TMS320F2838x Flash API

Version 1.60.00.00

Reference Guide

Literature Number: SPNU632
December 2019

TEXAS INSTRUMENTS

Contents

List of Figures

List of Tables

Reference Guide
SPNU632–December 2019

1 Introduction

This reference guide provides a detailed description of Texas Instruments' TMS320F2838x Flash API library functions that can be used to erase, program and verify Flash on TMS320F2838x device. Note that Flash API V1.60.xx.xx should be used only with TMS320F2838x devices.

The Flash API libraries are provided in *C2000Ware* at the locations below:

- C28x library: F2838x_C28x_FlashAPI.lib is available at C2000Ware_x_xx_xx_xx\libraries\flash_api\F2838x\c28x\lib folder.
- CM library: F2828x_CM_FlashAPI.lib is available at C2000Ware_x_xx_xx_xx\libraries\flash_api\F2838x\cm\lib.

1.1 Reference Material

Use this guide in conjunction with:

- *TMS320F2838x Microcontrollers Data Manual*
- *TMS320F2838x Microcontrollers Technical Reference Manual*

1.2 Function Listing Format

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

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

Description

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

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

Restrictions

Specifies any restrictions in using this function.

Return Value

Specifies any value or values returned by 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 or driverlib folder provided in C2000Ware, to demonstrate the usage of a given Flash API function in an application context.

2 TMS320F2838x Flash API Overview

2.1 Introduction

The Flash API is a library of routines, that when called with the proper parameters in the proper sequence, erases, programs, or verifies Flash memory. The Flash API can be used to program 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 and OTP Memory* chapter in the TRM. See *TMS320F2838x 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 Memory 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 ECC address to the main data space and then calls Fapi_issueProgrammingCommand() to program ECC
Fapi_issueFsmSuspendCommand()	Suspends FSM commands (program data and erase sector)
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 (FSM) 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 against erased state
Fapi_doBlankCheckByByte()[1]	Verifies specified Flash memory range against 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_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

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

Note that Fapi_getDeviceInfo() and Fapi_getBankSectors() are removed in TMS320F2838x Flash API since users can obtain this information (for example, number of banks, pin count, number of sectors, and so on) from other resources provided in the TRM.

The Fapi_UserDefinedFunctions.c file is not provided anymore since the functions in that file are now merged in the Flash API Library. Review Key Facts For Flash API Usage for information about servicing the watchdog function while using Flash API.

2.3 *Using API*

This section describes the flow for using various API functions

2.3.1 Initialization Flow

2.3.1.1 After Device Power Up

After the device is first powered up, the *Fapi_initializeAPI()* function must be called before any other API function (except for the *Fapi_getLibraryInfo()* function) can be used. This procedure 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 the *Fapi_getLibraryInfo()* function) can be used. This procedure will update the API internal state variables.

2.3.2 Building with the API

2.3.2.1 Object Library Files

The C28x Flash API and CM Flash API object files are distributed in the ARM standard EABI elf object format.

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 C2000Ware_x_xx_xx_xx\libraries\flash_api\f2838x\ folder:

- Library Files

  - F2838x_C28x_FlashAPI.lib – This is the Flash API object file (software API library) for the CPU1 and CPU2 subsystems in F2838x devices. This library is compiled with floating point support FPU32 option.
  - F2838x_CM_FlashAPI.lib – This is the Flash API object file (software API library) for the Connectivity Manager (CM) subsystem in F2838x devices.
  - Fixed point version of the API libraries are not provided.
- Include Files
  - These files set up compile-specific defines and then includes the F021.h master include file.
    - F021_F2838x_C28x.h – The master include file for TMS320F2838x C28x applications.
    - F021_F2838x_CM.h – The master include file for TMS320F2838x CM applications.
- 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 public API functions and includes all other include files.
  - Init.h – Defines the API initialization structure.
  - Registers_F2838x_C28x.h – Flash memory controller registers structure for C28x applications.
  - Registers_F2838x_CM.h – Flash memory controller registers structure for CM 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/F2838x.h – Constant definitions for F2838x devices.
  - Constants/F2838x_CM.h – Constant definitions for F2838x CM 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_”.
- 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 system clock frequency (SYSCLK for CPU1/CPU2 and CMCLK for CM) 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 execution is interruptible. However, there should not be any read or fetch access from the Flash bank on which erase or program operation is in progress. Therefore, the Flash API functions, the user application functions that call the Flash API functions, and any ISRs (Interrupt service routines,) must be executed from RAM. For example, the 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){}
```

- Flash API does not configure (enable/disable) watchdog. The user application can configure watchdog and service it as needed. Hence, the Fapi_ServiceWatchdogTimer() function is no longer provided.
- For C28x: Flash API uses EALLOW and EDIS internally as needed to allow/disallow writes to protected registers.
- For CM: Flash API unlocks and locks access internally as needed to allow/disallow writes to protected registers
- The Main Array flash and OTP programming must be aligned to 64-bit address boundaries (better is 128-bit alignment) and each 64-bit word may only be programmed once per write/erase cycle.
- It is permissible to program the data and ECC separately. However, each 64-bit dataword and the corresponding ECC word may only be programmed once per write/erase cycle.
- ECC should not be programmed for link-pointer locations. The API skips programming the ECC when the start address provided for the program operation is any of the three link-pointer addresses. API will use Fapi_DataOnly mode for programming these locations even if the user passes Fapi_AutoEccGeneration or Fapi_DataAndEcc mode as the programming mode parameter. The Fapi_EccOnly mode is not supported for programming these locations. The user application should exercise caution here. Care should be taken to maintain a separate structure/section for link-pointer locations in the application. Do not mix these fields with other DCSM OTP settings. If other fields are mixed with link-pointers, API will skip programming ECC for the non-link-pointer locations as well. This will cause ECC errors in the application.
- When using INTOSC as the clock source, a few SYSCLK frequency ranges need an extra wait state. Please refer to the data manual for more details.
- In order to avoid conflict between zone1 and zone2, a semaphore (FLSEM) is provided in the DCSM registers to configure Flash registers. The user application should configure this semaphore register before initializing the Flash and calling the Flash API functions. Please refer to *TMS320F2838x MicrocontrollersTechnical Reference Manual* for more details on this register.
- Note that the Flash API functions do not configure any of the DCSM registers. The user application should be sure to configure the required DCSM settings. For example, if a zone is secured, then Flash API should be executed from the same zone in order to be able to erase or program the Flash sectors of that zone. Or the zone should be unlocked. If not, Flash API’s writes to Flash registers will not succeed. Flash API does not check whether the writes to the Flash registers are going through or not.

It writes to them as required for the erase/program sequence and returns back assuming that the writes went through. This will cause the Flash API to return false success status. For example, Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, Address) when called, can return the success status but it does not mean that the sector erase is successful. Erase status should be checked using Fapi_getFSMStatus() and Fapi_doBlankCheck().

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
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 register value must be set before calling this function.

NOTE: Flash control register base address is hard coded in this function internally and does not use the value (first parameter passed to this function) provided by the user. User should still pass the parameter to avoid compile issues.

NOTE: The accuracy of the on-chip zero-pin oscillators (INTOSC1 or INTOSC2) will not meet the accuracy requirements for Flash erase and program operations. Hence, when using INTOSC as the PLL clock source, the value initialized by the user for the u32HclkFrequency parameter should be 3% more (should be rounded to the next highest integer) than the configured CMCLK. For example, when PLL is configured for a CMCLK of 121MHz with INTOSC as the PLL clock source, instead of initializing the u32HclkFrequency parameter as 121MHz, initialize it as 125MHz.

Above note applies only to CM but not for C28x. C28x Flash API takes care of it for C28x Flash API operations.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

Please refer to the example provided in C2000Ware at below location:

For C28x: C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\c28x\flash\

For CM: C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\cm\flash\

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

NOTE: Flash bank number is hard coded in this function internally and does not use the value (oNewFlashBank) provided by the user. User should still pass the parameter to avoid compile issues.

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 C2000Ware at below location:

For C28x: C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\c28x\flash\

For CM: C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\cm\flash\

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, this function always returns success status when the Fapi_EraseSector command is used. The user application must wait for the FMC to complete the erase operation before returning to any kind of Flash accesses. The Fapi_checkFsmForReady() function can be used to monitor the status of an issued command.

NOTE: This function does not check 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.

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

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_FeatureNotAvailable** (failure: User requested a command that is not supported)
- **Fapi_Error_FlashRegsNotWritable**(failure: Flash register write failed. The user should make sure that the API is executing from the same zone as that of the target address for flash operation OR the user should unlock before the flash operation)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

Sample Implementation

Please refer to the example provided in C2000Ware at below location:

For C28x: C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\c28x\flash\

For CM: C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\cm\flash\

3.2.3 Fapi_issueProgrammingCommand()

3.2.3.1 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. Data buffer should be 128-bit aligned.	
u16DataBufferSizeInWords [in]	Number of 16-bit words in the Data buffer	
pu16EccBuffer [in]	Pointer to the ECC buffer address	
u16EccBufferSizeInBytes [in]	Number of 8-bit bytes in the ECC buffer	
oMode [in]	Indicates the programming mode to use:	
	Fapi_DataOnly	Programs only the data buffer
	Fapi_AutoEccGeneration	Programs the data buffer and auto generates and programs the ECC.
	Fapi_DataAndEcc	Programs both the data and ECC buffers
	Fapi_EccOnly	Programs only the ECC buffer

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

Description

This function sets up the programming registers of the Flash State Machine based on the supplied parameters. It offers four different programming modes to the user for use in different scenarios as mentioned in Table 6.

Table 6. Uses of Different Programming Modes

Programming mode (oMode)	Arguments used	Usage purpose
Fapi_DataOnly	pu32StartAddress, pu16DataBuffer, u16DataBufferSizeInWords	Used when any custom programming utility or an user application (that embed/use Flash API) has to program data and corresponding ECC separately. Data is programmed using Fapi_DataOnly mode and then the ECC is programmed using Fapi_EccOnly mode. Generally, most of the programming utilities do not calculate ECC separately and instead use Fapi_AutoEccGeneration mode. However, some Safety applications may require to insert intentional ECC errors in their Flash image (which is not possible when Fapi_AutoEccGeneration mode is used) to check the health of the SECDED (Single Error Correction and Double Error Detection) module at run time. In such case, ECC is calculated separately (using 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.
Fapi_AutoEccGeneration	pu32StartAddress, pu16DataBuffer, u16DataBufferSizeInWords	Used when any custom programming utility or user application (that embed/use Flash API to program Flash at run time to store data or to do a firmware update) has to program data and ECC together without inserting any intentional errors. This is the most prominently used mode.
Fapi_DataAndEcc	pu32StartAddress, pu16DataBuffer, u16DataBufferSizeInWords, pu16EccBuffer, u16EccBufferSizeInBytes	Purpose of this mode is not different than that of using Fapi_DataOnly and Fapi_EccOnly modes together. However, this mode is beneficial when both the data and the calculated ECC can be programmed at the same time.
Fapi_EccOnly	pu16EccBuffer, u16EccBufferSizeInBytes	See the usage purpose given for Fapi_DataOnly mode.

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

Programming modes:

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

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

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

Here is an example:

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

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

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

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

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

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

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

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

Arguments 2 and 3 are ignored when using this mode.

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

NOTE: This function does not check 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.

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

This function does not wait until the program operation is over; it just issues the command and returns back. Hence, the user application must wait for the 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.
- The Main Array flash and OTP programming must be aligned to 64-bit address boundaries (better is 128-bit alignment) and each 64-bit word may only be programmed once per write/erase cycle.
- It is alright to program the data and ECC separately. However, each 64-bit dataword and the corresponding ECC word may only be programmed once per write or erase cycle.
- ECC should not be programmed for linkpointer locations. The API will issue the Fapi_DataOnly command for these locations even if the user chooses Fapi_AutoEccGeneration mode or Fapi_DataAndEcc mode. Fapi_EccOnly mode is not supported for linkpointer locations.

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)
- **Fapi_Error_FlashRegsNotWritable** (failure: Flash register writes failed. The user should make sure that the API is executing frm the same zone as that of the target address for flash operation OR the user should unlock before the flash operation.
- **Fapi_Error_FeatureNotAvailable** (failure: User passed a mode that is not supported)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)
- **Fapi_Error_InvalidCPUID** (failure: OTP has an invalid CPUID. Make sure that the Voltage rails are in valid range and the wait-states are configured properly. If the error persists, please contact TI)

Sample Implementation

Please refer to the example provided in C2000Ware at below location:

C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\c28x\flash\

3.2.3.2 For CM 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	
u8DataBufferSizeInBytes [in]	Number of bytes in the Data buffer	
pu8EccBuffer [in]	Pointer to the ECC buffer address	
u8EccBufferSizeInBytes [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, u8DataBufferSizeInWords	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.1.2 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.
Fapi_AutoEccGeneration	pu32StartAddress, pu8DataBuffer, u8DataBufferSizeInWords	Used when any custom programming utility or user application (that embed/use Flash API to program Flash at run time to store data or to do a firmware update) has to program data and ECC together without inserting any intentional errors. This is the most prominently used mode.

Table 7. Uses of Different Programming Modes (continued)

Programming mode (oMode)	Arguments used	Usage purpose
Fapi_DataAndEcc	pu32StartAddress, pu8DataBuffer, u8DataBufferSizeInWords, pu8EccBuffer, u8EccBufferSizeInBytes	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, u8EccBufferSizeInBytes	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 16 bytes. However, review the restrictions provided for this function to know the limitations of flash programming data size. The supplied starting address to program at plus the data buffer length cannot cross the 128-bit aligned address boundary. Arguments 4 and 5 are ignored when using this mode.

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

NOTE: Fapi_AutoEccGeneration mode will program the supplied data portion in Flash along with automatically generated ECC. The ECC is calculated for 64-bit aligned address and the corresponding 64-bit data. Any data not supplied is treated as 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.

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

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

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 2 and 3 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.

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

This function does not wait until the program operation is over; it just issues the command and returns back. Hence, the user application must wait for the 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.
- The Main Array flash programming and OTP must be aligned to 64-bit address boundaries (better is 128-bit alignment) and each 64-bit word may only be programmed once per write/erase cycle.

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)
- **Fapi_Error_FlashRegsNotWritable** (failure: Flash register writes failed. The user should make sure that the API is executing frm the same zone as that of the target address for flash operation OR the user should unlock before the flash operation.
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)
- **Fapi_Error_FeatureNotAvailable** (failure: User passed a mode that is not supported)

Sample Implementation

Please refer to the example provided in C2000Ware at below location:

C2000Ware_x_xx_xx_xx\driverlib\f2838x\examples\cm\flash\

3.2.4 Fapi_issueProgrammingCommandForEccAddresses()

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

3.2.4.1 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 same limitations for Fapi_issueProgrammingCommand() using Fapi_EccOnly mode applies to this function. The LSB of pu16EccBuffer corresponds to the lower 64 bits of the main array and the MSB of pu16EccBuffer corresponds to the upper 64 bits of the main array.

NOTE: The length of the pu16EccBuffer cannot exceed 2.

NOTE: This function does not check 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)
- **Fapi_Error_FlashRegsNotWritable**(failure: Flash register write failed. The user should make sure that the API is executing from the same zone as that of the target address for flash operation OR the user should unlock before the flash operation)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

3.2.4.2 For CM 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)
- **Fapi_Error_FlashRegsNotWritable** (failure: Flash register writes failed. The user should make sure that the API is executing frm the same zone as that of the target address for flash operation OR the user should unlock before the flash operation.
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

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 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)
- **Fapi_Error_FeatureNotAvailable** (failure: User passed a command that is not supported)

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. The primary use is to check if an Erase or Program operation has finished.

Return Value

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

3.2.8 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 8. FMSTAT Register

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

Table 9. FMSTAT Register Field Descriptions

Bit	Field	Description
31-13	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	Reserved

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]	N/A

Description

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

Return Value

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

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]	N/A

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, corresponding address and data will be returned in the poFlashStatusWord parameter.

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)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

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. Data buffer should be 128-bit aligned.
poFlashStatusWord [out]		returns the status of the operation if result is not Fapi_Status_Success
	->au32StatusWord[0]	address of first verify failure location
	->au32StatusWord[1]	data read at first verify failure location
	->au32StatusWord[2]	value of compare data
	->au32StatusWord[3]	N/A

Description

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

Return Value

- **Fapi_Status_Success** (success: region specified matches supplied data)
- **Fapi_Error_Fail** (failure: region specified does not match supplied data)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

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]	N/A

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.

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)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

3.3.5 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]	only normal mode is applicable. Use Fapi_NormalRead.

Description

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

Return Value

- **PSA value** (success)
- **0xA5A5A5A5U** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

3.3.6 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[3]	Actual PSA

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 is 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)
- **Fapi_Error_InvalidAddress** (failure: User provided an invalid address. Please refer Data Manual for the valid address range)

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 60 for F2838x 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)

3.5 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. There is no need to provide a left-shifted address to this function anymore. TMS320F2838x Flash API takes care of it.

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 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 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

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 Fapi_issueAsyncCommandWithAddress() for further information.

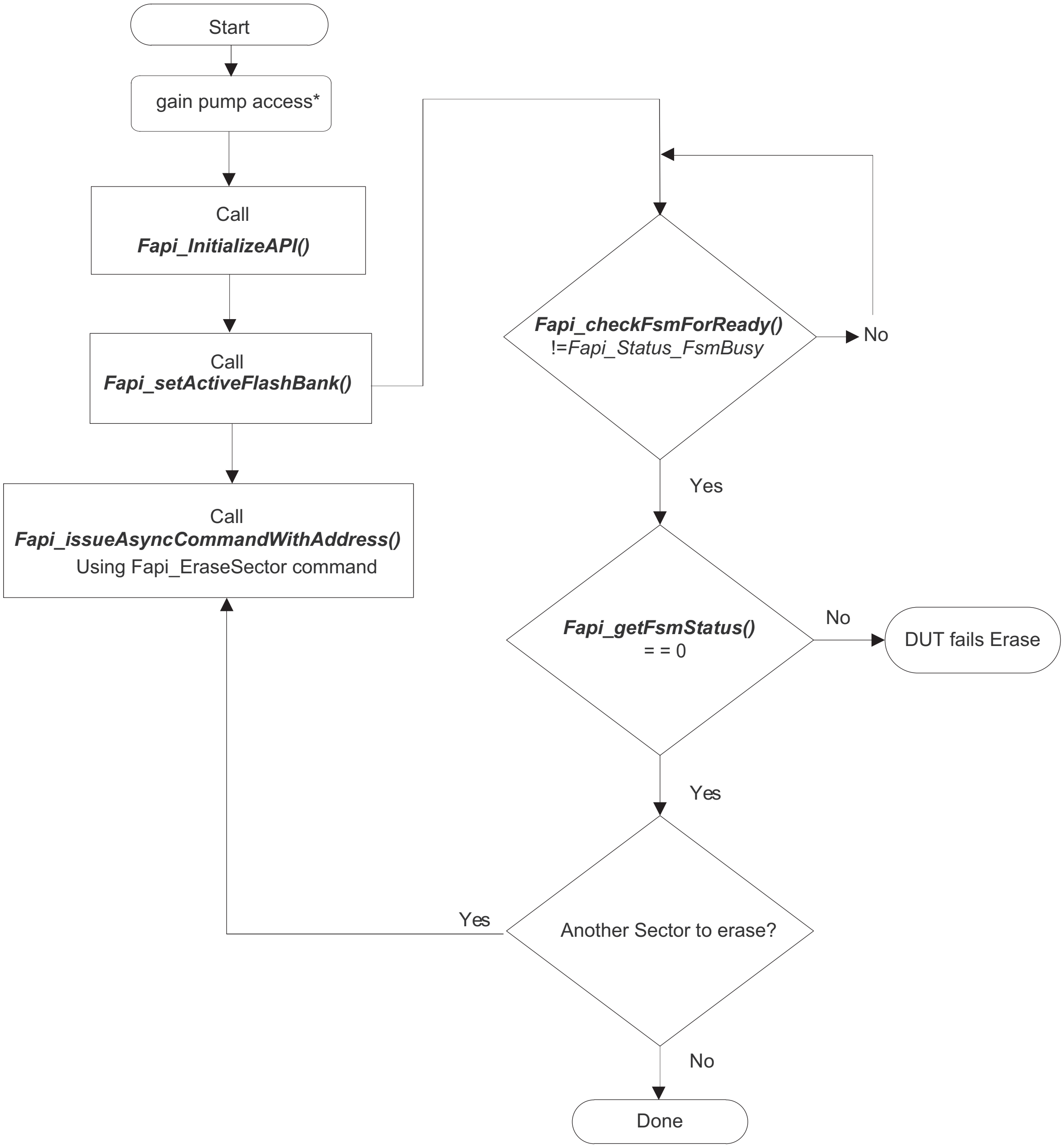

Figure 1. Recommended Erase Flow

NOTE: Pump access must be gained by the core using pump semaphore. Please refer to the technical reference manual for more information.

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 Recommended Erase Flow. Please refer to Fapi_issueProgrammingCommand() for further information.

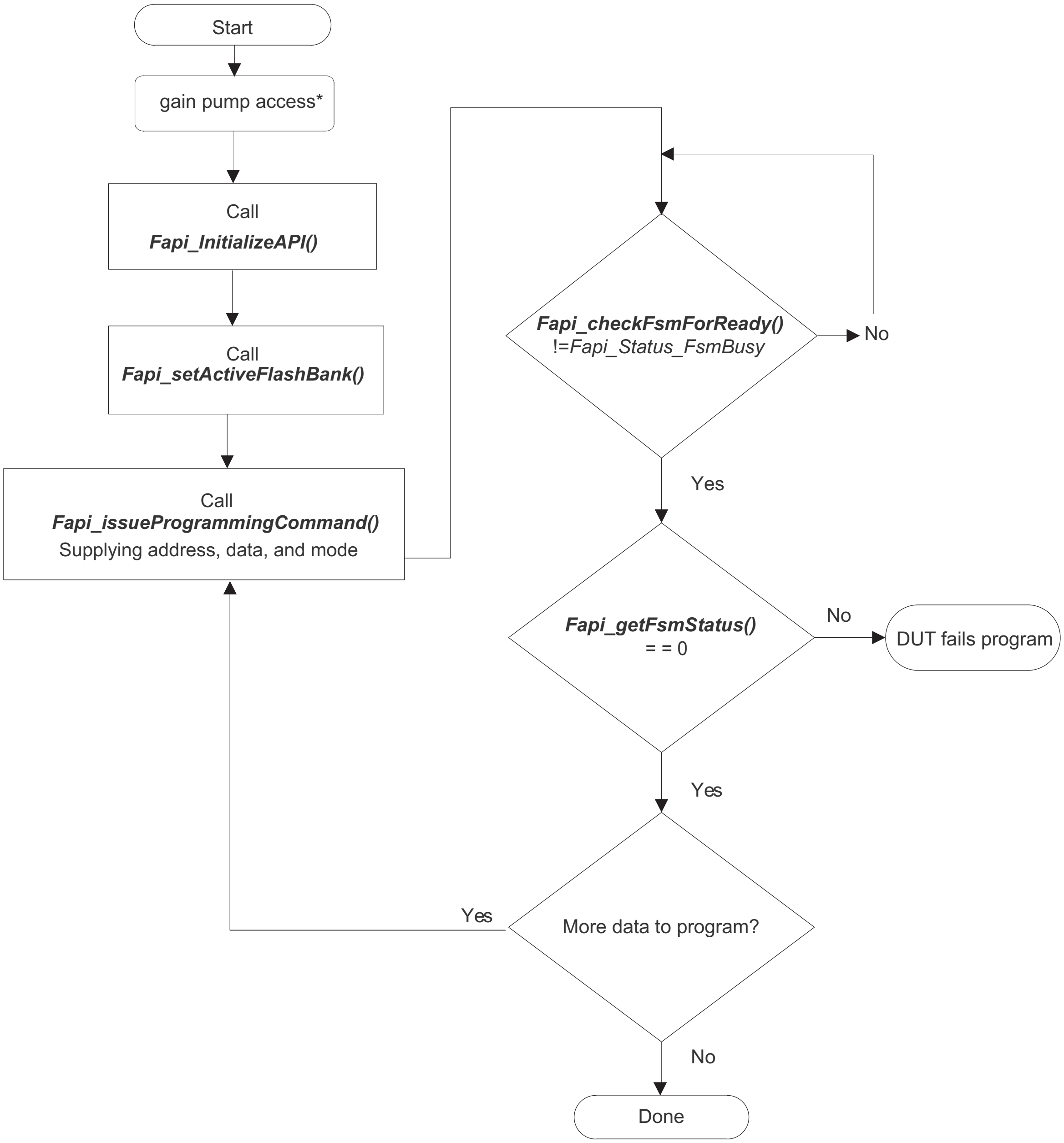

Figure 2. Recommended Program Flow

NOTE: Pump access must be gained by the core using pump semaphore. Please refer to the technical reference manual for more information.

Flash State Machine Commands

A.1 Flash State Machine Commands

Table 10. Flash State Machine Commands

Command	Description	Enumeration Type	API Call(s)

Program Data	Used to program data to any valid Flash address	Fapi_ProgramData	Fapi_issueProgrammingCommand()
			Fapi_issueProgrammingCommandForEccAddresses()

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()

Appendix B

SPNU632–December 2019

Object Library Function Information

B.1 TMS320F2838x C28x Flash API Library

Table 11. C28x Function Sizes and Stack Usage

Function Name	Size In Words	Worst Case Stack Usage
Fapi_calculateEcc	31	TBD
Fapi_calculateFletcherChecksum	44	TBD
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc	12	TBD
Fapi_checkFsmForReady	14	TBD
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_isAddressEcc	90	TBD
Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_isAddressEcc	15	TBD
Fapi_flushPipeline	21	TBD
Fapi_getFsmStatus	7	TBD
Fapi_getLibraryInfo	31	TBD
Fapi_initializeAPI	94	TBD
Fapi_isAddressEcc	34	TBD
Fapi_issueAsyncCommand	11	TBD
Fapi_issueAsyncCommandWithAddress	58	TBD
Fapi_issueFsmSuspendCommand	51	TBD
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc	414	TBD
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_remapEccAddress	21	TBD
Fapi_remapEccAddress	61	TBD
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	43	TBD

B.2 TMS320F2838x CM Flash API Library

Table 12. CM Function Sizes and Stack Usage

Function Name	Size In Bytes	Worst Case Stack Usage
Fapi_calculateEcc	78	TBD
Fapi_calculateFletcherChecksum	120	TBD
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc	34	TBD
Fapi_checkFsmForReady	26	TBD
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_isAddressEcc	208	TBD
Fapi_doBlankCheckByByte *Includes references to the following functions* • Fapi_flushPipeline	210	TBD
Fapi_doPsaVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_isAddressEcc	30	TBD
Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_isAddressEcc	34	TBD
Fapi_doVerifyByByte *Includes references to the following functions* • Fapi_flushPipeline	34	TBD
Fapi_flushPipeline	56	TBD
Fapi_getFsmStatus	14	TBD
Fapi_getLibraryInfo	70	TBD
Fapi_initializeAPI	174	TBD
Fapi_isAddressEcc	80	TBD
Fapi_issueAsyncCommand	24	TBD
Fapi_issueAsyncCommandWithAddress	148	TBD
Fapi_issueFsmSuspendCommand	92	TBD
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc	782	TBD
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_remapEccAddress	44	TBD
Fapi_remapEccAddress	128	TBD
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	104	TBD

Typedefs, defines, enumerations and structures

C.1 Type Definitions

C.1.1 For C28x

```
#if defined(__TMS320C28XX__)

typedef unsigned char       boolean;

typedef unsigned int           uint8; //This is 16 bits in C28x
typedef unsigned int           uint16;
typedef unsigned long int      uint32;
typedef unsigned long long int uint64;

#endif
```

C.1.2 For CM

```
typedef unsigned char       boolean;

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

C.2 Defines

```
#ifndef TRUE
   #define TRUE              1
#endif

#ifndef FALSE
   #define FALSE             0
#endif
```

C.3 Enumerations

C.3.1 Fapi_FlashProgrammingCommandsType

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

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

C.3.2 Fapi_FlashBankType

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

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

C.3.3 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.3.4 Fapi_FlashReadMarginModeType

This contains all the possible Flash State Machine commands.

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

C.3.5 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 */
   Fapi_Error_FlashRegsNotWritable, /* Returned if Flash registers are not writable due to
                                       security */
   Fapi_Error_InvalidCPUID          /* Returned if OTP has an invalid CPUID */
}  ATTRIBUTE_PACKED Fapi_StatusType;
```

C.3.6 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.4 Structures

C.4.1 Fapi_FlashStatusWordType

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

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

C.4.2 Fapi_LibraryInfoType

This is the structure used to return API information

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

TEXAS INSTRUMENTS

Appendix D

SPNU632–December 2019

Parallel Signature Analysis (PSA) Algorithm

D.1 Function Details

The functions Fapi_calculatePsa() and Fapi_doPsaVerify() 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;
}
```

Appendix E

SPNU632 – December 2019

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 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;
}
```

E.1.2 For CM

```
//
//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;
}
```

Frequently Asked Questions (FAQ)

F.1 Flash ECC, ECC test mode and Linker ECC

1. Do we need to program ECC for the Flash?
   - Yes, it is a must to program ECC for the Flash and OTP. ECC-check is enabled at reset. Hence, if ECC is not programmed, ECC errors will occur. Double bit ECC errors will cause NMI.
2. Do we need to program ECC even if my application disables ECC-check?
   - BootROM code does not disable ECC-check. Hence, even if the user application disables ECC at some point in its code, ECC errors still occur when the bootROM jumps to the application code since ECC is disabled only at a later stage in the user application. This will cause a continuous reset cycle and hence application code never gets executed.
3. Do we need to disable ECC-check when programming Flash or OTP?
   - There is no need to disable ECC-check during Flash or OTP programming.
4. Is it a must to enable the ECC-check for Flash/OTP reads or fetches?
   - It depends on the safety requirements of your application. Enabling ECC-check generates an interrupt for single-bit errors (when the threshold is met) and NMI for uncorrectable errors as mentioned in the above FAQ. Hence, TI suggests enabling ECC-check. Note that ECC must be programmed irrespective of whether ECC-check is enabled or disabled.
5. What do you mean by enabling or disabling ECC-check? Is it same as using Fapi_AutoEccGeneration when programming Flash/OTP using Flash API?
   - No, they are different things. You use Fapi_AutoEccGeneration to generate and program ECC when using Flash API to program Flash/OTP. ECC-check is related to the read path of the Flash/OTP. When it is enabled, SECDED logic will check if there are any single or uncorrectable errors in the code/data that is fetched/read from Flash or OTP.
6. How do you enable or disable ECC-check?
   - You can use ECC_ENABLE register. Check TRM for its description.
7. Can ECC catch more than two bit errors?
   - It can catch only a max of two bit errors.
8. Can SECDED logic correct double bit errors?
   - No, it can’t. It can only correct single bit errors before data is given to CPU.
9. Does SECDED logic correct the single bit error in Flash/OTP as well?
   - No, it will give the corrected data to CPU but it will not correct the error in the Flash. User application has to erase and program the Flash if the error in the Flash has to be corrected.
10. Does FMC catch errors in the ECC space as well?
    - Yes, SECDED logic uses the 64-bit aligned address and the corresponding 64-bit data and 8-bit ECC value to evaluate the correctness of the data read from Flash/OTP.
11. Can we read ECC space to catch errors?
    - No, reading the ECC space does not cause SECDED logic to catch errors. Application should read the main array Flash/OTP to catch errors.
12. Do the debugger reads go through the SECDED logic and get corrected?
    - No. Debugger shows the Flash data (including errors) as is. Debugger reads do not trigger SECDED logic to evaluate ECC.
13. Are there any cases where ECC does not get evaluated for Flash/OTP reads when ECC check is

enabled?

- Yes, SECDED logic skips ECC evaluation when all the 64-bits of data and the corresponding 8-bit ECC value are either all 1s or all 0s. Also, ECC is not evaluated for DCSM link-pointer locations in OTP. Also, avoid using last 128-bits of the Flash bank when prefetch is enabled in the FMC – This is to avoid ECC errors.

14. Can we use ECC space for application code/data?
    - No. Fetches are not allowed to Flash ECC space. TI suggests to always program ECC and hence this space can't be used for application code or data.
15. Can we get ECC algorithm?
    - Please refer ECC Calculation Algorithm.
16. What is the suggested threshold value for single-bit Flash ECC errors?
    - It depends on the needs of the application. If the user application can't tolerate any errors, then of course, a threshold of zero has to be used. Also, Flash technology used in these devices is very reliable. Based on the field empirical data, no fails are reported for this concern. Hence, a threshold of 0 can be used.
17. How many SECDED modules are there in the FMC?
    - There are two SECDED modules. Flash read bus width is 128-bits (aligned). There is a dedicated SECDED module for evaluating ECC of the lower 64-bits and upper 64-bits separately.
18. When is the ECC evaluated for the data read/fetched from Flash?
    - ECC is evaluated when a read or fetch is done from Flash and before the data is placed in the flash prefetch buffer and cache.
19. Is ECC check available for the cache and prefetch buffer as well?
    - No.
20. What are the benefits of using ECC over CRC?
    - Both have their own advantages and disadvantages. CRC can be done on the entire Flash image at regular intervals – this takes CPU bandwidth. ECC is analyzed for every read or fetch of the Flash right when the Flash is accessed – this does not take CPU bandwidth. For that matter, even entire Flash bank can be read at regular intervals with ECC enabled to catch any errors if needed. However, ECC can catch a max of two-bit errors.
21. How is ECC calculated?
    - An 8-bit ECC value is calculated for every 64-bit data aligned on a 64-bit boundary in the Flash main array. Address of the corresponding 64-bit data is also included in the ECC calculation. Algorithm used for ECC calculation is provided at ECC Calculation Algorithm
22. How can I calculate ECC to program?
    - When using Flash API in the application to program Flash, you can pass "Fapi_AutoECCGeneration" as the programming mode parameter for the program function. When this mode is used, API calculates ECC for the user-provided address (aligned on a 64-bit memory boundary) and the corresponding 64-bit data. API programs the calculated ECC along with the main array data when using this mode. If not for a program operation, user application can also calculate ECC using the Fapi_calculateEcc() function provided in the Flash API library. Note that this function uses the SECDED hardware logic in the FMC to calculate the ECC. For previous devices, this function needs a left-shifted (by 1 bit position) address. There is no need to provide a left-shifted address to this function anymore. TMS320F2838x Flash API takes care of it before calculating the ECC.

    If the user wants to calculate ECC on a host device and not on the target MCU, then ECC calculation algorithm provided at ECC Calculation Algorithm can be used.

    When TI Flash tools are used, a setting for "AutoEccGeneration" option is provided, which when enabled (enabled by default unless user disables it) will use the Fapi_AutoEccGeneration option when programming the executable in the Flash/OTP using the API. 3rd party Tools like CodeSkin, Elprotronic, Data I/O and others also use the Fapi_AutoEccGeneration mode to generate and program the ECC for the Flash image.

    TI compiler tools offer option to generate ECC at the link step. This requires using special ECC specifier and directive in the linker command file. Using these, ECC for the Flash image can be appended in the executable image. When linker generated ECC is part of the executable image, user should make sure to disable the “AutoEccGeneration” option in the TI Flash tools since ECC is already part of the executable image. When linker generated ECC option is used, there is a provision to insert intentional errors at the selected addresses/offsets at selected bit positions. Please refer to “Error Correcting Code Testing (--ecc Options)” topic in “Linker Description” chapter of the “TMS320C28x Assembly Language Tools” user’s guide for C28x and “ARM Assembly Language Tools” user’s guide for CM.

23. When using ECC specifier and directive in the linker command file, is there a provision in CCS project settings to insert errors?
    - Yes, go to project build settings -> Build -> C2000 Linker -> Advanced Options -> Linker Output.

      You will notice below options:
      - ***--ecc:data_error option***
        - Inserts error at given address (address can be in main array or ECC space)
        - For inserting error in ECC space, you should know the ECC address
        - Enter the address (or symbol+offset) where you want to insert error, page, 16-bit bitmask to insert error
      - ***--ecc:ecc_error option***
        - Inserts error in ECC memory corresponding to given main array address
        - For inserting error in ECC space, you don’t need ECC address
        - Enter the main array address where you want to insert error at its corresponding ECC memory location, page, bitmask (width is 8bit) to insert error.
24. Can I program Main Array data and ECC memory separately?
    - You can program the Main Array data and ECC separately. However, each 64-bit dataword and the corresponding ECC word may only be programmed once per write/erase cycle. Hence, note that all the 64-bits of the data (aligned on the 64-bit boundary) must be programmed at the same time. And also, all the 8-bits of ECC must be programmed at the same time. As mentioned in the API reference guides, this is true even when data and ECC are programmed together - The Main Array flash programming must be aligned to 64-bit address boundaries and each 64-bit word may only be programmed once per write/erase cycle.

      In case of DCSM OTP, programming must be aligned to 128-bit address boundaries and each 128-bit word may only be programmed once. The exceptions are:
      - The DCSM Zx-LINKPOINTER1 and Zx-LINKPOINTER2 values in the DCSM OTP should be programmed together, and may be programmed 1 bit at a time as required by the DCSM operation.
      - The DCSM Zx-LINKPOINTER3 value in the DCSM OTP may be programmed 1 bit at a time, as required by the DCSM operation.
25. Why is that ECC should not be programmed for Link-pointer locations?
    - Link-pointer locations need to be programmed 1 bit at a time whenever the ZoneSelect Block gets updated. Once ECC is programmed for any 64-bit data location, that location cannot be programmed again (even to program a 1 to 0) since ECC value would change for the new data value and it is not possible to change the ECC value without an erase. Hence, ECC should not be programmed for Link-pointer locations in order to allow it for later updates.
26. Will Flash API fail if I use Fapi_AutoEccGeneration or Fapi_DataAndEcc or Fapi_EccOnly modes for programming the Link-pointer locations?
    - No, instead API will use Fapi_DataOnly mode to avoid programming ECC for the Link-pointer locations. Users have to exercise caution here – When user application or the Flash tools send the link-pointer address as the starting address for program operation, Flash API programs only the data but not the ECC. Hence, when the address provided to the API for program operation is link-pointer address, the length (number of words/bytes) of the data provided to the API should be such that the start address of program plus length should not go beyond the link-pointer locations. If not, Flash API will skip programming ECC for the non-link-pointer locations as well. This will cause ECC errors when application either reads or fetches from those locations for which ECC is not programmed. Hence, in applications, link-pointers should be maintained in a separate structure/section so that when the executable image is streamed by the Flash tools, only the link-

pointer related address gets the Fapi_DataOnly mode for programming by the API.

27. How can we notify CPU about the ECC errors?
    - Make sure to enable ECC-check by writing a value of 0xA in ECC_ENABLE register. This is the reset value for this register. This will enable SECDED logic to evaluate the data fetched/read from the Flash for the validity. Configure PIE to allow FLASH_CORRECTABLE_ERROR (INT12.11) interrupt from SECDED. Uncorrectable errors (double-bit errors or address-bit errors) will be notified via NMI in C2000 devices.
28. What all info is logged when a single bit error is caught by the SECDED logic?
    - Please check “Single-Bit Data Error” topic in ‘Flash and OTP Memory” chapter of the TRM.
29. What all info is logged when an uncorrectable error is caught by the SECDED logic?
    - Please check “Uncorrectable Error” topic in ‘Flash and OTP Memory” chapter of the TRM.
30. Why is my application stuck in an endless loop of the Single-bit error ISR when a single-bit error is caught?
    - Please make sure the error threshold is configured for a value >= 1. Threshold value of ‘0’ can cause this behavior in some devices. Please check errata advisory “Flash: A Single-Bit ECC Error May Cause Endless Calls to Single-Bit-Error ISR”.
31. How can I make sure that the SECDED logic blocks are working correctly at run time?
    - Application can use the ECC test mode explained in the “SECDED Logic Correctness Check” topic in the ‘Flash and OTP Memory” chapter of the TRM. In this mode, application can insert errors in the ECC test mode registers (provided for address, data and ECC) and see whether SECDED logic blocks are able to catch the errors or not. When this test mode is enabled, Flash gets bypassed and any reads to Flash address will be made to ECC test mode registers. Address, data and ECC configured in the ECC test mode registers are sent to SECDED logic for evaluation. By using this method, user application can check SECDED logic at the runtime without actually inserting errors in the Flash memory space. Please remember that this mode should be used by executing the code out of RAM since Flash gets bypassed when this mode is enabled.

    If application requires that the errors should be evaluated from direct Flash reads (and not from ECC test mode registers), then linker ecc options can be used to insert errors in the Flash image before programming in the Flash. Please refer to “Error Correcting Code Testing (--ecc Options)” topic in “Linker Description” chapter of the “TMS320C28x Assembly Language Tools” user’s guide for C28x and “ARM Assembly Language Tools” user’s guide for CM.

    For applications that use Flash API in their firmware upgrade solution, Flash API has a ECC calculation (Fapi_calculateEcc()) function to calculate ECC for a given address and data. User application can use this to calculate ECC. Errors can be inserted in the data and/or ECC as needed by the application. Modified data and ECC can be then programmed in the Flash using Fapi_DataAndEcc mode as the parameter for the Fapi_issueProgrammingCommand().
32. When using ECC test mode, does NMI occur for an uncorrectable error?
    - Yes, if NMI is enabled.
33. When using ECC test mode, does Single bit error interrupt occur for a single-bit error?
    - Yes, if the interrupt is enabled.
34. When ECC test mode is enabled, do we need to have the ECC enabled (ECC_ENABLE register) for SECDED logic to evaluate the ECC?
    - No, SECDED logic evaluates ECC even if the ECC is disabled.
35. When ECC test mode is used, why is the single bit error interrupt or the NMI occur continuously when the ECC_TEST_EN bit is enabled after inserting errors?
    - ECC test mode, when enabled, will continuously evaluate the test mode registers for errors in every cycle and hence, single-bit error interrupt and/or NMI will occur continuously until the ECC test mode is disabled. To avoid the continuous interrupts, disable the ECC test mode (ECC_TEST_EN = 0) in the flash single-bit error ISR and in the NMI ISR.
36. When using ECC test mode, why do we have both ECC_TEST_EN and DO_ECC_CALC bits?
    - ECC_TEST_EN bit enables ECC test mode by routing address, data and ECC from ECC test mode registers to the SECDED logic blocks. DO_ECC_CALC bit triggers SECDED logic to evaluate the address, data, ECC in FADDR_TEST, FDATAx_TEST and FECC_TEST registers for

ECC errors for a single cycle.

37. Are there any advantages of using linker generated ECC than that of AutoEccGeneration?
    - Aim of both methods is to generate ECC for the Flash/OTP image. AutoEccGeneration is simple in usage, since users do not have to make any changes to their linker cmd file to use this. However, there are few advantages when ECC is generated as part of the link step:
        - Some safety applications may require generating the ECC without using the SECDED hardware logic in the target MCU. The source of ECC generation and the source of ECC evaluation should be different for such applications. Linker generated ECC satisfies this requirement since ECC is generated by the compiler during link-step and not by the SECDED logic in the MCU.
        - Some customers would like to provide the entire application image to a third party for programming their devices. In such scenarios, customers want to include a checksum for the entire image including the ECC space. Linker generated ECC satisfies this since ECC values are also part of the executable image. Once, third parties program the image in the Flash, customers can run the checksum for the entire Flash image.
        - Apart from these, linker provides options to insert errors in the Flash/ECC space so that application can read those error-inserted addresses to check the health of the SECDED logic at run time.
38. Are there any disadvantages of using linker generated ECC than that of AutoEccGeneration?
    - Program time is more when linker generated ECC is used since Main Array and ECC space are programmed separately. When AutoEccGeneration is used, both MainArray and ECC space are programmed at the same time. Also, linker command file needs some changes in order to enable linker-generated ECC.
39. What are the changes that we need to make in the linker command file to generate ECC during the link-step?
    - Below steps should be followed in changing the linker command file to be able to generate and append ECC to the executable file as a separate section during link-step.
        - Define ECC memory range for every Flash memory range defined in the MEMORY segment of the linker command file. For example, see below sample linker command file contents. FLASH_A is the Flash Sector A memory range. Corresponding ECC memory range is FLASH_A_ECC. ECC memory range for all the Flash/OTP ranges should be defined in the same way.

```
Memory

{

PAGE 0: /* Program Memory */

FLASH_A  : origin=0x80000, length=0x2000, vfill = 0xFFFF

FLASH_B  : origin=0x82000, length=0x2000, vfill = 0xFFFF


PAGE 1: /* Data Memory */

FLASH_C  : origin=0x84000, length=0x2000, vfill = 0xFFFF

FLASH_D  : origin=0x86000, length=0x2000, vfill = 0xFFFF


FLASH_A_ECC  : origin=0x1080000, length=0x400, ECC = { /* ECC Specifier */

                                                      input_range = FLASH_A,
                                                      input_page  = 0,
                                                      algorithm   = C2000_Algo,
                                                      fill        = true
                                                     } /* End of ECC Specifier */

FLASH_B_ECC  : origin=0x1080400, length=0x400, ECC = {
```

```
                                                      input_range = FLASH_B,
                                                      input_page  = 0,
                                                      algorithm   = C2000_Algo,
                                                      fill        = true
                                                     }

FLASH_C _ECC : origin=0x1080800, length=0x400, ECC = {

                                                      input_range = FLASH_C,
                                                      input_page  = 1,
                                                      algorithm   = C2000_Algo,
                                                      fill        = true
                                                     }

FLASH_D_ECC  : origin=0x1080C00, length=0x400, ECC = {

                                                      input_range = FLASH_D,
                                                      input_page  = 1,
                                                      algorithm   = C2000_Algo,
                                                      fill        = true
                                                     }
} /* End of MEMORY */


SECTIONS

{


/* Allocate sections to memory areas */


} /* End of SECTIONS */


ECC {


C2000_Algo: parity_mask = 0xFC Mirroring = F021


} /* End of ECC Directive */
```

– Along with the memory attributes like origin and length, you might have noticed that there is an “ECC” specifier in the ECC memory range definition. Linker needs this to realize that it is an ECC memory range defined for a particular Flash Main array definition.
– The first parameter in the ECC specifier is “input_range” - this should point to the corresponding Flash Main array memory range for a given ECC memory range. For example, FLASH_A is the input_range for FLASH_A_ECC memory.
– The second parameter in the ECC specifier is “input_page” - this should point to the PAGE of the corresponding Flash Main array memory range for a given ECC memory range. For example, FLASH_A is defined in PAGE 0. Hence, the input_page for FLASH_A_ECC should be 0.
– The third parameter in the ECC specifier is “algorithm” - this is a name that you would choose (can be any name) and is defined later in the command file using the ECC directive (after the SECTIONS segment is defined). In above example, the name of the Algo is chosen as C2000_Algo.
– The fourth parameter in the ECC specifier is “fill” – if you choose “true”, linker would generate ECC data for the holes in the initialized data of the input range. The default is "true". The data that is filled in the holes is specified by using the “vfill” attribute in the input_range. We suggest you to use 0xFFFF for C28x (0xFF for ARM) since default value of the Flash in erased state is all 1s. Note that the ECC will be generated for the holes assuming the provided “vfill” value but

the “vfill” value (here 0xFFFF) will not make it in to the executable output file. This will help reduce the size of the executable. If you don’t want to generate ECC for the holes so that you can use that memory later at run time, you can choose “false” input for the “fill” parameter of the ECC specifier. Note that a “vfill” value (suggested as 0xFFFF) should be provided even in the case of a “fill = false” to fill any incomplete 64-bit data (aligned on a 64-bit memory boundary). If “vfill” value is not provided in this case, the compiler will assume 0s for the incomplete data in a 64-bit data. “vfill” value provided in case of “fill=false” case will not be used to fill any 64-bit holes that are completely unused. For more details on the VFILL usage, please refer to “Using the VFILL Specifier in the Memory Map” topic in the “Linker Description” chapter of the “TMS320C28x Assembly Language Tools” user’s guide for C28x and “ARM Assembly Language Tools” user’s guide for CM.

- A top level “ECC” directive should be added to the linker command file as shown at the end of the above given linker command file example to provide parameters for the algorithm that generates ECC data. The name of the algorithm can be any but should match the one that is specified in the ECC specifier. In the above example, name of the algorithm is chosen as C2000_Algo. Parameters provided for the algorithm should be as shown here in the example and they should not be modified by the user. They are the parameters for the ECC algorithm that the compiler uses to generate the ECC values.
- When ECC specifier and the ECC directive are used in the linker command file, compiler will automatically generate the ECC and append it as separate data sections in the executable output.
- Note that AutoEccGeneration in the Flash tools should be disabled when using the ECC specifier and the ECC directive in the linker command file.

40. How can we insert errors in the flash executable output using the linker ecc options?
    - Please refer to “Error Correcting Code Testing (--ecc Options)” topic in the “Linker Description” chapter of the “TMS320C28x Assembly Language Tools” user’s guide for C28x and “ARM Assembly Language Tools” user’s guide for CM.

F.2 Flash API

1. Can you give a brief overview of how to use the prominent Flash API functions?
   - Below steps provide a general overview.
     a. Configure PLL
     b. Copy the Flash initialization code from Flash to RAM
     c. Copy the Flash API from Flash to RAM
     d. Initialize Flash wait-states, fall back power mode, performance features and ECC
     e. Grab the ownership of the Flash pump using pump semaphore
     f. Initialize the Flash API by providing the Flash register-base address and operating frequency
        i. `Fapi_initializeAPI(F021_CPUx_BASE_ADDRESS,CLK_FREQUENCY);`
     g. Initialize the Flash bank and FMC for erase and program operations
        i. `Fapi_setActiveFlashBank(Fapi_FlashBankX);`
     h. Erase Flash sectors
        i. `Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, uint32 *pu32StartAddress);`
     i. Do blank check
        i. `Fapi_doBlankCheck(uint32 *Address, uint32 u32Length, Fapi_FlashStatusWordType *poFlashStatusWord);`
     j. Program the Flash using AutoEccGeneration mode (other modes can also be used as needed)
        i. `Fapi_issueProgrammingCommand(uint32 *Address, uint16 *Buffer, uint16 BufferLength, 0, 0, Fapi_AutoEccGeneration);`
     k. Verify that the Flash is programmed correctly
        i. `Fapi_doVerify(uint32 *Address, uint32 u32Length, uint32 *pu32CheckValueBuffer, Fapi_FlashStatusWordType *poFlashStatusWord);`

2. Why are the above Flash API functions different than that of some other C2000 devices like F2802x, F2803x, F2805x, F2806x, F2833x?
    - Flash wrapper (or the Flash Module Controller, FMC) is different in these devices. FMC in F28M35x, F28M36x, F2837xD, F2837xS, F2807x, F28004x, F2838x devices support ECC, 128-bit programming, faster erase and program etc. API for these devices takes benefit of these features.
3. What are the common debug tips that we can consider when Flash API fails to erase or program?
    - Please check below items:
        1. Make sure that the PLL is configured properly for the required system frequency
        2. Make sure that the Flash wait-states are configured correctly as per the datasheet
        3. Make sure that the Flash initialization routine is executed from RAM. Copy the Flash initialization routine from Flash to RAM.
        4. Make sure to not execute the Flash API and functions that call Flash API from the Flash bank on which the current erase/program operation is targeted. In Single bank devices, these should be executed from RAM. In dual bank devices, these functions should be executed from RAM or the bank on which the current erase/program operation is not targeted. When the application is designed to execute the Flash API (and the functions that call Flash API) from RAM, copy these functions from Flash to RAM before executing them.
        5. Make sure to grab Flash pump semaphore for the core.
        6. Make sure to configure FLSEM register (called as SECZONEREQUEST in Concerto) for the respective security zone (where applicable).
        7. Make sure to execute the Flash API from the same security zone as that of the Flash sector on which the current erase or program operation is targeted.
        8. Make sure that the voltage supply is able to meet the data manual specification during the Flash operations.
        9. Make sure to check the FMSTAT register after each of the erase and program operations to identify any failures.
        10. After erase operation is over, use Fapi_doBlankCheck() to check whether the sector is erased or not.
        11. After program operation is over, use Fapi_doVerify() to check whether the data is programmed correctly or not. Note that 32-bits is the minimum that can be checked using this function.
4. Why and when should Fapi_initializeAPI() be called?
    - After the device is first powered up, this function should be called once with appropriate parameters to initialize the Flash API internal state variables before any other Flash API operation is performed. This function should also be called if the System frequency or RWAIT is changed after this function is initially called.
5. Why and when should Fapi_setActiveFlashBank() be called?
    - This function initializes the FMC for further operations to be performed on the bank. This function is required to be called after the Fapi_initializeAPI() function and before any other Flash API operation is performed. For a given FMC, there is no need to call this function more than once, unless the System frequency or RWAIT are changed after this function is initially called.
6. Can we erase the entire bank with a single call of Fapi_issueAsyncCommandWithAddress()?
    - No, this function can erase only sector at a time. In order to erase multiple sectors, this function has to be called with each sector's address provided as the parameter.
7. Does Fapi_issueAsyncCommandWithAddress() support any command other than Fapi_EraseSector?
    - No. Use only erase sector command (Fapi_EraseSector) with this function. Bank erase is not supported in C2000 devices.
8. Does Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, xx) function call return after completing the sector erase operation?
    - No, this function issues an erase command to the Flash State Machine for the user-provided sector address. This function does not wait until the erase operation is over; it just issues the command and returns back. Hence, this function always returns success status when the

Fapi_EraseSector command is used. The user application must wait for the FMC to complete the erase operation before accessing the Flash bank on which the erase operation is performed. The Fapi_checkFsmForReady() function can be used to monitor the status of an issued command.

9. If the Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, xx) does not wait for the erase operation completion, how do we know whether the erase operation succeeded or not?
   - 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. See device specific Flash API reference guide provided in C2000Ware for FMSTAT details. Also, the user application should use the Fapi_doBlankCheck() function to verify that the Flash is erased.
10. How many bits can be programmed at a time using Fapi_issueProgrammingCommand()?
    - Due to architectural reasons, FSM (in FMC) can program only within a 128-bit aligned memory at a time and not across two 128-bit aligned memory ranges. Hence,
      1. if the address is 128-bit aligned, then a max of eight 16-bit words (128-bits) can be programmed at a time.
      2. if the address is “128-bit aligned address+1”, then a max of seven 16-bit words can be programmed at a time.
      3. if the address is “128-bit aligned address+2”, then a max of six 16-bit words can be programmed at a time.
      4. if the address is “128-bit aligned address+3”, then a max of five 16-bit words can be programmed at a time.
      5. if the address is “128-bit aligned address+4”, then a max of four 16-bit words can be programmed at a time.
      6. if the address is “128-bit aligned address+5”, then a max of three 16-bit words can be programmed at a time.
      7. if the address is “128-bit aligned address+6”, then a max of two 16-bit words can be programmed at a time.
      8. if the address is “128-bit aligned address+7”, then a max of one 16-bit words can be programmed at a time.

      Example: Consider an address that is 128-bit (8*16-bit) aligned; Say 0x80000 (0x80000 mod 8 = 0).

      1. If the address given for programming is 0x80000, a max of 8*16-bits can be programmed
      2. If the address given for programming is 0x80001, a max of 7*16-bits can be programmed
      3. If the address given for programming is 0x80002, a max of 6*16-bits can be programmed……..
      4. If the address given for programming is 0x80007, a max of 1*16-bits can be programmed

      Again, only addresses within a 128-bit aligned memory range can be programmed at a time. In order to program multiple 128-bit aligned memory ranges, this function should be called to program each range separately.

      Even though API allows to program less than 64-bits, it is very important to know that the main array flash programming must be aligned to 64-bit address boundaries and each 64-bit word may only be programmed once per write or erase cycle - This means that all the bits in a given 64-bit aligned memory of main array flash should be programmed at a time. The DCSM OTP programming must be aligned to 128-bit address boundaries and each 128-bit word may only be programmed once. The exceptions are:

      – The DCSM Zx-LINKPOINTER1 and Zx-LINKPOINTER2 values in the DCSM OTP should be programmed together, and may be programmed 1 bit at a time (using Fapi_DataOnly mode) as required by the DCSM operation.

      – The DCSM Zx-LINKPOINTER3 values in the DCSM OTP may be programmed 1 bit at a time (using Fapi_DataOnly mode), as required by the DCSM operation.
11. Does Fapi_issueProgrammingCommand() function call return after completing the program operation?
    - No, this function issues a program command to the Flash State Machine for the user-provided address and data. This function does not wait until the program operation is over; it just issues the command and returns back. Hence, the success status returned by this function should not be taken for a successful program operation. A success-status return from this function means that

the address, data and/or ECC supplied by the user application meet the requirements of the program operation and that the program command is issued successfully. The user application must wait for the FMC to complete the program operation before accessing the Flash bank on which the program command is issued. The Fapi_checkFsmForReady() function can be used to monitor the status of an issued command.

12. If the Fapi_issueProgrammingCommand() function does not wait for the program operation completion, how do we know whether the program operation succeeded or not?
    - 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. See device specific Flash API reference guide provided in C2000Ware for FMSTAT details. Also, the user application should use the Fapi_doVerify() function to verify that the Flash is programmed correctly.
13. Why do you have different programming modes for Flash programming function (Fapi_issueProgrammingCommand())?
    - Four modes are available:
        1. Fapi_DataOnly: Programs only the data without ECC
        2. Fapi_AutoEccGeneration: Programs 64-bit or 128-bit data along with the correct ECC generated by API
        3. Fapi_DataAndEcc: Programs user provided data and ECC (must program either 64-bits or 128-bits at a time along with their ECC)
        4. Fapi_EccOnly: Programs only ECC without data

      Usage scenarios of above modes are provided in a table under the description of this function in the device specific Flash API reference guides provided in C2000Ware.

      When using any of the above programming modes (except Fapi_EccOnly) for Flash main array, it is important to note that either 64-bits (aligned) or 128-bits (aligned) should be programmed at a time. When using Fapi_EccOnly mode, either 2 bytes (both LSB and MSB at a location in ECC memory) or 1 byte (LSB at a location in ECC memory) can be programmed.

      When using any of the above programming modes (except Fapi_EccOnly) for the DCSM OTP, programming must be aligned to 128-bit address boundaries and each 128-bit word may only be programmed once. The exceptions are:

      – The DCSM Zx-LINKPOINTER1 and Zx-LINKPOINTER2 values in the DCSM OTP should be programmed together, and may be programmed 1 bit at a time (using Fapi_DataOnly mode) as required by the DCSM operation.

      – The DCSM Zx-LINKPOINTER3 values in the DCSM OTP may be programmed 1 bit at a time (using Fapi_DataOnly mode), as required by the DCSM operation.
14. When using Fapi_AutoEccGeneration mode, what is the minimum number of 16-bit words that can be programmed?
    - Four. Note that ECC gets generated for the entire 64-bit aligned memory range assuming 1s for holes. For example, if only one 16-bit word is provided, all the other three 16-bit words will be assumed as all 1s to calculate the ECC. Once ECC gets programmed for a given 64-bit aligned memory, the unprogrammed bits (1s) in that 64-bit word cannot be programmed later without a sector erase since ECC is already programmed for the 64-bit word. Hence, always program 64-bits (aligned) or 128-bits (aligned). This is applicable for Fapi_DataAndEcc mode as well.
15. When using Fapi_AutoEccGeneration mode, for a given 128-bit aligned memory, can I program lower 64-bits at one time and upper 64-bits at another time?
    - Yes, ECC gets programmed only for the 64-bits that you supply.
16. Can we enable interrupts while executing Flash API?
    - Yes, Flash API is interruptible. However, there should not be any access (fetches or reads) to the Flash bank on which an erase or program operation is in progress. Hence, ISR should be executed from RAM or from a Flash bank (on devices with dual bank per core) on which the current erase/program is not targeted.
17. In order to erase or program a sector, can we execute Flash API from a different sector of the same flash bank?

- No, Flash API functions and the application functions that call Flash API should not be executed from the same bank. They should be executed from RAM or other Flash bank (if another bank exists for the same core).

18. Why is the data buffer size is in 16-bit words for C28x Fapi_issueProgrammingCommand() and is in 32-bit words for Fapi_doVerify()?
    - Since Flash read bus-width is 128-bits, a 32-bit verify will balance the performance and flexibility and hence provided the 32-bit verify option. When you program only 16-bits, you can append the data with the other 16-bits of Flash data for verify or verify once 32 bits are programmed.
19. When using Flash API in F2837xS dual bank devices, why do we need to switch the pump semaphore between the two banks when both banks are assigned for the same core?
    - Even though both banks are connected to the same core, each bank has its own FMC (unlike F28004x dual bank devices where there is only one FMC) and hence the pump semaphore should be used to connect the pump to the appropriate FMC for its erase/program operations. Also, for the same reason, Fapi_initializeAPI() and Fapi_setActiveFlashBank() functions should be called with appropriate parameters whenever the target Flash bank is changed for erase/program operations in F2837xS (this is not required in F28004x since there is only one FMC).
20. Why Fapi_getDeviceInfo() is not supported in F2838x devices?
    - Information like the number of banks, pins, memory size etc. is provided in the Data Manual for every PART NUMBER (PARTID). User application can infer these details without the need of this function.
21. Why Fapi_getBankSectors() is not supported in F2838 devices?
    - Information like start addresses of banks/sectors and size of sectors is provided in the Data Manual for every PART NUMBER (PARTID). User application can infer these details without the need of this function.
22. ECC obtained by using Fapi_CalculateEcc() seems to be incorrect – why?
    - For previous devices, this function needs a left-shifted (by 1 bit position) address. There is no need to provide a left-shifted address to this function anymore. TMS320F2838x Flash API takes care of it before calculating the ECC.
23. When using Fapi_DataAndEcc programming mode, I understand that I can program either 64-bits (aligned) or 128-bits (aligned). If the start address is 64-bit aligned but not 128-bit aligned, then only four 16-bit words can be programmed at the same time. In this case, how should I provide the ECC byte (since C28x is 16-bit addressing mode)?
    - The LSB of pu16EccBuffer (4th parameter of the Fapi_issueProgrammingCommand()) corresponds to the lower 64-bits and the MSB of pu16EccBuffer corresponds to the upper 64-bits of the 128-bit aligned memory. Hence, fill the upper byte with the ECC value and leave the lower byte as 0xFF in the 16-bit ECC word passed to the Fapi_issueProgrammingCommand(). Note that the u16EccBufferSizeInBytes (5th parameter of the Fapi_issueProgrammingCommand()) should be initialized as 1 since only one byte of ECC is supplied in this case.
24. When using Fapi_EccOnly programming mode, can I program the lower ECC byte (ECC corresponding to lower 64-bits of a 128-bit aligned memory) and the upper ECC byte (ECC corresponding to upper 64-bits of a 128-bit aligned memory) separately or should I program both of them together at the same time?
    - This mode 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). MSB alone cannot be programmed. If MSB has to be programmed, add existing LSB (in memory) to the ECC buffer and program both LSB and MSB together.
25. When using Fapi_EccOnly programming mode for Fapi_issueProgrammingCommand(), should we provide Flash main array address or the corresponding ECC address?
    - Flash main array address (corresponding to the ECC address) should be provided, even though ECC bytes are provided for program operation.
26. What is the use of Fapi_issueProgrammingCommandForEccAddresses() function ?
    - Use this function when you want to program ECC data alone by providing ECC space address. Note that Fapi_issueProgrammingCommand() takes main array address whereas Fapi_issueProgrammingCommandForEccAddresses() takes ECC space address. This is useful

when streaming an output file that has a separate ECC section (generated by Linker ECC options or anything method) which contains ECC address and ECC data. In such scenario, this function can be used to program the streamed ECC data using the ECC address directly – No need to convert the ECC address to main array address.

27. In F2838x devices, can we program CM Flash bank by executing the Flash API on C28x core?
    - No, CM Flash bank can be programmed only by executing the CM Flash API from CM RAM.
28. In F2838x devices, can we program C28x Flash bank by executing the Flash API on CM core?
    - No, C28x Flash bank can be programmed only by executing the C28x Flash API from C28x RAM.
29. In F2838x devices, can we program CPU1 Flash bank by executing the Flash API on CPU2 core?
    - No, CPU1 Flash bank can be programmed only by executing the Flash API from CPU1 RAM.
30. In F2838x devices, can we program CPU2 Flash bank by executing the Flash API on CPU1 core?
    - No, CPU2 Flash bank can be programmed only by executing the Flash API from CPU2 RAM.
31. How to copy the Flash API from Flash to RAM to execute it from RAM?
    - Please check the Flash API usage example provided in C2000Ware. In the example project, you will notice that the Flash API functions are executed from RAM. Observe below from the example:
        a. Linker command file used in the example, allocates Flash API library to .TI.ramfunc section, which has a Flash load address and a RAM run address.
        b. Example's main function calls memcpy() to copy the contents of the above section from Flash to RAM before they get executed.
32. Can we use Flash API to program OTP?
    - Yes, as mentioned in the Flash API reference guides, OTP can be programmed by using Flash API. There is no any difference in the API function usage for programming Flash vs OTP. However, note that OTP can not be erased. Make sure to look at the device's data-manual and TRM to know the OTP memory map and the fields available in OTP for users to program or use. Along with that, in F2837xD, F2837xS, F2807x and F28004x devices, note that the DCSM OTP programming must be aligned to 128-bit address boundaries and each 128-bit word may only be programmed once. The exceptions are:
        a. The DCSM Zx-LINKPOINTER1 and Zx-LINKPOINTER2 values in the DCSM OTP should be programmed together, and may be programmed 1 bit at a time as required by the DCSM operation.
        b. The DCSM Zx-LINKPOINTER3 value in the DCSM OTP may be programmed 1 bit at a time as required by the DCSM operation.

      Also, note that reserved fields in OTP should not be programmed.
33. Apart from using the Flash API library, is there any other way to access Flash state machine's registers to program or erase Flash at run-time?
    - No, Flash API library is the only way to erase/program the Flash by the application.

F.3 Flash Linker cmd file API

1. Why do you use align directive (ALIGN(x)) in the linker cmd files provided in the C2000Ware examples?
    - Fapi_AutoEccGeneration mode (one of the Flash API programming modes) programs the supplied data portion in Flash along with automatically generated ECC. The ECC is calculated for 64-bit aligned address and the corresponding 64-bit data. Any data not supplied (within a given 64-bit aligned memory) 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 for C28x (For ARM, ALIGN(8) should be used):

```
SECTIONS

{

.text       : > FLASH, ALIGN(4)

.cinit      : > FLASH, ALIGN(4)

.const      : > FLASH, ALIGN(4)

.init_array : > FLASH, ALIGN(4)

.switch     : > FLASH, ALIGN(4)


}
```

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

IMPORTANT NOTICE AND DISCLAIMER

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

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

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