# TMS320F2837xD Flash API

## Version 1.54

# Reference Guide



Literature Number: SPNU629
March 2017

# Contents

## List of Figures



## List of Tables

# 1 Introduction

**NOTE:** This document has undergone extensive changes from SPNU595 to SPNU629. We have separated these two documents for more clarity. SPNU595 is applicable only for TMS320F28M35x/36x devices; SPNU629 is applicable only for TMS320F2837xD devices.

This reference guide provides a detailed description of Texas Instruments' TMS320F2837xD Flash API functions that can be used to erase, program, and verify Flash on TMS320F2837xD devices.

## 1.1 Reference Material

Use this guide in conjunction with the device-specific data manual and technical reference manual that is being used.

## 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]* | Pointer to x |
| *parameter_2 [out]* | Handle for y |
| *parameter_n [in/out]* | Pointer to z |

Parameter passing is categorized as follows:

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

### Description

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

- Function blocks or might block 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 function **function_name()**.

**See Also**

Lists other functions or data types related to function **function_name()**.

**Example**

Provides an example (or a reference to an example) that illustrates the use of function **function_name()**.

# 2 TMS320F2837xD 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 API verifies that the appropriate RWAIT value is set for the specified system frequency.

**NOTE:** Please refer to the C2000 device-specific technical reference manual and data manual for more details regarding waitstates.

## 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 Functions**

| API Function | Description |
|---|---|
| Fapi_getFsmStatus() | Returns the FMSTAT status register value from the Flash memory controller (FMC) |
| Fapi_checkFsmForReady() | Returns whether or not the Flash state machine is ready or busy |
| Fapi_setActiveFlashBank() | Initializes the FMC and bank for an erase or program command |
| Fapi_issueFsmSuspendCommand() | Suspends FSM commands program data and erase sector |
| Fapi_flushPipeline() | Flushes the data cache in FMC |
| Fapi_isAddressEcc() | Determines if address falls in ECC ranges |
| Fapi_remapEccAddress() | Remaps an ECC address to the corresponding main address |

**Table 3. Summary of Asynchronous Operation Functions**

| API Function | Description |
|---|---|
| Fapi_issueAsyncCommandWithAddress() | Issues an erase sector command to FSM for the given sector address |
| Fapi_issueAsyncCommand() | Issues a command (Clear Status, Program Resume, Erase Resume, Clear_More) to FSM for operations that do not require an address |

**Table 4. Summary of Programming Functions**

| API Function | Description |
|---|---|
| Fapi_issueProgrammingCommand() | Sets up the required registers for programming and issues the command to the FSM |

**Table 5. Summary of Read Functions(1)**

| API Function | Description |
|---|---|
| Fapi_doVerify() | Verifies specified Flash memory range against supplied values |
| Fapi_doBlankCheck() | Verifies specified Flash memory range against erased state |
| Fapi_doPsaVerify() | Verifies a specified Flash memory range against the supplied PSA value |
| Fapi_calculatePsa() | Calculates a PSA value for the specified Flash memory range |

(1) These functions are not supported for F2837xD ECC memory space.

**Table 6. Summary of Information Functions**

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

(1) These functions are deprecated in subsequent devices. Therefore, TI suggests not to use these functions.

**Table 7. Summary of User-Defined Functions**

| API Function | Description |
|---|---|
| Fapi_serviceWatchdogTimer() | User-modifiable function to service watchdog timer |
| Fapi_setupEepromSectorEnable()(1) | Users should not modify nor edit the contents of this function. This function should be used as provided by TI. |
| Fapi_setupBankSectorEnable()(1) | User should not modify nor edit the contents of this function. This function should be used as provided by TI. |

(1) Users should not modify these functions, even though these functions are provided in the Fapi_User Defined Functions.c file. These functions are not merged into the library and are provided in the User-Defined section to maintain the same code across TI devices that share common code. These functions are merged into the library by TI in subsequent devices.

**Table 8. Summary of Utility Functions**

| API Function | Description |
|---|---|
| Fapi_calculateFletcherChecksum() | Function calculates a Fletcher checksum for the memory range specified |
| Fapi_calculateEcc() | Calculates the ECC for the supplied address and 64-bit word |

## 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 can be used except for the Fapi_getLibraryInfo() and Fapi_getDeviceInfo() functions. This 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 Fapi_initializeAPI(), this function must be called again before any other API function except Fapi_getLibraryInfo() and Fapi_getDeviceInfo() can be used. This will update the API internal state variables.

### 2.3.2 Building With the API

#### 2.3.2.1 Object Library Files

The Flash API object file is distributed in the standard TI COFF object format

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

#### 2.3.2.2 Distribution Files

The following API files are distributed with the installer:

- Library Files
  - F021_API_F2837xD_C28x.lib – This is the Flash API object file for both CPU1 and CPU2 applications in F2837xD devices.
  - F021_API_F2837xD_C28x_FPU32.lib – This is the Flash API object file for both CPU1 and CPU2 applications in F2837xD that are using the floating point unit.
- Source Files
  - Fapi_UserDefinedFunctions.c – This is file that contains the user definable functions. The appropriate include file for the user's device must be uncommented and the file must be compiled with the user's code. Note that users should not modify the Fapi_setupEeprom_SectorEnable() and Fapi_setupBankSectorEnable() functions.
- Include Files
  - This file sets up compile-specific defines and then includes the F021.h master include file.
    - F021_F2837xD_C28x.h – The master include file for F2837xD devices.
- The following include files should not be included directly by the user's code, but are listed here for user reference:
  - F021.h – This include file lists all API functions and includes all other include files.
  - Helpers.h – Set of helper defines.
  - Init.h – Defines the API initialization structure.
  - Registers_C28x.h – Little endian Flash memory controller registers structure.
  - Registers.h – Definitions common to all register implementations and includes the appropriate register include file for the selected device type.
  - Types.h – Contains all the enumerations and structures used by the API.
  - Constants/Constants.h – Constant definitions common to some C2000 devices.
  - Constants/F2837xD.h – Constant definitions for F2837xD devices.

### 2.3.3 Quick Facts About API Usage

Here are some important facts about API usage:

- Names of the Flash API functions start with a prefix "Fapi_".
- EALLOW and EDIS should be executed before and after calling Flash API functions to allow and disallow writes to protected registers, respectively.
- Pump semaphore should be gained by a CPU before performing Flash operations (erase, program, verify) on its bank.
- Flash API does not configure PLL. The user application should configure the PLL as needed and pass the configured CPUCLK value to Fapi_initializeAPI() function (details of this function are given later in this document).
- Flash API execution is interruptible. However, there should not be any read/fetch access from the Flash bank when an erase/program operation is in progress. Therefore, Flash API and the user application functions that call Flash API functions must be executed from RAM. Reason even the user application functions that call Flash API should be in RAM is that the return value (from Flash API function) assignment code also should be in RAM so that Flash is not accessed until the Flash operation is over and not just until when the Flash API function execution is over. For example, entire below code snippet should be executed from RAM and not just the Flash API functions. It is because the Fapi_issueAsyncCommandWithAddress() function issues the erase command to the FMC but it does not wait until the erase operation is over. Hence, as long as the FMC is busy with the current operation, there should not be a Flash access. This is just an example. This guideline is applicable for

all the Flash API functions.

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

- 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.
- It is permissable to program the data and ECC separately. However, each 64-bit dataword and 16-bit ECC word may only be programmed once per write/erase cycle.
- 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 as required by the DCSM operation.
  - The DCSM Zx-LINKPOINTER3 values in the DCSM OTP may be programmed 1 bit at a time on a 64-bit boundary to separate it from Zx-PSWDLOCK, which must only be programmed once.
- ECC should not be programmed for LINKPOINTER locations. Use Fapi_DataOnly mode for programming these locations.
- When using INTOSC as the clock source, a few SYSCLK frequency ranges need an extra waitstate to perform erase and program operations. After the operation is over, that extra waitstate is not needed. Please refer to the data manual for more details.
- Always configure waitstates as per the data manual before calling Flash API functions.
- Flash API does not configure (enable/disable) the watchdog. The user application can configure watchdog and service it as needed. In subsequent devices, the Fapi_ServiceWatchdogTimer() function is no longer supported. Therefore, TI suggests to not use this function - Instead the user applications can service the watchdog at regular interrupts (say by using a timer ISR) as needed.

# 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 |
| *u32HclkFrequency [in]* | System clock frequency in MHz |

#### Description

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

**NOTE:** The RWAIT register value must be set before calling this function.

#### Return Value

- **Fapi_Status_Success** (success)

#### Sample Implementation

```
#include “F021_F2837xD_C28x.h”

#define CPUCLK_FREQUENCY 200 /* 200 MHz System frequency */

int main(void)
{
   // Initialize System Control:
   InitSysCtrl();

   // Call Flash Initialization to setup flash waitstates
   // This function must reside in RAM
   InitFlash();

   // Gain pump semaphore
   SeizeFlashPump();

   //Jump to RAM and call the Flash API functions
   Example_CallFlashAPI();
}

#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
   Fapi_StatusType oReturnCheck;

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

```
    /* User code for further flash operations */
}
```

## 3.2 Flash State Machine Functions

### 3.2.1 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/program operation is successfully completed, in progress, suspended, or failed. The user application should check the value of this registrer to determine if there is any failure occurence after each erase and program operation.

#### Return Value

**Table 9. FMSTAT Register**

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

**Table 10. FMSTAT Register Field Descriptions**

| Bit | Field | Description |
|---|---|---|
| 31-15 | RSVD | Reserved |
| 14 | ILA | Illegal Address. When set, indicates that an illegal address is detected. The conditions below can set an illegal address flag:<br>• Writing to an address location in an unimplemented flash space.<br>• The address range does not match the type of FSM command. |
| 13 | RSVD | Reserved |
| 12 | PGV | Program verify. When set, indicates that a word is not successfully programmed 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 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 is 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. |

**Table 10. FMSTAT Register Field Descriptions (continued)**

| Bit | Field | Description |
|---|---|---|
| 2 | ESUSP | Erase Suspend. When set, this bit indicates that the flash module has received and processed an erase suspend operation. This bit remains set until the erase resume command has been issued or until the Clear_More command is run. |
| 1 | PSUSP | Program Suspend. When set, this bit indicates that the flash module has received and processed a program suspend operation. This bit remains set until the program resume command has been issued or until the Clear_More command is run. |
| 0 | RSVD | RSVD |

### 3.2.2 Fapi_checkFsmForReady()

Returns the status of the Flash State Machine

#### Synopsis

```
Fapi_StatusType Fapi_checkFsmForReady(void)
```

#### Parameters

None

#### Description

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

#### Return Value

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

### 3.2.3 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 the TMS320F2837xD device, only Fapi_FlashBank0 should be used for this parameter. This is true for both CPU1 and CPU2. |

#### Description

This function sets the Flash Memory Controller for further operations to be performed on the bank.

#### Return Value

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

#### Sample Implementation

```
#include “F021_F2837xD_C28x.h”

#define CPUCLK_FREQUENCY 200 /* 200 MHz System frequency */

int main(void)
{
   // Initialize System Control:
   InitSysCtrl();

   // Call Flash Initialization to setup flash waitstates
   // This function must reside in RAM
   InitFlash();

   // Gain pump semaphore
   SeizeFlashPump();

   //Jump to RAM and call the Flash API functions
   Example_CallFlashAPI();
}

#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
   Fapi_StatusType oReturnCheck;

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

   oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
   if(oReturnCheck != Fapi_Status_Success)
   {
      // Check Flash API documentation for possible errors
      Example_Error (oReturnCheck);
```

```
    }

    /* User code for further flash operations */
}
```

### 3.2.4 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, 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)

### 3.2.5 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

### 3.2.6 Fapi_remapEccAddress()

Takes ECC address and remaps it to the main address space

**Synopsis**

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

**Parameters**

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

**Description**

This function returns the main Flash address for the given ECC Flash address.

**Return Value**

- **32-bit Main Flash Address**

### 3.2.7 Fapi_isAddressEcc()

Indicates an address is in the Flash Memory Controller ECC space

**Synopsis**

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

**Parameters**

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

**Description**

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

**Return Value**

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

## 3.3 Asynchronous Functions

### 3.3.1 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_Erase sector. |
| *pu32StartAddress [in]* | Flash sector address for erase operation |

#### Description

This function issues an erase command to the Flash State Machine for the user-provided sector address. This function does not wait until the erase operation is over, it just issues the command and returns back. Hence, the user application must wait for the FMC to complete the erase operation before returning to any kind of Flash accesses.

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

#### Return Value

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

#### Sample Implementation

```
#include “F021_F2837xD_C28x.h”

#define CPUCLK_FREQUENCY 200 /* 200 MHz System frequency */

int main(void)
{
   // Initialize System Control:
   InitSysCtrl();

   // Call Flash Initialization to setup flash waitstates
   // This function must reside in RAM
   InitFlash();

   // Gain pump semaphore
   SeizeFlashPump();

   //Jump to RAM and call the Flash API functions
   Example_CallFlashAPI();
}

#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
   Fapi_StatusType oReturnCheck;
   Fapi_FlashStatusType oFlashStatus;
```

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

    oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
    if(oReturnCheck != Fapi_Status_Success)
    {
       // Check Flash API documentation for possible errors
       Example_Error (oReturnCheck);
    }

    // Erase a Sector
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, (uint32 *)0x0080000);
    // Wait until the erase operation is over
    while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}
    if(oReturnCheck != Fapi_Status_Success)
    {
       // Check Flash API documentation for possible errors
       Example_Error (oReturnCheck);
    }

    // Read FMSTAT register contents to know the status of FSM
    // after erase command to see if there are any erase operation
    // related errors
    oFlashStatus = Fapi_getFsmStatus();
    if (oFlashStatus!=0)
    {
       FMSTAT_Fail();
    }

    // Do blank check.
    // Verify that the sector is erased.
    oReturnCheck = Fapi_doBlankCheck((uint32 *)0x0080000, Bzero_16KSector_u32length,
                                     &oFlashStatusWord);

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

    /* User code for further flash operations */
}
```

### 3.3.2 Fapi_issueAsyncCommand()

Issues a command to the Flash State Machine

#### 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. 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.

Below are the details of these commands:

- Clear Status (Fapi_ClearStatus): Executing this command clears the ILA, PGV, EV, CSTAT, VOLTSTAT, and INVDAT bits in the FMSTAT register. Flash API issues this command before issuing a program or an erase command.
- Clear More (Fapi_ClearMore): Executing this command clears everything the Clear Status command clears and additionally, clears the ESUSP and PSUSP bits in the FMSTAT register.
- Program Resume (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.
- Erase Resume (Fapi_Erase Resume): 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)

#### Sample Implementation

```
#include “F021_F2837xD_C28x.h”

#define CPUCLK_FREQUENCY 200 /* 200 MHz System frequency */

int main(void)
{
   // Initialize System Control:
   InitSysCtrl();

   // Call Flash Initialization to setup flash waitstates
   // This function must reside in RAM
   InitFlash();

   // Gain pump semaphore
   SeizeFlashPump();

   //Jump to RAM and call the Flash API functions
```

```
    Example_CallFlashAPI();
}

#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
    Fapi_StatusType oReturnCheck;
    Fapi_FlashStatusType oFlashStatus;

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

    oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
    if(oReturnCheck != Fapi_Status_Success)
    {
        // Check Flash API documentation for possible errors
        Example_Error (oReturnCheck);
    }

    // Issue an async command
    oReturnCheck = Fapi_issueAsyncCommand(Fapi_ClearMore);
    // Wait until the operation is over
    while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}
    if(oReturnCheck != Fapi_Status_Success)
    {
        // Check Flash API documentation for possible errors
        Example_Error (oReturnCheck);
    }

    // Read FMSTAT register contents to know the status of FSM
    // after the command to see if there are any operation specific errors
    oFlashStatus = Fapi_getFsmStatus();
    if (oFlashStatus!=0)
    {
        FMSTAT_Fail();
    }

    /* User code for further flash operations */
}
```

## 3.4 Program Functions

### 3.4.1 Fapi_issueProgrammingCommand()

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

**Synopsis**

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

**Parameters**

| | | |
|---|---|---|
| *pu32StartAddress [in]* | start address in Flash for the data and ECC to be programmed | |
| *pu16DataBuffer [in]* | pointer to the Data buffer address | |
| *u16DataBufferSizeInWords [in]* | number of 16-bit words in the Data buffer | |
| *pu16EccBuffer [in]* | pointer to the ECC buffer address | |
| *u16EccBufferSizeInBytes [in]* | number of 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. The pu16EccBuffer word corresponds to the main array aligned on a 128-bit address boundary. The LSB of pu16EccBuffer corresponds to the lower 64-bits of the main array and the MSB of pu16EccBuffer corresponds to the upper 64-bits of the main array.

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.

**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.

*Fapi_AutoEccGeneration* – This 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, and data not supplied is treated as 0xFF. The data restrictions for Fapi_DataOnly also exist for this option.

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

Here is an example:

```
SECTIONS
{
.text   : > FLASH, PAGE = 0, ALIGN(4)
.cinit  : > FLASH, PAGE = 0, ALIGN(4)
.const  : > FLASH, PAGE = 0, ALIGN(4)
.econst : > FLASH, PAGE = 0, ALIGN(4)
.pinit  : > FLASH, PAGE = 0, ALIGN(4)
.switch : > FLASH, PAGE = 0, ALIGN(4)
}
```

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

*Fapi_DataAndEcc* – This 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. (For example, data buffer length is 4 words, the ECC buffer must be 1 byte).

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.

*Fapi_EccOnly* – This mode will only program the ECC portion in Flash at the address specified. It can program either 1 byte or 2 bytes (2 is max).

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 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.

### 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 loop to program 128-bits (or 64-bits as needed by application) at a time.
- The Main Array flash programming must be aligned to 64-bit address boundaries and each 64-bit word may only be programmed once per write/erase cycle.
- It is alright to program the data and ECC separately. However, each 64-bit dataword and 16-bit ECC word may only be programmed once per write/erase cycle.
- 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 as required by the DCSM operation.
- The DCSM Zx-LINKPOINTER3 values in the DCSM OTP may be programmed 1 bit at a time on a 64-bit boundary to separate it from Zx-PSWDLOCK, which must only be programmed once.

### Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (failure: Data buffer size specifiedis incorrect)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: ECC buffer size specified is incorrect)
- **Fapi_Error_AsyncDataEccBufferLengthMismatch** (failure: Data buffer size either is not 64-bit aligned or data length crosses the 128-bit aligned memory boundary)

### Sample Implementation

```
#include “F021_F2837xD_C28x.h”

#define CPUCLK_FREQUENCY 200 /* 200 MHz System frequency */
int main(void)
{
   // Initialize System Control:
   InitSysCtrl();

   // Call Flash Initialization to setup flash waitstates
   // This function must reside in RAM
   InitFlash();

   // Gain pump semaphore
   SeizeFlashPump();

   //Jump to RAM and call the Flash API functions
   Example_CallFlashAPI();

}

#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
   Fapi_StatusType oReturnCheck;
   Fapi_FlashStatusType oFlashStatus;
   uint16 au16DataBuffer[8] = {0x0001, 0x0203, 0x0405, 0x0607, 0x0809, 0x0A0B, 0x0C0D, 0x0E0F};
   uint32 *DataBuffer32 = (uint32 *)au16DataBuffer;
   uint32 u32Index = 0;

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

   oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
   if(oReturnCheck != Fapi_Status_Success)
   {
      // Check Flash API documentation for possible errors
      Example_Error (oReturnCheck);
   }

   for(u32Index = 0x80000; (u32Index < 0x80200) &&
                        (oReturnCheck == Fapi_Status_Success); u32Index+=8)
   {
      // Issue program command
      oReturnCheck = Fapi_issueProgrammingCommand((uint32 *)u32Index, au16DataBuffer, 8,
                                                  0, 0, Fapi_AutoEccGeneration);

      // Wait until the Flash program operation is over
```

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

        // Read FMSTAT register contents to know the status of FSM after
        // program command to see if there are any program operation related errors
        oFlashStatus = Fapi_getFsmStatus();
        if(oFlashStatus != 0)
        {
           //Check FMSTAT and debug accordingly
           FMSTAT_Fail();
        }

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

    /* User code for flash operations */
}
```

## 3.5 Read Functions

### 3.5.1 Fapi_doBlankCheck()

Verifies region specified is erased value

**Synopsis**

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

**Parameters**

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

**Description**

This function checks the device for blank (erase state) starting at the specified address for the length of 32-bit words specified. If a non-blank location is found, these results will be returned in the poFlashStatusWord parameter.

**Restrictions**

This function is not supported for F2837xD ECC memory space.

**Return Value**

- **Fapi_Status_Success** (success)
- **Fapi_Error_Fail** (failure: region specified is not blank)

### 3.5.2 Fapi_doVerify()

Verifies region specified against supplied data

#### Synopsis

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

#### Parameters

| | | |
|---|---|---|
| *pu32StartAddress [in]* | | start address for region to verify |
| *u32Length [in]* | | length of region in 32-bit words to verify |
| *pu32CheckValueBuffer [in]* | | address of buffer to verify region against |
| *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. This will use normal read, read margin 0 and read margin 1 modes for verifying the data.

#### Restrictions

This function is not supported for F2837xD ECC memory space.

#### Return Value

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

### 3.5.3 Fapi_doPsaVerify()

Verifies region specified against specified PSA value

#### Synopsis

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

#### Parameters

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

#### Description

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

#### Restrictions

This function is not supported for F2837xD ECC memory space.

#### Return Value

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

### 3.5.4 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.

#### Restrictions

This function is not supported for F2837xD ECC memory space.

#### Return Value

- **PSA value** (success)

## 3.6 Informational Functions

### 3.6.1 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
- u8ApiMinorVersion – Minor version number of this compile of the API. Minor version is 54 for F2837xD 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. Tech type used in this device 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)

### 3.6.2 Fapi_getDeviceInfo()

Returns information about specific to device code is being executed on

#### Synopsis

```
Fapi_DeviceInfoType Fapi_getDeviceInfo(void)
```

#### Parameters

*None*

#### Description

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

- u16NumberOfBanks – Number of banks on the device
- u16DevicePackage – Device package pin count
- u16DeviceMemorySize – Device memory size
- u32AsicId – Device ASIC id
- u32LotNumber – Device lot number
- u16FlowCheck – Device Flow check
- u16WaferNumber – Device wafer number
- u16WaferXCoordinate – Device wafer X coordinate
- u16WaferYCoordinate – Device wafer Y coordinate

#### Restrictions

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

#### Return Value

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

### 3.6.3 Fapi_getBankSectors()

Returns the sector information for the requested bank

#### Synopsis

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

#### Parameters

| | |
|---|---|
| *oBank [in]* | Bank to get information on |
| *poFlashBankSectors [out]* | Returned structure with the bank information |

#### Description

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

- oFlashBankTech – Indicates if bank is an FLEP, FLEE or FLES bank type
- u32NumberOfSectors – Indicates the number of sectors in the bank.
- u32BankStartAddress – Starting address of the bank.
- au8SectorSizes[] – An array of sectors sizes for each sector in the bank.

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

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

### Restrictions

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

### Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_FeatureNotAvailable** (failure: Not all devices have this support in the Flash Memory Controller)
- **Fapi_Error_InvalidBank** (failure: Bank does not exist on this device)

## 3.7 Utility Functions

### 3.7.1 Fapi_calculateFletcherChecksum()

Calculates the Fletcher checksum from the given address and length

**Synopsis**

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

**Parameters**

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

**Description**

This function generates a 32-bit Fletcher checksum starting at the supplied address for the number of 16-bit words specified.

**Restrictions**

This function is not supported for F2837xD ECC memory space.

**Return Value**

- 32-bit Fletcher Checksum value

### 3.7.2 Fapi_calculateEcc()

Calculates the ECC for a 64-bit value

**Synopsis**

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

**Parameters**

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

**Description**

This function will calculate the ECC for a 64-bit aligned word including address. Note that the user application should left-shift the address by 1 position before passing to this function.

**Return Value**

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

## 3.8 User Definable Functions

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

### 3.8.1 Fapi_serviceWatchdogTimer()

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

#### Synopsis

```
Fapi_StatusType Fapi_serviceWatchdogTimer(void)
```

#### Parameters

*None*

#### Description

This function allows the user to service their Watchdog timer in the Read Functions, Table 5. This function is called in the Read functions when the address being read crosses the 256-word (16-bit word) aligned address boundaries.

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

#### Restrictions

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

#### Return Value

- **Fapi_Status_Success** (success)

#### Sample Implementation

```
#include “F021_F2837xD_C28x.h”
Fapi_StatusType Fapi_serviceWatchdogTimer(void)
{
   /* User to add their own watchdog servicing code here */

   return(Fapi_Status_Success);
}
```

### 3.8.2 Fapi_setupEepromSectorEnable()

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

#### Synopsis

```
Fapi_StatusType Fapi_setupEepromSectorEnable(void)
```

#### Parameters

*None*

#### Description

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

#### Restrictions

This function is deprecated and not supported in subsequent devices (but users should not remove or edit this function in TMS320F2837xD devices).

#### Return Value

- **Fapi_Status_Success** (success)

### 3.8.3 Fapi_setupBankSectorEnable()

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

#### Synopsis

```
Fapi_StatusType Fapi_setupBankSectorEnable(void)
```

#### Parameters

*None*

#### Description

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

#### Restrictions

Note that users should not edit the contents of this function even though it is provided in the Fapi_UserDefinedFunctions.C file. This function should be used as provided by TI. The reason TI provides this function outside of the API Library is to keep source compatibility across TI devices where applicable. This function is deprecated and not supported in subsequent devices, but users should not remove or edit this function in TMS320F2837xD devices.

#### Return Value

- **Fapi_Status_Success** (success)

# 4 Recommended FSM Flows

## 4.1 New devices from Factory

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

## 4.2 Recommended Erase Flow

The following diagram describes the high-level flow for erasing a sector(s). Please refer to Figure 1 for further information.

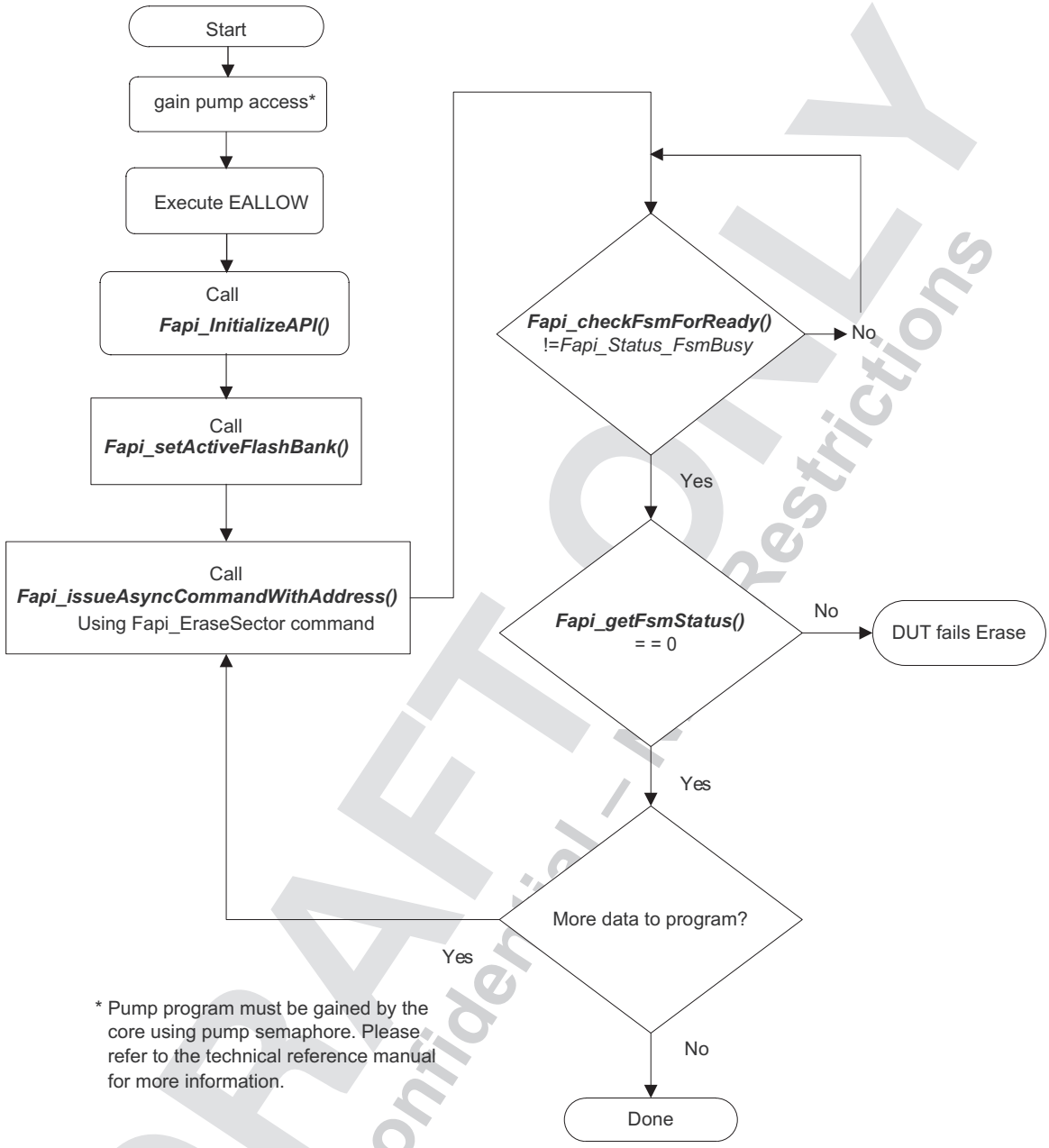


**Figure 1. Recommended Erase Flow**

### 4.3 Recommended Program Flow

The following diagram describes the high-level flow for programming a device. This flow assumes the user has already erased all affected sectors or banks following the Recommended Erase Flow (see Figure 2). See Section 3.4 for further information.

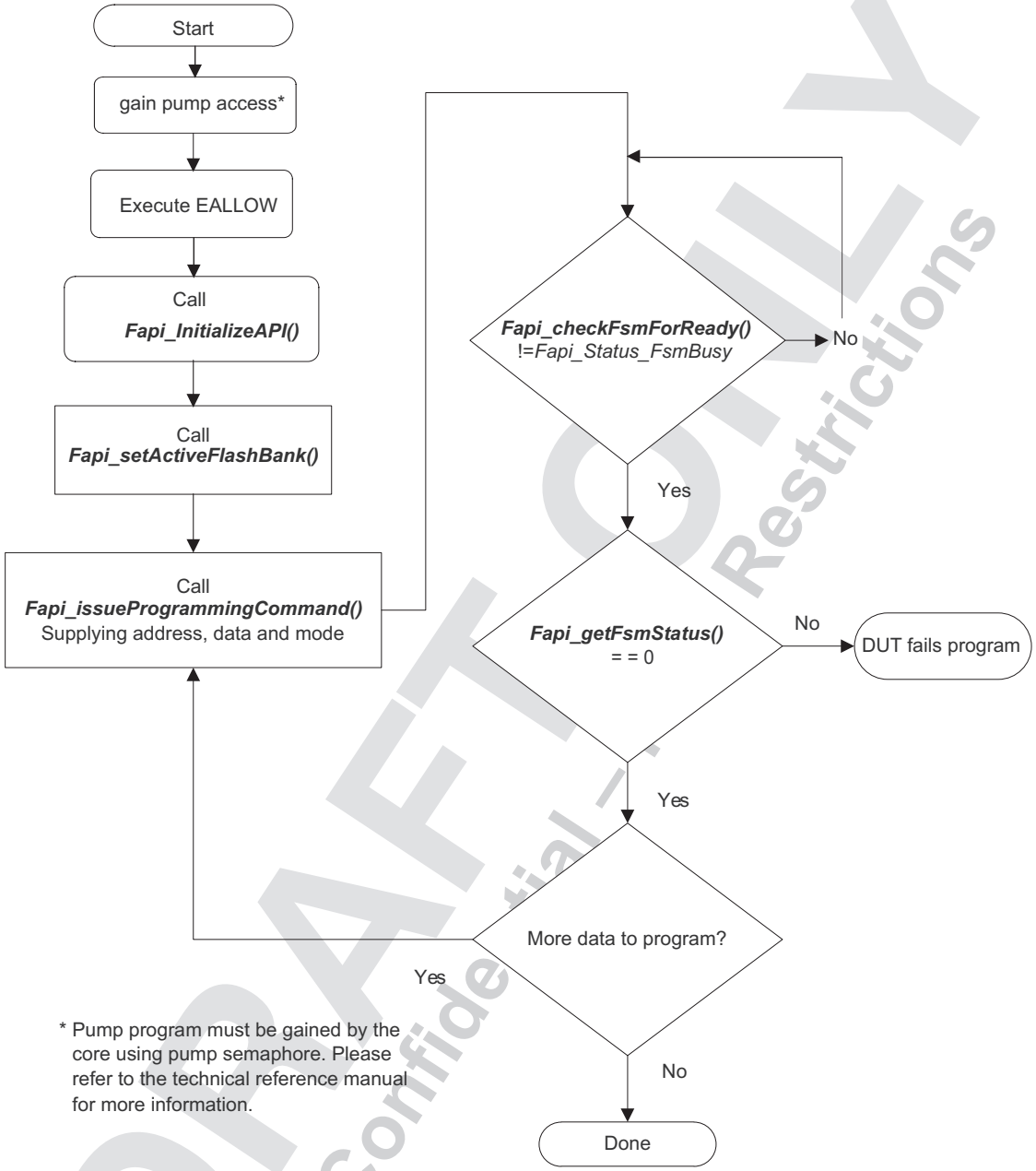


**Figure 2. Recommended Program Flow**

# Flash State Machine Commands

## A.1 Flash State Machine Commands

**Table 11. Flash State Machine Commands**

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

# Object Library Function Information

## B.1 C28x Library

**Table 12. C28x Function Sizes and Stack Usage**

| Function Name | Size In Words | Worst Case Stack Usage |
|---|---|---|
| Fapi_calculateEcc | TBD | TBD |
| Fapi_calculateFletcherChecksum | TBD | TBD |
| Fapi_calculatePsa<br>*Includes references to the following functions*<br>• Fapi_isAddressEcc<br>• Fapi_serviceWatchdogTimer | TBD | TBD |
| Fapi_checkFsmForReady | TBD | TBD |
| Fapi_doBlankCheck<br>*Includes references to the following functions*<br>• Fapi_flushPipeline<br>• Fapi_serviceWatchdogTimer<br>• Fapi_waitDelay<br>• Fapi_isAddressEcc | TBD | TBD |
| Fapi_doVerify<br>*Includes references to the following functions*<br>• Fapi_flushPipeline<br>• Fapi_serviceWatchdogTimer<br>• Fapi_waitDelay<br>• Fapi_isAddressEcc | TBD | TBD |
| Fapi_flushPipeline<br>*Includes references to the following functions*<br>• Fapi_waitDelay | TBD | TBD |
| Fapi_getBankSectors | TBD | TBD |
| Fapi_getDeviceInfo | TBD | TBD |
| Fapi_getFsmStatus | TBD | TBD |
| Fapi_getLibraryInfo | TBD | TBD |
| Fapi_initializeAPI | TBD | TBD |
| Fapi_isAddressEcc | TBD | TBD |
| Fapi_issueAsyncCommand | TBD | TBD |
| Fapi_issueAsyncCommandWithAddress<br>*Includes references to the following functions*<br>• Fapi_setupBankSectorEnable<br>• Fapi_setupEepromSectorEnable | TBD | TBD |
| Fapi_issueFsmSuspendCommand | TBD | TBD |

**Table 12. C28x Function Sizes and Stack Usage (continued)**

| | | |
|---|---|---|
| Fapi_issueProgrammingCommand<br>*Includes references to the following functions*<br>• Fapi_calculateEcc<br>• Fapi_setupBankSectorEnable<br>• Fapi_setupEepromSectorEnable | TBD | TBD |
| Fapi_remapEccAddress | TBD | TBD |
| Fapi_setActiveFlashBank<br>*Includes references to the following functions*<br>• Fapi_calculateFletcherChecksum | TBD | TBD |
| Fapi_serviceWatchdogTimer[(1)] | TBD | TBD |
| Fapi_setupBankSectorEnable | TBD | TBD |
| Fapi_setupEepromSectorEnable | TBD | TBD |

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

# Typedefs, defines, enumerations and structures

## C.1 Type Definitions

```
#if defined(__TMS320C28XX__)

typedef unsigned char       boolean;

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

typedef unsigned int        uint16_least;
typedef unsigned long int   uint32_least;

typedef signed int          sint16_least;
typedef signed long int     sint32_least;

typedef float                    float32;
typedef long double              float64;

#else

typedef unsigned char       boolean;

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

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

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

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

typedef float               float32;
typedef double              float64;

#endif
```

## C.2 Defines

```
#if FALSE != 0
#define false FALSE
#endif
#if TRUE != 1
#define true TRUE
```

```
#endif
```

## C.3 Enumerations

### C.3.1 Fapi_CpuType

This is used to indicate which type of CPU is being used.

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

### C.3.2 Fapi_AddressMemoryType

This is used to indicate which type of Address is being used.

```
typedef enum
{
   Fapi_Flash,
   Fapi_FlashEcc,
   Fapi_Otp,
   Fapi_OtpEcc,
   Fapi_Undefined
}  ATTRIBUTE_PACKED Fapi_AddressMemoryType;
```

### C.3.3 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.3.4 Fapi_FlashBankType

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

```
typedef enum
{
   Fapi_FlashBank0,
   Fapi_FlashBank1, /* Not used for TMS320F2837xD devices */
   Fapi_FlashBank2, /* Not used for TMS320F2837xD devices */
   Fapi_FlashBank3, /* Not used for TMS320F2837xD devices */
   Fapi_FlashBank4, /* Not used for TMS320F2837xD devices */
   Fapi_FlashBank5, /* Not used for TMS320F2837xD devices */
   Fapi_FlashBank6, /* Not used for TMS320F2837xD devices */
   Fapi_FlashBank7  /* Not used for TMS320F2837xD devices */
}  ATTRIBUTE_PACKED Fapi_FlashBankType;
```

### C.3.5 Fapi_FlashBankTechType

This is used to indicate what F021 Bank Technology the bank is

```
typedef enum
```

```
{
    Fapi_FLEP,
    Fapi_FLEE,
    Fapi_FLES,
    Fapi_FLHV,
    Fapi_TechTBD
}  ATTRIBUTE_PACKED Fapi_FlashBankTechType;
```

### C.3.6 Fapi_FlashStateCommandsType

This contains all the possible Flash State Machine commands.

```
typedef enum
{
   Fapi_ProgramData    = 0x0002,
   Fapi_EraseSector    = 0x0006,
   Fapi_EraseBank      = 0x0008,  /* Not available for TMS320F2837xD devices */
   Fapi_ValidateSector = 0x000E,  /* Not available for TMS320F2837xD devices */
   Fapi_ClearStatus    = 0x0010,
   Fapi_ProgramResume  = 0x0014,
   Fapi_EraseResume    = 0x0016,
   Fapi_ClearMore      = 0x0018
}  ATTRIBUTE_PACKED Fapi_FlashStateCommandsType;
```

### C.3.7 Fapi_FlashReadMarginModeType

This contains all the possible Flash State Machine commands.

```
typedef enum
{
   Fapi_NormalRead = 0x0,
   Fapi_RM0        = 0x1, /* Technology used in TMS320F2837xD devices does not need this */
   Fapi_RM1        = 0x2  /* Technology used in TMS320F2837xD devices does not need this */
}  ATTRIBUTE_PACKED Fapi_FlashReadMarginModeType;
```

### C.3.8 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 HClk and RWAIT/EWAIT */
   Fapi_Error_InvalidCpu,           /* Returned if the specified Cpu does not exist */
   Fapi_Error_InvalidBank,          /* Returned if the specified bank does not exist */
   Fapi_Error_InvalidAddress,       /* Returned if the specified Address does not exist in Flash
or OTP */
   Fapi_Error_InvalidReadMode,      /* Returned if the specified read mode does not exist */
   Fapi_Error_AsyncIncorrectDataBufferLength,
   Fapi_Error_AsyncIncorrectEccBufferLength,
   Fapi_Error_AsyncDataEccBufferLengthMismatch,
   Fapi_Error_FeatureNotAvailable  /* FMC feature is not available on this device */
}  ATTRIBUTE_PACKED Fapi_StatusType;
```

### C.3.9 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_EngineeringRowType

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

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

### C.4.2 Fapi_FlashStatusWordType

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

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

### C.4.3 Fapi_LibraryInfoType

This is the structure used to return API information

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

### C.4.4 Fapi_DeviceInfoType

This is the structure used to return device information

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

### C.4.5 Fapi_FlashBankSectorsType

This gives the structure of a bank and technology type

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

# Parallel Signature Analysis (PSA) Algorithm

## D.1 Function Details

The functions Section 3.5.3 and Section 3.5.4 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 primative polynomial:

```
f(X) = 1 + X + X2 + X22 + X31

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;
}
```

## IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

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

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

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

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

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

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

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

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

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