TMS320F28004x Flash API

Version 1.56.01.00

Reference Guide

Literature Number: SPNU628A
February 2017–Revised December 2017

Contents

List of Figures

List of Tables

1 Introduction

This reference guide provides a detailed description of Texas Instruments' TMS320F28004x Flash API Library (F021_API_F28004x_FPU32.lib or F021_ROM_API_F28004x_FPU32.lib) functions that can be used to erase, program and verify Flash on TMS320F28004x devices. Note that Flash API V1.56.xx.xx should be used only with TMS320F28004x devices. The Flash API Library is provided in C2000Ware at C2000Ware_x_xx_xx_xx\libraries\flashapi\f28004x.

1.1 Reference Material

Use this guide in conjunction withUse this guide in conjunction with *TMS320F28004x Delfino™ Microcontrollers Data Manual* and *TMS320F28004x™Piccolo Microcontrollers Technical Reference Manual*.

1.2 Function Listing Format

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

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

Synopsis

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

```
<return_type> function_name(
                <type_1> parameter_1,
                <type_2> parameter_2,

                <type_n> parameter_n
                         )
```

Parameters

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

Parameter passing is categorized as follows:

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

Description

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

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

Restrictions

Specifies any restrictions in using this function.

Return Value

Specifies any value or values returned by the function.

See Also

Lists other functions or data types related to the function.

Sample Implementation

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

2 TMS320F28004x 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 read the data manual for Flash and OTP 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 *TMS320F28004x Piccolo Microcontrollers 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 Flash Memory Controller (FMC) and banks for an erase or program command
Fapi_issueAsyncCommandWithAddress()	Issues an erase sector command to FSM for the given address
Fapi_issueProgrammingCommand()	Sets up the required registers for programming and issues the command to the FSM
Fapi_issueProgrammingCommandForEccAddress()	Remaps an ECC address to the main data space and then call Fapi_issueProgrammingCommand() to program ECC
Fapi_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_doVerify()	Verifies specified Flash memory range against supplied values
Fapi_calculatePsa()	Calculates a PSA value for the specified Flash memory range
Fapi_doPsaVerify()	Verifies a specified Flash memory range against the supplied PSA value

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

Table 5. Summary of Utility Functions (continued)

API Function	Description
Fapi_isAddressEcc()	Determines if the address falls in ECC ranges
Fapi_remapEccAddress()	Remaps an ECC address to corresponding main address
Fapi_calculateFletcherChecksum()	Function calculates a Fletcher checksum for the memory range specified

Note that Fapi_getDeviceInfo() and Fapi_getBankSectors() are removed in TMS320F28004x 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 Section 2.3.3 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 FMC and Bank Setup

Before performing a Flash operation for the first time, the *Fapi_setActiveFlashBank()* function must be called. There is only one FMC on F28004x devices despite there being two flash banks. Therefore, calling this function once, and using Fapi_FlashBank0 as the passed parameter is sufficient. There is no need to recall this function when switching flash banks.

2.3.1.3 On System Frequency Change

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

2.3.2 Building With the API

2.3.2.1 Object Library Files

The Flash API object file is distributed in the standard Common Object File format (COFF).

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

2.3.2.2 Distribution Files

The following API files are distributed in the C2000Ware\libraries\flash_api\f28004x\ folder:

- Library Files
  - TMS320F28004x Flash API is embedded into the Boot ROM of this device. This differs from other C28x devices where the API is wholly software. As such, both a software library (F021_API_F28004x_FPU32.lib) and a BootROM API symbols library (F021_ROM_API_F28004x_FPU32.lib) are provided. In order for the application to be able to erase or program the Flash/OTP, one of these two library files should be included in the application with

the symbol library given preference.

- F021_API_F28004x_FPU32.lib – This is the Flash API object file (software API library) for TMS320F28004x devices.
- F021_ROM_API_F28004x_FPU32.lib – This is the Boot ROM Flash API symbols library for TMS320F28004x devices. This contains the addresses of the various Flash API functions that are embedded into the device Boot ROM. Since all of the functions reside in ROM, adding the boot ROM symbols to the application takes up only a small amount of Flash and/or RAM space when compared to that of the Software API library.

Fixed point version of the API library is not provided.

- Include Files
  - F021_F28004x_C28x.h – The master include file for TMS320F28004x devices. This file sets up compile-specific defines and then includes the F021.h master include file
- 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_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/F28004x.h – Constant definitions for F28004x devices.

Note that Helpers.h is not provided since it is not needed for user applications.

2.3.3 Key Facts For Flash API Usage

Here are some important facts about API usage:

- Names of the Flash API functions start with a prefix “Fapi_”.
- Flash API does not configure PLL. The user application should configure the PLL as needed and pass the configured CPUCLK value to Fapi_initializeAPI() function (details of this function are given later in this document).
- Always configure waitstates as per the device data manual before calling Flash API functions. The Flash API will issue an error if the waitstate configured by the application is not appropriate for the operating frequency of the application. See the Fapi_Set ActiveFlashBank() function for more details.
- Flash API execution is interruptible. However, there should not be any read/fetch access from the Flash bank on which an erase/program operation is in progress. In single-bank devices, 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. In dual-bank devices, Flash API can execute from one bank to perform erase/program operations on another bank.

```
//
// Erase a Sector
//
oReturnCheck = Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector,(uint32*)0x0080000);

//
// Wait until the erase operation is over
//
while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}
```

- Flash API does not configure (enable/disable) watchdog. The user application can configure watchdog and service it as needed. Hence, the Fapi_ServiceWatchdogTimer() function is no longer provided.
- Flash API uses EALLOW and EDIS internally as needed to allow/disallow writes to protected registers.
- The Main Array flash programming must be aligned to 64-bit address boundaries 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.
- There is no pump semaphore in TMS320F28004x devices.
- ECC should not be programmed for link-pointer locations. The API skips programming the ECC when the start address provided for the program operation is any of the three link-pointer addresses. API will use Fapi_DataOnly mode for programming these locations even if the user passes Fapi_AutoEccGeneration or Fapi_DataAndEcc mode as the programming mode parameter. The Fapi_EccOnly mode is not supported for programming these locations. The user application should exercise caution here. Care should be taken to maintain a separate structure/section for link-pointer locations in the application. Do not mix these fields with other DCSM OTP settings. If other fields are mixed with link-pointers, API will skip programming ECC for the non-link-pointer locations as well. This will cause ECC errors in the application.
- 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.
- 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 TMS320F28004x Piccolo Microcontrollers Technical 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().
- Flash API is embedded in ROM for this device. In order to use the Flash API in ROM, users must embed the F021_ROM_API_F28004x_FPU32.lib (ROM API symbols) library provided in C2000Ware at C2000Ware_x_xx_xx_xx\libraries\flash_api\f28004x\lib. When ROM API is used, there is no need to embed the F021_API_F28004x_FPU32.lib (software API) in your application for Flash erase/program purposes. When ROM API is used, make sure you do not allocate flash-load and RAM-run addresses for the API library in the linker command file since it already exists in ROM. However, any application functions that call the Flash API functions must be executed from RAM (for single bank devices) or from the other bank (for dual bank devices) on which a Flash erase/program operation is not targeted. Also note that there should not be any access to the Flash bank/OTP on which the Flash erase/program operation is in progress.

3 API Functions

3.1 Initialization Functions

3.1.1 Fapi_initializeAPI()

Initializes the Flash API

Synopsis

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

Parameters

poFlashControlRegister [in]	Pointer to the Flash 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 the System frequency or RWAIT is changed.

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

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include "F021_F28004x_C28x.h"

#define CPUCLK_FREQUENCY   100   /* 100 MHz System frequency */


int main(void)
{

     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
     //
     Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);


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

}

#pragma CODE_SECTION(Example_CallFlashAPI, ramFuncSection);
void Example_CallFlashAPI(void)
{
```

```
    Fapi_StatusType oReturnCheck;

    EALLOW;

    //
    // This function is required to initialize the Flash API based on
    // System frequency before any other Flash API operation can be performed
    // Note that the FMC register base address and system frequency are passed as the parameters
    //
    oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, 100);

    if(oReturnCheck != Fapi_Status_Success)
    {
         Example_Error(oReturnCheck);
    }

    //
    // Fapi_setActiveFlashBank function initializes Flash banks
    // and FMC for erase and program operations.
    // Note: It does not matter which bank is passed as the parameter to initialize.
    // Both Banks and FMC get initialized with one function call unlike F2837xS.
    // Hence, there is no need to execute Fapi_setActiveFlashBank() for each bank.
    //
    oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);

    if(oReturnCheck != Fapi_Status_Success)
    {
         Example_Error(oReturnCheck);
    }

    //
    // Bank0 Erase Program
    //

    /* User code for further Bank0 flash operations */
    .
    .
    .
    .

    //
    // Bank1 Erase Program
    //

    //
    // There is no need to call Fapi_initializeAPI() and Fapi_setActiveFlashBank()
    // when switching to Bank1 Flash operations
    //

    /* User code for further Bank1 flash operations */
    .
    .
    .
    .

     EDIS;

    //
    // Example is done here
    //

    Example_Done();

}
```

3.2 Flash State Machine Functions

3.2.1 Fapi_setActiveFlashBank()

Initializes the FMC for erase and program operations. There is only one FMC on F28004x devices despite there being two flash banks. Therefore, calling this function once, and using Fapi_FlashBank0 as the passed parameter is sufficient. There is no need to recall this function when switching flash banks.

Synopsis

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

Parameters

oNewFlashBank [in]	Fapi_FlashBank0 should be used. This will configure FMC for both banks. No need to call this function with Fapi_FlashBank1 as the 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.

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

See example provided in Section 3.1.1.

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)

Sample Implementation

```
#include “F021_F28004x_C28x.h”

#define CPUCLK_FREQUENCY   100   /* 100 MHz System frequency */


int main(void)
{
     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
     //
     Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);


     //
     // 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;


     EALLOW;

     //
     // This function is required to initialize the Flash API based on
     // System frequency before any other Flash API operation can be performed
     // Note that the FMC register base address and system frequency are passed as the parameters
     //
     oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, 100);

     if(oReturnCheck != Fapi_Status_Success)
     {
```

```
        Example_Error(oReturnCheck);
    }

    //
    // Fapi_setActiveFlashBank function initializes Flash banks
    // and FMC for erase and program operations.
    // Note: It does not matter which bank is passed as the parameter to initialize.
    // Both Banks and FMC get initialized with one function call unlike F2837xS.
    // Hence, there is no need to execute Fapi_setActiveFlashBank() for each bank.
    //
    oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error(oReturnCheck);
    }

    //
    // Bank0 Flash operations
    //

    //
    // Erase Bank0 Sector4
    //

    oReturnCheck = Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, (uint32 *)0x84000);

    //
    // Wait until FSM is done with erase sector operation
    //
    while(Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error (oReturnCheck);
    }

    //
    // Read FMSTAT 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 *)0x84000, 0x800,&oFlashStatusWord);


    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error(oReturnCheck);
    }

    //
    // * User code for further Bank0 flash operations *
    //
    .
    .
    .
    .
```

```
        //
        // Bank1 Flash operations
        //


        //
        // There is no need to call Fapi_initializeAPI() and Fapi_setActiveFlashBank()
        // when switching to Bank1 Flash operations
        //

        //
        // Erase Bank1 Sector2
        //
        oReturnCheck = Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector, (uint32 *)0x92000);

        //
        // Wait until FSM is done with erase sector operation
        //
        while(Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

        if(oReturnCheck != Fapi_Status_Success)
        {
             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 *)0x92000, 0x800, &oFlashStatusWord);


        if(oReturnCheck != Fapi_Status_Success)
        {
             Example_Error(oReturnCheck);
        }

        //
        // * User code for further Bank1 flash operations *
        //
        .
        .
        .
        .

        EDIS;

        //
        // Example is done here
        //
        Example_Done();

   }
```

3.2.3 Fapi_issueProgrammingCommand()

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

Synopsis

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

Parameters

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

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

Description

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

Table 6. Uses of Different Programming Modes (continued)

Programming mode (oMode)	Arguments used	Usage purpose
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 Fap_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, PAGE = 0, ALIGN(4)
.cinit  : > FLASH, PAGE = 0, ALIGN(4)
.const  : > FLASH, PAGE = 0, ALIGN(4)
.econst : > FLASH, PAGE = 0, ALIGN(4)
.pinit  : > FLASH, PAGE = 0, ALIGN(4)
.switch : > FLASH, PAGE = 0, ALIGN(4)
}
```

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

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

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

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

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

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

Arguments two and three are ignored when using this mode.

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

NOTE: This function does not check 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 should be used to monitor the status of an issued command.

Restrictions

- As described above, this function can program only a max of 128-bits (given the address provided is 128-bit aligned) at a time. If the user wants to program more than that, this function should be called in a loop to program 128-bits (or 64-bits as needed by application) at a time.
- The Main Array flash programming must be aligned to 64-bit address boundaries and each 64-bit word may only be programmed once per write or erase cycle.
- It is alright to program the data and ECC separately. However, each 64-bit dataword and 16-bit ECC word may only be programmed once per write or 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. 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 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)
- **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.

Sample Implementation

This example does not show the erase operation. Note that a sector should be erased before it can be reprogrammed.

```
#include “F021_F28004x_C28x.h”

#define CPUCLK_FREQUENCY   100   /* 100 MHz System frequency */


int main(void)
{
     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
     //
     Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);


     //
     // 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;


    EALLOW;


    //
    // This function is required to initialize the Flash API based on
    // System frequency before any other Flash API operation can be performed
    // Note that the FMC register base address and system frequency are passed as the parameters
    //
    oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, 100);

    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error(oReturnCheck);
    }

    //
    // Fapi_setActiveFlashBank function initializes Flash banks
    // and FMC for erase and program operations.
    // Note: It does not matter which bank is passed as the parameter to initialize.
    // Both Banks and FMC get initialized with one function call unlike F2837xS.
    // Hence, there is no need to execute Fapi_setActiveFlashBank() for each bank.
    //
    oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);

    if(oReturnCheck != Fapi_Status_Success)
    {
        Example_Error(oReturnCheck);
    }

    //
    // Bank0 Program
    //

    //
    // Program 0x200 16-bit words in Bank0 Sector 4
    //

    for(u32Index = 0x84000; (u32Index < 0x84200) &&
                            (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)
        {
             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 further Bank0 flash operations *
   //
   .
   .
   .
   .


   //
   // Bank1 Program
   //


   //
   // There is no need to call Fapi_initializeAPI() and Fapi_setActiveFlashBank()
   // when switching to Bank1 Flash operations
   //

   //
   // Program 0x200 16-bit words in Bank1 Sector 2
   //

   for(u32Index = 0x92000; (u32Index < 0x92200) &&
                           (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)
          {
               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 further Bank1 flash operations *
     //
     .
     .
     .
     .


     EDIS;

     //
     // Example is done here
     //
     Example_Done();

}
```

3.2.4 Fapi_issueProgrammingCommandForEccAddresses()

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

Synopsis

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

Parameters

pu32StartAddress [in]	ECC start address in Flash for the ECC to be programmed
pu16EccBuffer [in]	pointer to the ECC buffer address
u16EccBufferSizeInBytes [in]	number of bytes in the ECC buffer 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: Data 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.

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 and Erase Sector, when they are the current active command. Use Fapi_getFsmStatus() to check to see if the operation is successful.

Return Value

- **Fapi_Status_Success** (success)

3.2.6 Fapi_issueAsyncCommand()

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

Synopsis

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

Parameters

oCommand [in]	Command to issue to the FSM

Description

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

Below are the details of these commands:

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

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

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include “F021_F28004x_C28x.h”

#define CPUCLK_FREQUENCY   100   /* 100 MHz System frequency */


int main(void)
{
     //
     // Initialize System Control
     //
     Device_init();

     //
     // Call Flash Initialization to setup flash waitstates
     // This function must reside in RAM
```

```
        //
        Flash_initModule(FLASH0CTRL_BASE, FLASH0ECC_BASE, DEVICE_FLASH_WAITSTATES);


        //
        // 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;

        //
        // Bank0 operations
        //
        EALLOW;


        //
        // This function is required to initialize the Flash API based on
        // System frequency before any other Flash API operation can be performed
        // Note that the FMC register base address and system frequency are passed as the parameters
        //
        oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS, 100);

        if(oReturnCheck != Fapi_Status_Success)
        {
            Example_Error(oReturnCheck);
        }

        //
        // Fapi_setActiveFlashBank function initializes Flash banks
        // and FMC for erase and program operations.
        // Note: It does not matter which bank is passed as the parameter to initialize.
        // Both Banks and FMC get initialized with one function call unlike F2837xS.
        // Hence, there is no need to execute Fapi_setActiveFlashBank() for each bank.
        //
        oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);

        if(oReturnCheck != Fapi_Status_Success)
        {
            Example_Error(oReturnCheck);
        }

        //
        // Issue an async command
        //
        oReturnCheck = Fapi_issueAsyncCommand(Fapi_ClearMore);

        //
        // Wait until the Fapi_ClearMore operation is over
        //
        while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

        if(oReturnCheck != Fapi_Status_Success)
        {
```

```
            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();
        }

        //
        // * User code for further Bank0 and Bank1 flash operations *
        //
        .
        .
        .
        .



        EDIS;

        //
        // Example is done here
        //
        Example_Done();

    }
```

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 or in progress or suspended or failed. The user application should check the value of this register to determine if there is any failure after each erase and program operation.

Return Value

Table 7. 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 8. 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 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.

Table 8. FMSTAT Register Field Descriptions (continued)

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

3.3 Read Functions

3.3.1 Fapi_doBlankCheck()

Verifies region specified is erased value

Synopsis

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

Parameters

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

Description

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

Restrictions

The region being blank-checked cannot cross bank address boundary.

Return Value

- **Fapi_Status_Success (success)** - specified Flash locations are found to be in erased state
- **Fapi_Error_Fail** (failure: region specified is not blank)

3.3.2 Fapi_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.

Restrictions

The region being verified cannot cross bank address boundary.

Return Value

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

3.3.3 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

The region that the PSA is being calculated on cannot cross bank address boundary

Return Value

- **PSA value**

3.3.4 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 value is returned in the poFlashStatusWord parameter.

Restrictions

The region being verified cannot cross bank address boundary.

Return Value

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

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
- u8ApiMinorVersion – Minor version number of this compile of the API. Minor version is 56 for F28004x devices.
- u8ApiRevision – Revision version number of this compile of the API. The latest API revision is 1. In TMS320F28004x FlashAPI V.1.56.00.00, Fapi_doPsaVerify() and Fapi_CalculatePsa() do not support ECC memory address range. This limitation does not exist in V.1.56.01.00.
- 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. This field returns a value of 0x4.
- u8ApiTechnologyRevision – Indicates the revision of the technology supported by the API
- u8ApiEndianness – This field always returns as 1 (Little Endian) for F28004x devices.
- u32ApiCompilerVersion – Version number of the Code Composer Studio code generation tools used to compile the API

Return Value

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

3.5 Utility Functions

3.5.1 Fapi_flushPipeline()

Flushes the 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 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 to calculate ECC on (should be in little endian order)

Description

This function will calculate the ECC for a 64-bit aligned word including address. There is no need to provide a left-shifted address to this function anymore. TMS320F28004x Flash API takes care of it.

Return Value

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

3.5.3 Fapi_isAddressEcc()

Indicates is 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)

3.5.4 Fapi_remapEccAddress()

Takes ECC address and remaps it to main address space

Synopsis

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

Parameters

u32EccAddress [in]	ECC address to remap

Description

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

Return Value

- **32-bit Main Flash Address**

3.5.5 Fapi_calculateFletcherChecksum()

Calculates the Fletcher checksum from the given address and length

Synopsis

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

Parameters

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

Description

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

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) on a device. Please refer to Section 3.2.2 for further information.

Figure 1. Recommended Erase Flow

4.3 Recommended Program Flow

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

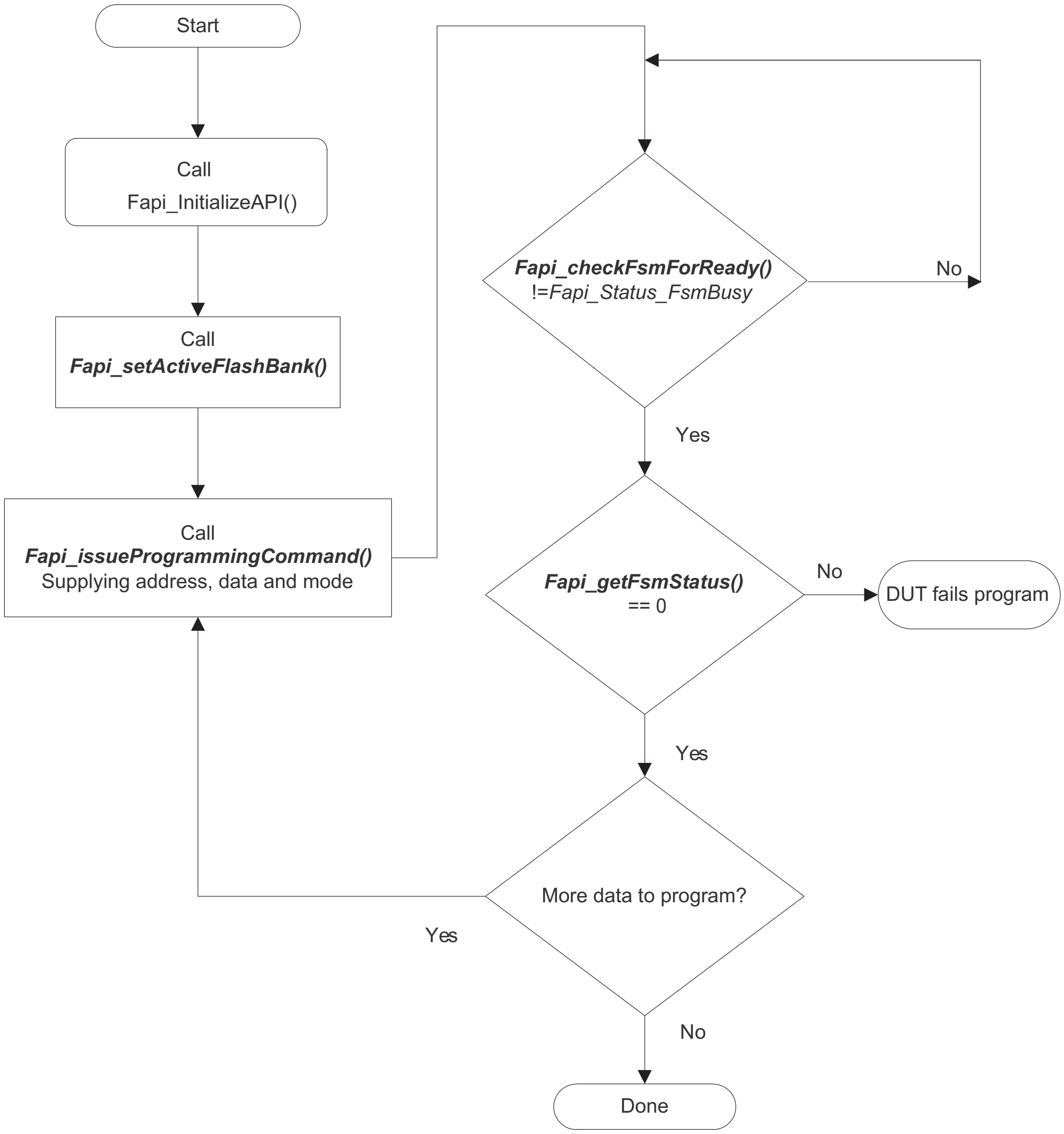

Figure 2. Recommended Program Flow

Appendix A
SPNU628A–February 2017–Revised December 2017

Flash State Machine Commands

A.1 Flash State Machine Commands

Table 9. 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_issueProgrammingCommandForEccAddress()
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 TMS320F28004x Flash API Library

Table 10. 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 *Includes references to the following functions* • Fapi_isAddressEcc • Fapi_serviceWatchdogTimer	TBD	TBD
Fapi_checkFsmForReady	TBD	TBD
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	TBD	TBD
Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	TBD	TBD
Fapi_flushPipeline *Includes references to the following functions* • Fapi_waitDelay	TBD	TBD
Fapi_getFsmStatus	TBD	TBD
Fapi_getLibraryInfo	TBD	TBD
Fapi_initializeAPI	TBD	TBD
Fapi_isAddressEcc	TBD	TBD
Fapi_issueAsyncCommand	TBD	TBD
Fapi_issueAsyncCommandWithAddress *Includes references to the following functions* • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	TBD	TBD
Fapi_issueFsmSuspendCommand	TBD	TBD
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	TBD	TBD

Table 10. C28x Function Sizes and Stack Usage (continued)

Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable • Fapi_remapEccAddress	TBD	TBD
Fapi_remapEccAddress	TBD	TBD
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	TBD	TBD

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;

#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 (defined(__TMS320C28xx__) && __TI_COMPILER_VERSION__ < 6004000)
#if !defined(__GNUC__)
#error "F021 Flash API requires GCC language extensions. Use the –gcc option."
#endif
#endif

#ifndef TRUE
```

```
   #define TRUE           1
#endif

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

C.3 Enumerations

C.3.1 Fapi_CpuType

This is used to indicate what type of Cpu is being used.

```
typedef enum
{
   Cpu_ARM7,
   Cpu_M3,
   Cpu_R4,
   Cpu_R4F,
   Cpu_C28,
   Cpu_Undefined
}  ATTRIBUTE_PACKED Fapi_CpuType;
```

C.3.2 Fapi_FlashProgrammingCommandsType

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

```
typedef enum
{
Fapi_AutoEccGeneration,   /* Command will autogenerate 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.3.3 Fapi_FlashBankType

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

```
typedef enum
{
   Fapi_FlashBank0,
   Fapi_FlashBank1, /*Not used in TMS320F28004x Flash API*/
   Fapi_FlashBank2, /*Not used in TMS320F28004x Flash API*/
   Fapi_FlashBank3, /*Not used in TMS320F28004x Flash API*/
   Fapi_FlashBank4, /*Not used in TMS320F28004x Flash API*/
   Fapi_FlashBank5, /*Not used in TMS320F28004x Flash API*/
   Fapi_FlashBank6, /*Not used in TMS320F28004x Flash API*/
   Fapi_FlashBank7  /*Not used in TMS320F28004x Flash API*/
}  ATTRIBUTE_PACKED Fapi_FlashBankType;
```

C.3.4 Fapi_FlashStateCommandsType

This contains all the possible Flash State Machine commands.

```
typedef enum
{
   Fapi_ProgramData    = 0x0002,
   Fapi_EraseSector    = 0x0006,
   Fapi_EraseBank      = 0x0008, /*Not available in TMS320F28004x devices */
   Fapi_ValidateSector = 0x000E, /*Not available in TMS320F28004x devices */
   Fapi_ClearStatus    = 0x0010,
   Fapi_ProgramResume  = 0x0014,
   Fapi_EraseResume    = 0x0016,
   Fapi_ClearMore      = 0x0018
```

```
} ATTRIBUTE_PACKED Fapi_FlashStateCommandsType;
```

C.3.5 Fapi_FlashReadMarginModeType

This contains all the possible Flash State Machine commands.

```
typedef enum
{
   Fapi_NormalRead = 0x0,
   Fapi_RM0        = 0x1, /*Technology used in TMS320F28004x does not need this*/
   Fapi_RM1        = 0x2  /*Technology used in TMS320F28004x does not need this*/
} ATTRIBUTE_PACKED Fapi_FlashReadMarginModeType;
```

C.3.6 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 SYSCLK and RWAIT */
   Fapi_Error_InvalidCpu,           /* Returned if the specified Cpu does not exist */
   Fapi_Error_InvalidBank,          /* Returned if the specified bank does not exist */
   Fapi_Error_InvalidAddress,       /* Returned if the specified Address does not exist in Flash
                                       or OTP */
   Fapi_Error_InvalidReadMode,      /* Returned if the specified read mode does not exist */
   Fapi_Error_AsyncIncorrectDataBufferLength,
   Fapi_Error_AsyncIncorrectEccBufferLength,
   Fapi_Error_AsyncDataEccBufferLengthMismatch,
   Fapi_Error_FeatureNotAvailable,  /* FMC feature is not available on this device */
   Fapi_Error_FlashRegsNotWritable  /* Returned if Flash registers are not writable due to
                                       security */
} ATTRIBUTE_PACKED Fapi_StatusType;}  ATTRIBUTE_PACKED Fapi_StatusType;
```

C.3.7 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;
```

Parallel Signature Analysis (PSA) Algorithm

D.1 Function Details

The functions Section 3.3.4 and Section 3.3.3 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 + X^2 + X^22 + X^31

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

ECC Calculation Algorithm

E.1 Function Details

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

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

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

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

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

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

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

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

                      //
                      //Fold the masked data into a single bit for parity calculation.
                      //The result will be in the LSB.
                      //
```

```
            xorData = xorData ^ (xorData >> 32);
            xorData = xorData ^ (xorData >> 16);
            xorData = xorData ^ (xorData >> 8);
            xorData = xorData ^ (xorData >> 4);
            xorData = xorData ^ (xorData >> 2);
            xorData = xorData ^ (xorData >> 1);

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

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

Revision History

Changes from February 28, 2017 to December 15, 2017	Page
• This document has undergone extensive changes. It is advised that you review each section for updates.	4

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

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