included directly by the user’s code, but are listed here for user reference:
  - F021.h – This include file lists all API functions and includes all other include files.
  - Helpers.h – Set of Helper defines.
  - Init.h – Defines the API initialization structure.
  - Registers_Concerto_C28x.h – Flash memory controller registers structure for Concerto C28x applications.
  - Registers_Concerto_Cortex.h – Flash memory controller registers structure for Concerto Cortex M3 applications.
  - 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/Concerto.h – Constant definitions for Concerto 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_”.
- For C28x: EALLOW and EDIS should be executed before and after calling Flash API functions, respectively, to allow and disallow writes to protected registers.
- For M3: Before calling Flash API functions, MWRALLOW should be configured to allow API to write to protected register writes. Protected register writes can be disabled as needed after the Flash API usage.
- Pump semaphore should be gained by a CPU before performing Flash operations (erase, program, verify) on its bank. Flash API does not configure the pump semaphore.
- Flash API does not configure the 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).
- Always configure waitstates as per the device data manual before calling Flash API functions. Flash API will issue an error if the waitstate configured by the application is not appropriate for the operating frequency of the application. See Fapi_Set ActiveFlashBank() function for more details.
- Flash API does not configure (enable or disable) the watchdog. The user application can configure the watchdog and service it as needed. In subsequent devices, the Fapi_ServiceWatchdogTimer() function is no longer supported. Therefore, TI suggests to not use this function; instead, the user applications can service the watchdog at regular interrupts (for example, by using a timer ISR) as needed.
- Flash API execution is interruptible; however, there should not be any read or fetch access from the Flash bank/OTP when an erase or program operation is in progress on that Flash bank/OTP. 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 entire code snippet shown below should be executed from RAM and not just 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, there should not be a Flash access.

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

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 Memory Controller Registers base address Use F021_CPU0_BASE_ADDRESS for this parameter.
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 System frequency or RWAIT is changed.

NOTE: RWAIT value must be configured by the user before calling this function.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

Please refer to the example provided in controlSUITE at below location:

For TMS320F28M35x: ti\controlSUITE\device_support\f28m35x\vx\F28M35x_examples_Dual\flash_prog

For TMS320F28M36x: ti\controlSUITE\device_support\f28m36x\vx\F28M36x_examples_Dual\flash_prog

3.2 Flash State Machine Functions

3.2.1 Fapi_setActiveFlashBank()

Initializes the FMC for erase and program operations

Synopsis

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

Parameters

oNewFlashBank [in]	Bank number to set as active. Since there is only one bank per FMC in these devices, only Fapi_FlashBank0 should be used for this parameter.

Description

This function configures FMC for Flash operations to be performed on the bank.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_InvalidBank** (failure: Bank specified does not exist on device)
- **Fapi_Error_InvalidHclkValue** (failure: System clock does not match specified wait value)
- **Fapi_Error_OtpChecksumMismatch** (failure: Calculated TI OTP checksum does not match value in TI OTP)

Sample Implementation

Please refer to the example provided in controlSUITE at below location:

For TMS320F28M35x: ti\controlSUITE\device_support\f28m35x\vx\F28M35x_examples_Dual\flash_prog

For TMS320F28M36x: ti\controlSUITE\device_support\f28m36x\vx\F28M36x_examples_Dual\flash_prog

3.2.2 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, the user application must wait for the FMC to complete the erase operation before returning to any kind of Flash accesses.

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

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_FeatureNotAvailable** (failure: user requested a command that is not supported)

Sample Implementation

Please refer to the example provided in controlSUITE at below location:

For TMS320F28M35x: ti\controlSUITE\device_support\f28m35x\vx\F28M35x_examples_Dual\flash_prog

For TMS320F28M36x: ti\controlSUITE\device_support\f28m36x\vx\F28M36x_examples_Dual\flash_prog

3.2.3 Fapi_issueProgrammingCommand()

3.2.3.1 For ARM Cortex devices

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

Synopsis

```
Fapi_StatusType Fapi_issueProgrammingCommand(
        uint32 *pu32StartAddress,
        uint8  *pu8DataBuffer,
        uint8   u8DataBufferSizeInBytes,
        uint8  *pu8EccBuffer,
        uint8   u8EccBufferSizeInBytes,
        Fapi_FlashProgrammingCommandType oMode)
```

Parameters

pu32StartAddress [in]	start address in Flash for the data and ECC to be programmed	
pu8DataBuffer [in]	pointer to the Data buffer address	
pu8DataBufferSizeInBytes [in]	number of bytes in the Data buffer	
pu8EccBuffer [in]	pointer to the ECC buffer address	
pu8EccBufferSizeInBytes [in]	number of 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

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 7.

Table 7. Uses of Different Programming Modes

Programming mode (oMode)	Arguments used	Usage purpose
Fapi_DataOnly	pu32StartAddress, pu8DataBuffer, pu8DataBufferSizeInWords	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 either the ECC calculation algorithm provided in Section E.1.1 or 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. Also, this mode is used when not all the 64 bits of the data is available to program along with ECC. In this case, users can program less than 64 bits without ECC using this mode. And then later program all the 64 bits when available along with ECC. However, note that in subsequent devices, it is restricted to program all the 64 bits of data at a time and hence TI suggests to program all the 64-bits at a time if possible.
Fapi_AutoEccGeneration	pu32StartAddress, pu8DataBuffer, pu8DataBufferSizeInWords	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.
Fapi_DataAndEcc	pu32StartAddress, pu8DataBuffer, pu8DataBufferSizeInWords, pu8EccBuffer, pu8EccBufferSizeInBytes	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	pu8EccBuffer, pu8EccBufferSizeInBytes	See the usage purpose given for Fapi_DataOnly mode.

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 16 bytes. 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 portion in Flash along with automatically generated ECC. The ECC is calculated for every 64-bit data aligned on a 64-bit memory boundary. All the 64-bits of the data must be supplied. Data not supplied is assumed as all 1s (0xFF). 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.

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 0xFF. 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 0xFF 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 0xFF 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, PAGE = 0, ALIGN(8)
.cinit  : > FLASH, PAGE = 0, ALIGN(8)
.const  : > FLASH, PAGE = 0, ALIGN(8)
.econst : > FLASH, PAGE = 0, ALIGN(8)
.pinit  : > FLASH, PAGE = 0, ALIGN(8)
.switch : > FLASH, PAGE = 0, ALIGN(8)
}
```

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.

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 word and the length of data must correlate to the supplied ECC. That means, the data buffer length (in 8-bit bytes) should be 8 when the ECC buffer length is 1 byte and the data buffer length (in 8-bit bytes) should be 16 when the ECC buffer length is 2 bytes. The supplied starting address to program at plus the data buffer length cannot cross the 128-bit aligned address boundary. 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 (if the starting address to program is 128-bit aligned) or 1 byte (if the starting address to program is a 128-bit aligned address+8). Arguments two and three are ignored when using this mode.

NOTE: The length of pu8DataBuffer and pu8EccBuffer cannot exceed 16 and 2, respectively.

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

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 FMC to complete the program operation before returning to any kind of Flash accesses. The Fapi_checkFsmForReady() function can 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 as needed.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (failure: Data buffer size specified is incorrect)
- **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)

Sample Implementation

Please refer to the example provided in controlSUITE at below location:

For TMS320F28M35x:
ti\controlSUITE\device_support\f28m35x\vx\F28M35x_examples_Dual\flash_prog

For TMS320F28M36x:
ti\controlSUITE\device_support\f28m36x\vx\F28M36x_examples_Dual\flash_prog

3.2.3.2 For C28x devices

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	
pu16DataBuffer [in]	Pointer to the Data buffer address	
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 8.

Table 8. Uses of Different Programming Modes

Programming mode (oMode)	Arguments used	Usage purpose
Fapi_DataOnly	pu32StartAddress, pu16DataBuffer, pu16DataBufferSizeInWords	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 Single Error Correction and Double Error Detection (SECDED) module at run time. In such case, ECC is calculated separately (using either the ECC calculation algorithm provided in Section E.1.2 or 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. Also, this mode is used when not all the 64 bits of the data is available to program along with ECC. In this case, users can program less than 64 bits without ECC using this mode. And then later program all the 64 bits when available along with ECC. However, note that in subsequent devices, it is restricted to program all the 64-bits of data at a time and hence TI suggests to program all the 64 bits at a time if possible.
Fapi_AutoEccGeneration	pu32StartAddress, pu16DataBuffer, pu16DataBufferSizeInWords	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.
Fapi_DataAndEcc	pu32StartAddress, pu16DataBuffer, pu16DataBufferSizeInWords, pu16EccBuffer, pu16EccBufferSizeInBytes	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, pu16EccBufferSizeInBytes	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. 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 portion in Flash along with automatically generated ECC. The ECC is calculated for every 64-bit data aligned on a 64-bit memory boundary. All the 64-bits of the data must be supplied. Data not supplied is assumed as all 1s (0xFFFF). 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.

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, PAGE = 0, ALIGN(4)
.cinit  : > FLASH, PAGE = 0, ALIGN(4)
.const  : > FLASH, PAGE = 0, ALIGN(4)
.econst : > FLASH, PAGE = 0, ALIGN(4)
.pinit  : > FLASH, PAGE = 0, ALIGN(4)
.switch : > FLASH, PAGE = 0, 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.

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 word and the length of data must correlate to the supplied ECC. That means, the data buffer length (in 16-bit words) should be 4 when the ECC buffer length is 1 byte and the data buffer length (in 16-bit words) should be 8, when the ECC buffer length is 2 bytes. The supplied starting address to program at plus the data buffer length cannot cross the 128-bit aligned address boundary.

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 igmored when using this mode.

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

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

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 FMC to complete the program operation before returning to any kind of Flash accesses. The Fapi_checkFsmForReady() function can 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.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (failure: Data buffer size specified is incorrect)
- **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)

Sample Implementation

Please refer to the example provided in controlSUITE at below location:

For TMS320F28M35x: ti\controlSUITE\device_support\f28m35x\vx\F28M35x_examples_Dual\flash_prog

For TMS320F28M36x: ti\controlSUITE\device_support\f28m36x\vx\F28M36x_examples_Dual\flash_prog

3.2.4 Fapi_issueProgrammingCommandForEccAddresses()

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

3.2.4.1 For ARM Cortex devices

Synopsis

```
Fapi_StatusType Fapi_issueProgrammingCommandForEccAddresses(
        uint32 *pu32StartAddress,
        uint8  *pu8EccBuffer,
        uint8   u8EccBufferSizeInBytes)
```

Parameters

pu32StartAddress [in]	ECC start address in Flash ECC space for the ECC to be programmed
pu8EccBuffer [in]	pointer to the ECC buffer address
u8EccBufferSizeInBytes [in]	number of bytes in the ECC buffer

Description

This function will remap an address in the ECC memory space to the corresponding Flash main array 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.

NOTE: The length of pu8EccBuffer cannot exceed 2.

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

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: ECC buffer size specified is incorrect)

3.2.4.2 For C28x devices

Synopsis

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

Parameters

pu32StartAddress [in]	ECC start address in Flash ECC space 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 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 limitations given 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 FMSTAT after issuing the program command. The user application must check the FMSTAT value when FSM has completed the program operation. FMSTAT indicates if there is any failure occurrence during the program operation. The user application can use the Fapi_getFSMStatus function to obtain the FMSTAT value.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: ECC buffer size specified is incorrect)

3.2.5 Fapi_issueFsmSuspendCommand()

Issues Flash State Machine suspend command

Synopsis

```
Fapi_StatusType Fapi_issueFsmSuspendCommand(void)
```

Parameters

None

Description

This function issues a Suspend Now command which will suspend the FSM commands Program Data, and Erase Sector when they are the current active command. Use Fapi_getFsmStatus() to determine if the operation is successful.

Return Value

- **Fapi_Status_Success** (success)

3.2.6 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

Description

This function issues a command to the Flash State Machine for commands not requiring any additional information (such as address). Typical commands are Clear Status, Program Resume, Erase Resume and Clear_More. This function does not wait until the command is over; it just issues the command and returns back. Hence, the user application must wait for the FMC to complete the given command before returning to any kind of Flash accesses. The Fapi_checkFsmForReady() function can be used to monitor the status of an issued command.

Below are the details of these commands:

- Fapi_ClearStatus: Executing this command clears the ILA, PGV, EV, CSTAT, VOLTSTAT, and INVDAT bits in the FMSTAT register. Flash API issues this command before issuing a program or an erase command.
- Fapi_ClearMore: Executing this command clears everything the Clear Status command clears and additionally, clears the ESUSP and PSUSP bits in the FMSTAT register.
- Fapi_ProgramResume: Executing this command will resume the previously suspended program operation. Issuing a resume command when suspend is not active has no effect. Note that a new program operation cannot be initiated while a previous program operation is suspended.
- Fapi_EraseResume: Executing this command will resume the previously suspended erase operation. Issuing a resume command when suspend is not active has no effect. Note that a new erase operation cannot be initiated while a previous erase operation is suspended.

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

Return Value

- **Fapi_Status_Success** (success)

3.2.7 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. 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)

3.2.8 Fapi_getFsmStatus()

Returns the value of the FMSTAT register

Synopsis

```
Fapi_FlashStatusType Fapi_getFsmStatus(void)
```

Parameters

None

Description

This function returns the value of the FMSTAT register. This register allows the user application to determine whether an erase or program operation is successfully completed, in progress, 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 9. FMSTAT Register

Bits 3 1	...	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rsvd			ILA	Rsvd	PGV	Rsvd	EV	Rsvd	Busy	ERS	PGM	INV DAT	CSTAT	Volt Stat	ESUSP	PSUSP	Rsvd

Table 10. FMSTAT Register Field Descriptions

Bit	Field	Description
31-15	RSVD	Reserved
14	ILA	Illegal Address. When set, indicates that an illegal address is detected. The conditions below can set an illegal address flag: • Writing to an address location in an unimplemented flash space. • The address range does not match the type of FSM command.
13	RSVD	Reserved
12	PGV	Program verify. When set, indicates that a word is not successfully programmed, even after the maximum allowed number of program pulses are given for program operation.
11	RSVD	Reserved
10	EV	Erase verify. When set, indicates that a sector is not successfully erased, even after the maximum allowed number of erase pulses are given for erase operation. During Erase verify command, this flag is set immediately if a bit is found to be 0.
9	RSVD	Reserved
8	Busy	When set, this bit indicates that a program, erase, or suspend operation is being processed.
7	ERS	Erase Active. When set, this bit indicates that the flash module is actively performing an erase operation. This bit is set when erasing starts and is cleared when erasing is complete. It is also cleared when the erase is suspended and set when the erase resumes.
6	PGM	Program Active. When set, this bit indicates that the flash module is currently performing a program operation. This bit is set when programming starts and is cleared when programming is complete. It is also cleared when programming is suspended and set when programming resumes.
5	INVDAT	Invalid Data. When set, this bit indicates that the user attempted to program a “1” where a “0” was already present. This bit is cleared by the Clear Status command.
4	CSTAT	Command Status. Once the FSM starts, any failure will set this bit. When set, this bit informs the host that the program or erase command failed and the command was stopped. This bit is cleared by the Clear Status command. For some errors, this will be the only indication of an FSM error because the cause does not fall within the other error bit types.
3	VOLTSTAT	Core Voltage Status. When set, this bit indicates that the core voltage generator of the pump power supply dipped below the lower limit allowable during a program or erase operation. This bit is cleared by the Clear Status command.
2	ESUSP	Erase Suspend. When set, this bit indicates that the flash module has received and processed an erase suspend operation. This bit remains set until the erase resume command has been issued or until the Clear_More command is run.
1	PSUSP	Program Suspend. When set, this bit indicates that the flash module has received and processed a program suspend operation. This bit remains set until the program resume command has been issued or until the Clear_More command is run.
0	RSVD	RSVD

3.3 Read Functions

3.3.1 Fapi_doBlankCheck()

Verifies if the region specified is erased or not

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]	indicates read mode that failed blank check

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, these results will be returned in the poFlashStatusWord parameter. This will use normal read, read margin 0 and read margin 1 modes checking for blank.

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)

3.3.2 Fapi_doBlankCheckByByte()

Verifies if the region specified is erased or not. The CPU checks one byte at a time.

Synopsis

```
Fapi_StatusType Fapi_doBlankCheckByByte(
        uint8 *pu8StartAddress,
        uint32  u32Length,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

Parameters

pu8StartAddress [in]		start address for region to blank check
u32Length [in]		length of region in 8-bit bytes 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 0xFF)
	->au32StatusWord[3]	indicates read mode that failed blank check

Description

This function checks if the Flash is blank (erased state) starting at the specified address for the length of 8-bit bytes specified. If a non-blank location is found, these results will be returned in the poFlashStatusWord parameter. This will use normal read, read margin 0 and read margin 1 modes checking for blank.

Restrictions

This function is not applicable for C28x cores.

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)

3.3.3 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
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]	indicates read mode that failed verify

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. This will use normal read, read margin 0 and read margin 1 modes for verifying the data.

Return Value

- **Fapi_Status_Success** (success: region specified matches supplied data)
- **Fapi_Error_Fail** (failure: region specified does not match supplied data)

3.3.4 Fapi_doVerifyByByte()

Verifies region specified against supplied data by byte

Synopsis

```
Fapi_StatusType Fapi_doVerifyByByte(
        uint8 *pu8StartAddress,
        uint32  u32Length,
        uint8 *pu8CheckValueBuffer,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

Parameters

pu8StartAddress [in]		start address for region to verify by byte
u32Length [in]		length of region in 8-bit bytes to verify
pu8CheckValueBuffer [in]		address of buffer to verify region against by byte
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]	indicates read mode that failed verify

Description

This function verifies the device against the supplied data by byte starting at the specified address for the length of 8-bit bytes specified. If a location fails to compare, these results will be returned in the poFlashStatusWord parameter. This will use normal read, read margin 0, and read margin 1 modes checking for verifying the data.

Restrictions

This function is not applicable for C28x cores.

Return Value

- **Fapi_Status_Success** (success: region specified matches supplied data)
- **Fapi_Error_Fail** (failure: region specified does not match supplied data)

3.3.5 Fapi_doMarginRead()

Reads the specified Flash Memory region using the specified margin mode and returns the data in a user-given buffer

Synopsis

```
Fapi_StatusType Fapi_doMarginRead(
        uint32 *pu32StartAddress,
        uint32 *pu32ReadBuffer,
        uint32  u32Length,
        Fapi_FlashReadMarginModeType oReadMode)
```

Parameters

pu32StartAddress [in]	start address for region to read
pu32ReadBuffer [out]	address of buffer to return read data
u32Length [in]	length of region in 32-bit words to read
oReadMode [in]	indicates which margin mode (normal, RM0, RM1) to use

Description

This function reads the region specified starting at pu32StartAddress for u32Length 32-bit words and stores the read values in pu32ReadBuffer.

Return Value

- **Fapi_Status_Success** (success: specified memory range is read and data is returned)
- **Fapi_Error_InvalidReadMode** (failure: read mode specified is not valid)

3.3.6 Fapi_doMarginReadByByte()

Reads the specified Flash Memory region using the specified margin mode by byte and returns the data in a user-given buffer.

Synopsis

```
Fapi_StatusType Fapi_doMarginReadByByte(
        uint8 *pu8StartAddress,
        uint8 *pu8ReadBuffer,
        uint32  u32Length,
        Fapi_FlashReadMarginModeType oReadMode)
```

Parameters

pu8StartAddress [in]	start address for region to read by byte
pu8ReadBuffer [out]	address of buffer to return read data by byte
u32Length [in]	length of region in 8-bit bytes to read
oReadMode [in]	indicates which margin mode (normal, RM0, RM1) to use

Description

This function reads the region specified starting at pu8StartAddress for u32Length 8-bit bytes and stores the read values in pu8ReadBuffer.

Restrictions

This function is not applicable for C28x cores.

Return Value

- **Fapi_Status_Success** (success: specified memory range is read and data is returned)
- **Fapi_Error_InvalidReadMode** (failure: read mode specified is not valid)

3.3.7 Fapi_calculatePsa()

Calculates the PSA for a specified region

Synopsis

```
Uint32 Fapi_calculatePsa(
         uint32 *pu32StartAddress,
         uint32  u32Length,
         uint32  u32PsaSeed,
         Fapi_FlashReadMarginModeType oReadMode)
```

Parameters

pu32StartAddress [in]	start address for region to calculate PSA value
u32Length [in]	length of region in 32-bit words to calculate PSA value
u32PsaSeed [in]	seed value for PSA calculation
oReadMode [in]	indicates which margin mode (normal, RM0, RM1) to use

Description

This function calculates the PSA value for the region specified starting at pu32StartAddress for u32Length 32-bit words using u32PsaSeed value in the margin mode specified. The PSA algorithm is given in Appendix D.

Return Value

- PSA value

3.3.8 Fapi_doPsaVerify()

Verifies region specified against specified PSA value

Synopsis

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

Parameters

pu32StartAddress [in]		start address for region to verify PSA value
u32Length [in]		length of region in 32-bit words to verify PSA value
u32PsaValue [in]		PSA value to compare region against
poFlashStatusWord [out]		returns the status of the operation if result is not Fapi_Status_Success
	->au32StatusWord[0]	Actual PSA for read-margin 0
	->au32StatusWord[1]	Actual PSA for read-margin 1
	->au32StatusWord[2]	Actual PSA for normal read

Description

This function verifies the device against the supplied PSA value starting at the specified address for the length of 32-bit words specified. The calculated PSA values for all 3 margin modes are returned in the poFlashStatusWord parameter.

Return Value

- **Fapi_Status_Success** (success: region specified matches supplied PSA value)
- **Fapi_Error_Fail** (failure: region specified does not match supplied PSA value)

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 1.
- u8ApiMinorVersion – Minor version number of this compile of the API. Minor version is 52 for F28M35x and F28M36x devices.
- u8ApiRevision – Revision version number of this compile of the API
- oApiProductionStatus – Production status of this compile *(Alpha_Internal, Alpha, Beta_Internal, Beta, Production)*
- u32ApiBuildNumber – Build number of this compile. Used to differentiate between different alpha and beta builds
- u8ApiTechnologyType – Indicates the Flash technology supported by the API. Technology type used in these devices is of type 0x4.
- u8ApiTechnologyRevision – Indicates the revision of the Technology supported by the API
- u8ApiEndianness – Always returns a value of 1 (Little Endian)
- 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)

3.4.2 Fapi_getDeviceInfo()

Returns information specific to the device on which the code is being executed

Synopsis

```
Fapi_DeviceInfoType Fapi_getDeviceInfo(void)
```

Parameters

None

Description

This function returns information about the specific device on which the Flash API library is being executed. The information is returned in a struct Fapi_DeviceInfoType. The members are as follows:

- u16NumberOfBanks – Number of banks for this FMC
- u16DevicePackage – Device package pin count
- u16DeviceMemorySize – Flash memory size for this FMC in KB
- u32AsicId – N/A
- u32LotNumber – N/A
- u16FlowCheck – N/A
- u16WaferNumber – N/A
- u16WaferXCoordinate – N/A
- u16WaferYCoordinate – N/A

Restrictions

This function is deprecated and not supported in subsequent devices. Therefore, TI suggest to not use this function.

Return Value

- **Fapi_DeviceInfoType** (gives the above information retrieved about the device)

3.4.3 Fapi_getBankSectors()

Returns the sector information for the requested bank

Synopsis

```
Fapi_StatusType Fapi_getBankSectors(
             Fapi_FlashBankType oBank,
             Fapi_FlashBankSectorsType *poFlashBankSectors)
```

Parameters

oBank [in]	Bank on which to get information. Use Fapi_FlashBank0.
poFlashBankSectors [out]	Returned structure with the bank information

Description

This function returns information about the bank starting address, number of sectors, sector sizes, and bank technology type. The information is returned in a struct Fapi_FlashBankSectorsType. The members are as follows:

- oFlashBankTech – N/A
- u32NumberOfSectors – Indicates the number of sectors in the bank.
- u32BankStartAddress – Starting address of the bank.
- au8SectorSizes[] – An array of sectors sizes for each sector in the bank.

Sector size returned by Fapi_getBankSectors() function can be decoded as shown below:

Sector size value returned by Fapi_getBankSectors()	Corresponding Flash sector size
0x08	16K
0x10	32K
0x20	64K
0x40	128K

Restrictions

This function is deprecated and not supported in subsequent devices. Therefore, TI suggest to not use this function.

This function returns the sector information for the maximum flash bank size configuration in any given family. For example, in F28M36x devices, some PART numbers will have 1MB Flash in M3 subsystem and some PART numbers will have 512KB Flash in M3 subsystem. However, this function is hardcoded to return the sector information, assuming the Flash size as 1MB (max Flash size in F28M36x M3 subsystem).

Return Value

- **Fapi_Status_Success** (success: requested data is provided)
- **Fapi_Error_FeatureNotAvailable** (failure: not all devices have this support in the Flash Memory Controller)
- **Fapi_Error_InvalidBank** (failure: bank does not exist on this device)

3.5 Utility Functions

3.5.1 Fapi_flushPipeline()

Flushes the FMC pipeline buffers

Synopsis

```
void Fapi_flushPipeline(void)
```

Parameters

None

Description

This function flushes the FMC 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 data

Synopsis

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

Parameters

u32Address [in]	Address of the 64-bit data for which ECC has to be calculated
u64Data [in]	64-bit data for which ECC has to be calculated (data should be in little-endian order)

Description

This function will calculate the ECC for a 64-bit aligned word including address. For C28x, note that the user application should left-shift the address by 1 position before passing to this function. Left-shifting of address is not required for M3.

Return Value

- 8-bit calculated ECC (For C28x, the upper 8 bits of the 16-bit return value should be ignored)

3.5.3 Fapi_isAddressEcc()

Indicates whether or not an address is in the Flash ECC memory 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** (Value of 0 -The address specified is not in ECC memory range)
- **TRUE** (Value of 1 -The address specified is in ECC memory range)

3.5.4 Fapi_remapEccAddress()

Takes the 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 from which to start calculating the checksum
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.

Return Value

- 32-bit Fletcher Checksum value and address

3.6 User Definable Functions

These functions are distributed in the file Fapi_UserDefinedFunctions.c. These are the base functions called by the API and can be modified to meet the user's need for these operations. This file must be compiled with the user's code.

3.6.1 Fapi_serviceWatchdogTimer()

This function services the Watchdog timer. Flash API does not configure (enable or disable) the Watchhdog. It is up to the user to decide whether Watchdog should be enabled or disabled during Flash API execution. Flash API is interruptible. Therefore, the user application can service the Watchdog via an ISR (for example, timer ISR) as needed, instead of using this function. However, ISR should be mapped in RAM since Flash should not be accessed when Flash API execution is in progress. Users should pay special attention to the **Description** and **Restrictions** of this function provided below.

Synopsis

```
Fapi_StatusType Fapi_serviceWatchdogTimer(void)
```

Parameters

None

Description

This function allows the user to service their Watchdog timer in the Read Functions, Table 3. This function is called in the Read functions when the address being read crosses the 256-word (16-bit words for C28x and 8-bit words for M3) aligned address boundaries.

NOTE: Users may modify the Fapi_serviceWatchdogTimer() function as needed, but must ensure that they include **EALLOW (for C28x) and MWRALLOW (for M3 to allow writes to protected registers)** before the return statement at the end of this function so that Flash API can write to protected registers as needed.

Restrictions

This function is deprecated and not supported in subsequent devices. Therefore, TI suggests to not use this function.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

For M3:

```
Fapi_StatusType Fapi_serviceWatchdogTimer(void)
{
   /* Users to add their own watchdog servicing code here */
   .
   .
   .

   //
   // Allow writes to protected registers
   //
   HWREG(SYSCTL_MWRALLOW) =  0xA5A5A5A5;

   return(Fapi_Status_Success);
}
```

For C28x:

```
Fapi_StatusType Fapi_serviceWatchdogTimer(void)
{
```

```
    /* Users to add their own watchdog servicing code here */
    .
    .
    .

    //
    // Allow writes to protected registers
    //
    EALLOW;

    return(Fapi_Status_Success);
}
```

3.6.2 Fapi_setupEepromSectorEnable()

Sets up the sectors available on the EEPROM bank for erase and programming. However, note that users should not edit the contents of this function and should be used as provided by TI. These functions are left in Fapi_UserDefinedFunctions.c to keep source compatibility across TI devices that use similar Flash technology.

Synopsis

```
Fapi_StatusType Fapi_setupEepromSectorEnable(void)
```

Parameters

None

Description

This function sets up the sectors in the EEPROM bank (does not exist in TMS320F28M35x/6x devices) that are available for erase and programming operations.

Restrictions

This function is deprecated and not supported in subsequent devices (but users should not remove or edit this function in TMS320F28M35x/6x devices).

Return Value

- **Fapi_Status_Success** (success)

3.6.3 Fapi_setupBankSectorEnable()

Sets up the sectors available on the bank for erase and programming

Synopsis

```
Fapi_StatusType Fapi_setupBankSectorEnable(void)
```

Parameters

None

Description

This function sets up the sectors in the bank that are available for erase and programming operations.

Restrictions

Note that users should not edit the contents of this function even though it is provided in the Fapi_UserDefinedFunctions.C file. This function should be used as provided by TI. The reason TI provides this function outside of the API Library is to keep source compatibility across TI devices where applicable. This function is deprecated and not supported in subsequent devices, but users should not remove or edit this function in TMS320F28M35x/6x devices.

Return Value

- **Fapi_Status_Success** (success)

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

The following diagram describes the flow for erasing a sector(s). Please refer to Section 3.2.2 for further information.

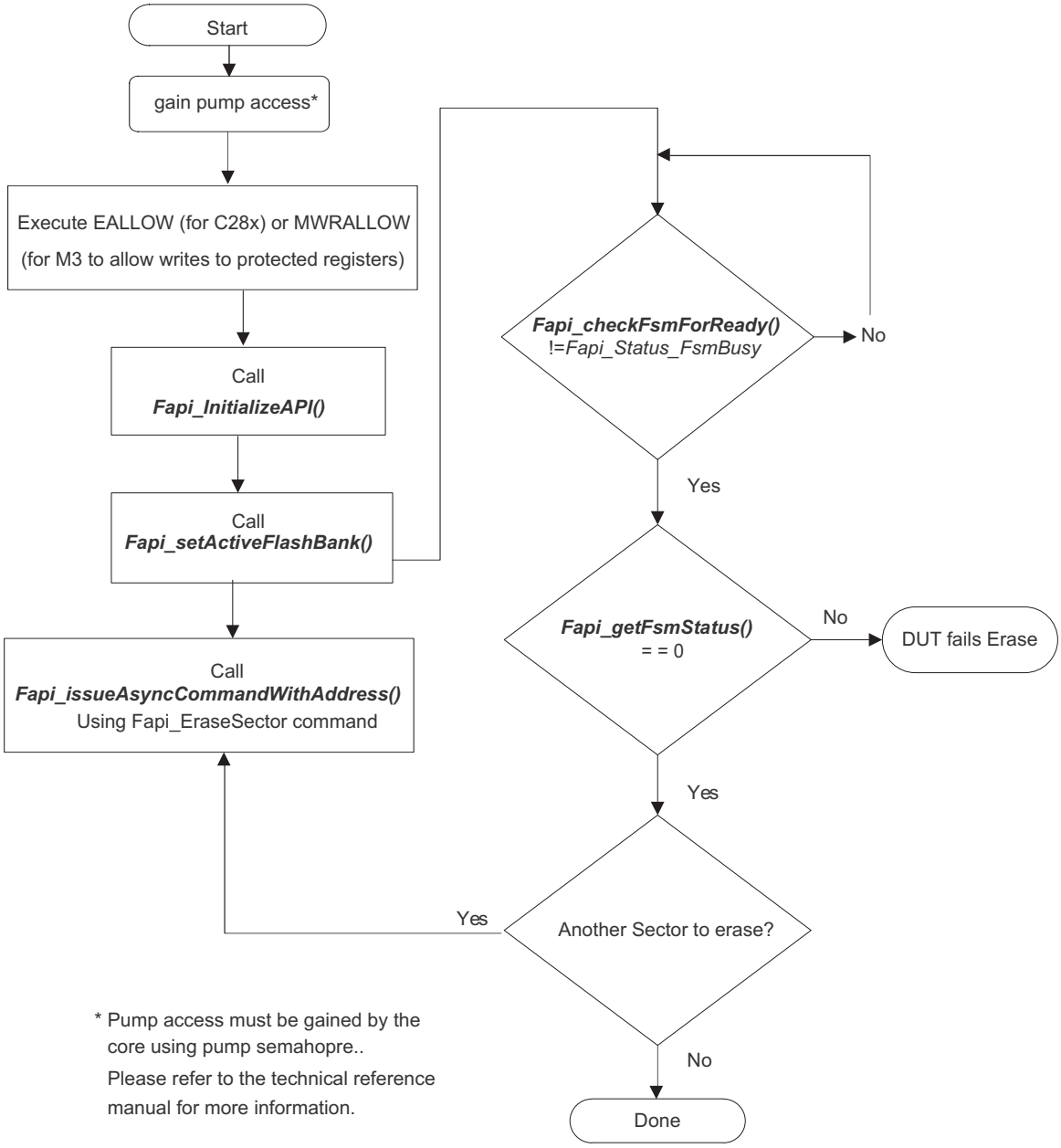

Figure 1. Recommended Erase Flow

4.3 Recommended Program Flow

The following diagram describes the flow for programming a device. This flow assumes the user has already erased all affected sectors following the Section 4.2. Please refer to Section 3.2.3 for further information.

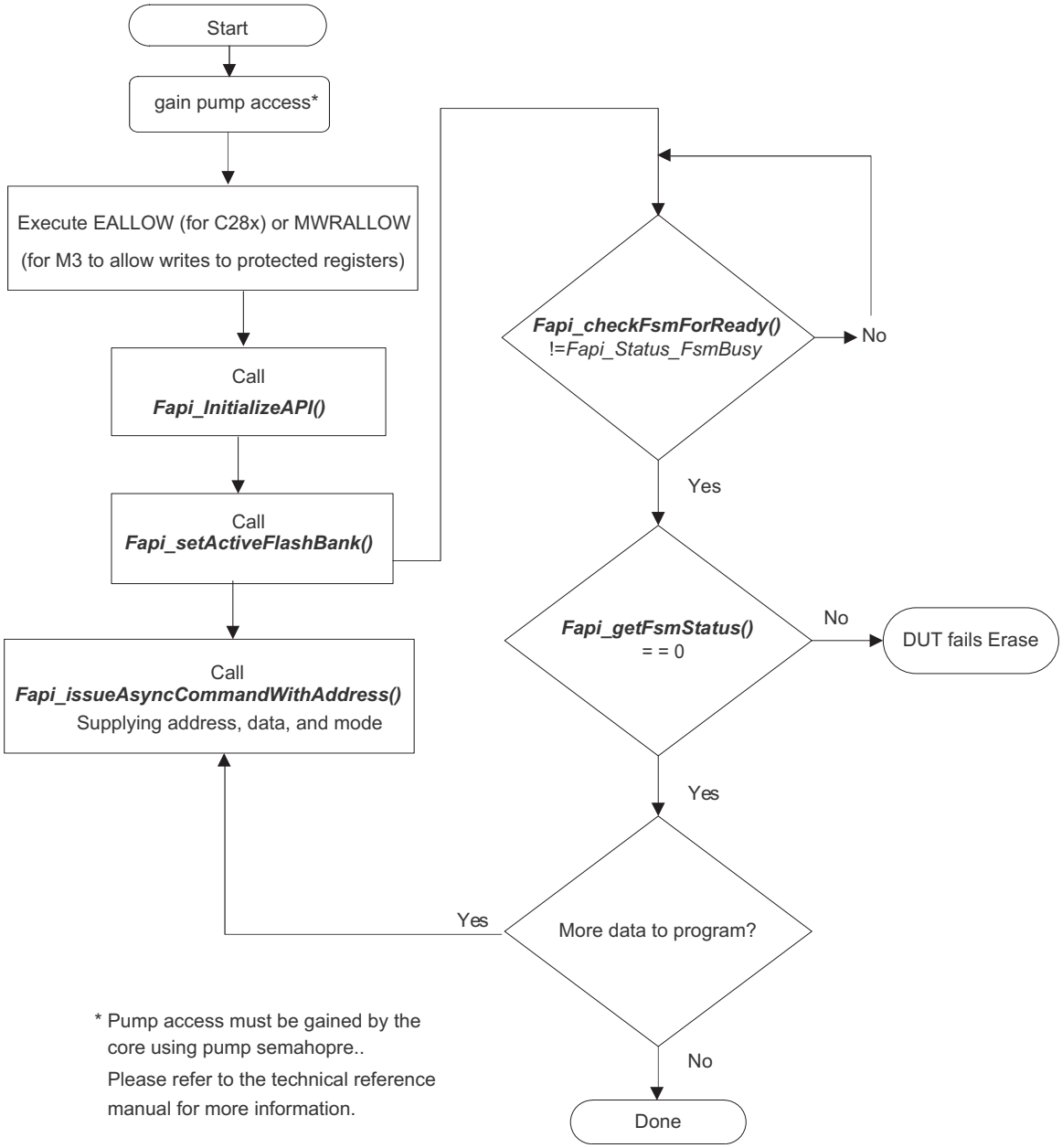

Figure 2. Recommended Program Flow

Flash State Machine Commands

A.1 Flash State Machine Commands

Table 11. Flash State Machine Commands

Command	Description	Enumeration Type	API Call(s)
Program Data	Used to program data to any valid Flash address	Fapi_ProgramData	Section 3.2.3
Erase Sector	Used to erase a Flash sector located by the specified address	Fapi_EraseSector	Fapi_issueAsyncCommandWithAddress()
Clear Status	Clears the status register	Fapi_ClearStatus	Fapi_issueAsyncCommand()
Program Resume	Resumes a suspended programming operation	Fapi_ProgramResume	Fapi_issueAsyncCommand()
Erase Resume	Resumes a suspended erase operation	Fapi_EraseResume	Fapi_issueAsyncCommand()
Clear More	Clears the status register	Fapi_ClearMore	Fapi_issueAsyncCommand()

Object Library Function Information

B.1 ARM CortexM3 Library

Table 12. ARM CortexM3 Function Sizes and Stack Usage

Function Name	Size In Bytes	Worst Case Stack Usage
Fapi_calculateEcc	20	0
Fapi_calculateFletcherChecksum	52	16
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc • Fapi_serviceWatchdogTimer	284	64
Fapi_checkFsmForReady	18	0
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	402	80
Fapi_doBlankCheckByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	252	64
Fapi_doMarginRead *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	352	48
Fapi_doMarginReadByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	268	48
Fapi_doPsaVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	390	64

Table 12. ARM CortexM3 Function Sizes and Stack Usage (continued)

Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	550	104
Fapi_doVerifyByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	392	96
Fapi_flushPipeline *Includes references to the following functions* • Fapi_waitDelay	78	16
Fapi_getBankSectors	138	16
Fapi_getDeviceInfo	78	32
Fapi_getFsmStatus	8	0
Fapi_getLibraryInfo	60	24
Fapi_initializeAPI	72	0
Fapi_isAddressEcc	58	0
Fapi_issueAsyncCommand	90	8
Fapi_issueAsyncCommandWithAddress *Includes references to the following functions* • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	172	24
Fapi_issueFsmSuspendCommand	56	0
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	680	64
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable • Fapi_remapEccAddress	754	88
Fapi_remapEccAddress	46	0
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	918	144
Fapi_serviceWatchdogTimer[(1)]	?	?
Fapi_setupBankSectorEnable		
Fapi_setupEepromSectorEnable		

(1) As this is a user modifiable function, this information is variable and dependent on the user's code

B.2 C28x Library

Table 13. C28x Function Sizes and Stack Usage

Function Name	Size In Words	Worst Case Stack Usage

Table 13. C28x Function Sizes and Stack Usage (continued)

Fapi_calculateEcc	19	0
Fapi_calculateFletcherChecksum	38	2
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc • Fapi_serviceWatchdogTimer	185	26
Fapi_checkFsmForReady	10	2
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	145	32
Fapi_doMarginRead *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	201	24
Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	263	42
Fapi_flushPipeline *Includes references to the following functions* • Fapi_waitDelay	45	8
Fapi_getBankSectors	151	10
Fapi_getDeviceInfo	37	14
Fapi_getFsmStatus	6	2
Fapi_getLibraryInfo	29	14
Fapi_initializeAPI	35	2
Fapi_isAddressEcc	25	2
Fapi_issueAsyncCommand	41	6
Fapi_issueAsyncCommandWithAddress *Includes references to the following functions* • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	96	8
Fapi_issueFsmSuspendCommand	23	2
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	442	30
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable • Fapi_remapEccAddress	497	40
Fapi_remapEccAddress	35	2

Table 13. C28x Function Sizes and Stack Usage (continued)

Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	594	118
Fapi_serviceWatchdogTimer[(1)]	?	?
Fapi_setupBankSectorEnable		
Fapi_setupEepromSectorEnable		

(1) As this is a user modifiable function, this information is variable and dependent on the user's code

Typedefs, defines, enumerations and structures

C.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;

typedef unsigned int        uint16_least;
typedef unsigned long int   uint32_least;

typedef signed int          sint16_least;
typedef signed long int     sint32_least;

typedef float                    float32;
typedef long double              float64;

#else

typedef unsigned char       boolean;

typedef unsigned char          uint8;
typedef unsigned short         uint16;
typedef unsigned int           uint32;
typedef unsigned long long int uint64;

typedef signed char           sint8;
typedef signed short          sint16;
typedef signed int            sint32;
typedef signed long long int sint64;

typedef unsigned int        uint8_least;
typedef unsigned int        uint16_least;
typedef unsigned int        uint32_least;

typedef signed int          sint8_least;
typedef signed int          sint16_least;
typedef signed int          sint32_least;

typedef float               float32;
typedef double              float64;

#endif
```

C.2 Enumerations

C.2.1 Fapi_CpuSelectorType

This can be used to indicate which CPU is being used.

```
typedef enum
{
   Fapi_MasterCpu,
   Fapi_SlaveCpu0
} ATTRIBUTE_PACKED Fapi_CpuSelectorType;
```

C.2.2 Fapi_CpuType

This can be used to indicate what type of CPU is being used.

```
typedef enum
{
   ARM7,
   M3,
   R4,
   R4F,
   C28,
   Undefined
}  ATTRIBUTE_PACKED Fapi_CpuType;
```

C.2.3 Fapi_FlashProgrammingCommandsType

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

```
typedef enum
{
Fapi_AutoEccGeneration, /* Command 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;
```

C.2.4 Fapi_FlashBankType

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

```
typedef enum
{
   Fapi_FlashBank0,
   Fapi_FlashBank1,  /* Not used for TMS320F28M35x/36x devices*/
   Fapi_FlashBank2,  /* Not used for TMS320F28M35x/36x devices*/
   Fapi_FlashBank3,  /* Not used for TMS320F28M35x/36x devices*/
   Fapi_FlashBank4,  /* Not used for TMS320F28M35x/36x devices*/
   Fapi_FlashBank5,  /* Not used for TMS320F28M35x/36x devices*/
   Fapi_FlashBank6,  /* Not used for TMS320F28M35x/36x devices*/
   Fapi_FlashBank7,  /* Not used for TMS320F28M35x/36x devices*/
}  ATTRIBUTE_PACKED Fapi_FlashBankType;
```

C.2.5 Fapi_FlashStateCommandsType

This contains all the possible Flash State Machine commands.

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

C.2.6 Fapi_FlashReadMarginModeType

This contains all the possible Flash State Machine commands.

```
typedef enum
{
   Fapi_NormalRead = 0x0,
   Fapi_RM0        = 0x1,
   Fapi_RM1        = 0x2
}  ATTRIBUTE_PACKED Fapi_FlashReadMarginModeType;
```

C.2.7 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_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 System Frequency and RWAIT */
   Fapi_Error_InvalidCpu,           /* Returned if the specified Cpu does not exist */
   Fapi_Error_InvalidBank,          /* Returned if the specified bank does not exist */
   Fapi_Error_InvalidAddress,       /* Returned if the specified Address does not exist in Flash
                                       or OTP */
   Fapi_Error_InvalidReadMode,      /* Returned if the specified read mode does not exist */
   Fapi_Error_AsyncIncorrectDataBufferLength,
   Fapi_Error_AsyncIncorrectEccBufferLength,
   Fapi_Error_AsyncDataEccBufferLengthMismatch,
   Fapi_Error_FeatureNotAvailable  /* FMC feature is not available on this device */
}  ATTRIBUTE_PACKED Fapi_StatusType;
```

C.2.8 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;
```

C.3 Structures

C.3.1 Fapi_EngineeringRowType

This is used to return the information from the engineering row in the TI OTP.

```
typedef struct
{
   uint32 u32AsicId;
   uint8  u8Revision;
   uint32 u32LotNumber;
   uint16 u16FlowCheck;
   uint16 u16WaferNumber;
   uint16 u16XCoordinate;
   uint16 u16YCoordinate;
}  ATTRIBUTE_PACKED Fapi_EngineeringRowType;
```

C.3.2 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;
```

C.3.3 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;
```

C.3.4 Fapi_DeviceInfoType

This is the structure used to return device information

```
typedef struct
{
#if defined(_LITTLE_ENDIAN)
   uint16 u16NumberOfBanks;
   uint16 u16Reserved;
   uint16 u16DeviceMemorySize;
   uint16 u16DevicePackage;
   uint32 u32AsicId;
   uint32 u32LotNumber;
   uint16 u16WaferNumber;
   uint16 u16FlowCheck;
   uint16 u16WaferYCoordinate;
   uint16 u16WaferXCoordinate;
#else
   uint16 u16Reserved;
   uint16 u16NumberOfBanks;
   uint16 u16DevicePackage;
   uint16 u16DeviceMemorySize;
   uint32 u32AsicId;
   uint32 u32LotNumber;
   uint16 u16FlowCheck;
   uint16 u16WaferNumber;
   uint16 u16WaferXCoordinate;
   uint16 u16WaferYCoordinate;
#endif
} Fapi_DeviceInfoType;
```

C.3.5 Fapi_FlashBankSectorsType

This gives the structure of a bank and technology type

```
typedef struct
{
    Fapi_FlashBankTechType oFlashBankTech;
    uint32 u32NumberOfSectors;
    uint32 u32BankStartAddress;
    uint8  au8SectorSizes[16];
} Fapi_FlashBankSectorsType;
```

Parallel Signature Analysis (PSA) Algorithm

D.1 Function Details

The functions Section 3.3.7 and Section 3.3.8 make use of the Parallel Signature Analysis (PSA) algorithm. Those functions are typically used to verify a particular pattern is programmed in the Flash Memory without transferring the complete data pattern. The PSA signature is based on this primitive polynomial:

```
f(X) = 1 + X + X^2 + X^22 + X^31

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

ECC Calculation Algorithm

E.1 Function Details

The function below can be used to calculate ECC for a given 64-bit aligned address (no need to left-shift the address) and the corresponding 64-bit data.

E.1.1 For M3

```
//
//Calculate the ECC for an address/data pair
//

uint8 CalcEcc(uint32 address, uint64 data)
{
                const uint32 addrSyndrome[8] =  {0x554ea, 0x0bad1, 0x2a9b5, 0x6a78d,
                                                 0x19f83, 0x07f80, 0x7ff80, 0x0007f};

                const uint64 dataSyndrome[8] =  {0xb4d1b4d14b2e4b2e, 0x1557155715571557,
                                                 0xa699a699a699a699, 0x38e338e338e338e3,
                                                 0xc0fcc0fcc0fcc0fc, 0xff00ff00ff00ff00,
                                                 0xff0000ffff0000ff, 0x00ffff00ff0000ff};
                const uint8 parity = 0xfc;

                uint64 xorData;
                uint32 xorAddr;
                uint8 bit, eccBit, eccVal;

                //
                //Extract bits "21:3" of the address
                //
                address = (address >> 3) & 0x7ffff;

                //
                //Compute the ECC one bit at a time.
                //
                eccVal = 0;
                for (bit = 0; bit < 8; bit++)
                {
                      //
                      //Apply the encoding masks to the address and data
                      //
                      xorAddr = address & addrSyndrome[bit];
                      xorData = data & dataSyndrome[bit];

                      //
                      //Fold the masked address into a single bit for parity calculation.
                      //The result will be in the LSB.
                      //
                      xorAddr = xorAddr ^ (xorAddr >> 16);
                      xorAddr = xorAddr ^ (xorAddr >> 8);
                      xorAddr = xorAddr ^ (xorAddr >> 4);
                      xorAddr = xorAddr ^ (xorAddr >> 2);
                      xorAddr = xorAddr ^ (xorAddr >> 1);
```

```
                        //
                        //Fold the masked data into a single bit for parity calculation.
                        //The result will be in the LSB.
                        //
                        xorData = xorData ^ (xorData >> 32);
                        xorData = xorData ^ (xorData >> 16);
                        xorData = xorData ^ (xorData >> 8);
                        xorData = xorData ^ (xorData >> 4);
                        xorData = xorData ^ (xorData >> 2);
                        xorData = xorData ^ (xorData >> 1);

                        //
                        //Merge the address and data, extract the ECC bit, and add it in
                        //
                        eccBit = ((uint8)xorData ^ (uint8)xorAddr) & 0x0001;
                        eccVal |= eccBit << bit;
                  }

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

E.1.2 For C28x

The LSB 8 bits of the return value from the below function contains ECC. The MSB 8 bits in the return value can be ignored.

```
//
//Calculate the ECC for an address/data pair
//

uint16 CalcEcc(uint32 address, uint64 data)
{
                  const uint32 addrSyndrome[8] =  {0x554ea, 0x0bad1, 0x2a9b5, 0x6a78d,
                                                   0x19f83, 0x07f80, 0x7ff80, 0x0007f};

                  const uint64 dataSyndrome[8] =  {0xb4d1b4d14b2e4b2e, 0x1557155715571557,
                                                   0xa699a699a699a699, 0x38e338e338e338e3,
                                                   0xc0fcc0fcc0fcc0fc, 0xff00ff00ff00ff00,
                                                   0xff0000ffff0000ff, 0x00ffff00ff0000ff};
                  const uint16 parity = 0xfc;

                  uint64 xorData;
                  uint32 xorAddr;
                  uint16 bit, eccBit, eccVal;

                  //
                  //Extract bits "20:2" of the address
                  //
                  address = (address >> 2) & 0x7ffff;

                  //
                  //Compute the ECC one bit at a time.
                  //
                  eccVal = 0;
                  for (bit = 0; bit < 8; bit++)
                  {
                        //
                        //Apply the encoding masks to the address and data
                        //
                        xorAddr = address & addrSyndrome[bit];
                        xorData = data & dataSyndrome[bit];
```

```
            //
            //Fold the masked address into a single bit for parity calculation.
            //The result will be in the LSB.
            //
            xorAddr = xorAddr ^ (xorAddr >> 16);
            xorAddr = xorAddr ^ (xorAddr >> 8);
            xorAddr = xorAddr ^ (xorAddr >> 4);
            xorAddr = xorAddr ^ (xorAddr >> 2);
            xorAddr = xorAddr ^ (xorAddr >> 1);

            //
            //Fold the masked data into a single bit for parity calculation.
            //The result will be in the LSB.
            //
            xorData = xorData ^ (xorData >> 32);
            xorData = xorData ^ (xorData >> 16);
            xorData = xorData ^ (xorData >> 8);
            xorData = xorData ^ (xorData >> 4);
            xorData = xorData ^ (xorData >> 2);
            xorData = xorData ^ (xorData >> 1);

            //
            //Merge the address and data, extract the ECC bit, and add it in
            //
            eccBit = ((uint16)xorData ^ (uint16)xorAddr) & 0x0001;
            eccVal |= eccBit << bit;
        }

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

Revision History

Changes from January 17, 2014 to July 1, 2017	Page
• This document has undergone extensive edits to remove the details related to non-C2000 devices and non-TMS320F28M35x/6x devices. The functions were reorgnized in a more detailed manner to add clarity. PSA algorithm and ECC algorithm appendices were also added	61

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265