C2000 F021 Flash API

Version 1.53

Reference Guide

TEXAS INSTRUMENTS

Literature Number: SPNU595
January 2014

Contents

List of Figures

List of Tables

1 Introduction

This reference guide provides a detailed description of Texas Instruments' F021 Flash API functions that can be used to erase, program and verify F021 Flash on TI devices.

1.1 Reference Material

Use this guide in conjunction with the device-specific data sheet 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

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 C2000 F021 Flash API Overview

2.1 Introduction

The F021 Flash API is a library of routines that when called with the proper parameters in the proper sequence, erases, programs, or verifies Flash memory on Texas Instruments microcontrollers using the F021 process. On ARM Cortex devices, these routines must be run a in a privilege mode (a mode other than user) to allow access to the Flash memory controller registers. Additionally, for C2000 devices, enabling writes to protected mode registers must be done before calling these routines. The API verifies for the selected bank, that the appropriate RWAIT or EWAIT value is set for the specified system frequency.

NOTE: Please refer to the C2000 device-specific technical reference manual and System Control and Interrupts Reference Guide for more details.

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 status register value from the Flash memory controller
Fapi_checkFsmForReady()	Returns whether or not the Flash state machine is ready or busy
Fapi_connectFlashPumpToCpu()[1]	Connects the Flash pump to the active Flash memory controller
Fapi_enableBanksForOtpWrite()[1]	Enables banks to allow programming of customer OTP
Fapi_disableBanksForOtpWrite()[1]	Disables all banks from programming customer OTP
Fapi_setActiveFlashBank()	Sets the active bank for a erase or program command
Fapi_issueFsmSuspendCommand()	Suspends FSM commands program data, erase sector and erase bank
Fapi_writeEwaitValue()[1]	Writes value to EWait register
Fapi_flushPipeline()	Flushes the pipeline buffers in the Flash memory controller
Fapi_isAddressEcc()	Determines if address falls in Flash memory controller ECC ranges
Fapi_remapEccAddress()	Remaps an ECC address to corresponding main address

(1) Not applicable for Concerto and F2837xD devices.

Table 3. Summary of Asynchronous Operation Functions

API Function	Description
Fapi_issueAsyncCommandWithAddress()	Issues a command to FSM for operations that require an address
Fapi_issueAsyncCommand()	Issues a command 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
Fapi_issueProgrammingCommandForEccAddress()[1]	Remaps an ECC address to the main data space and then call Fapi_issueProgrammingCommand()

(1) This function is not supported in F2837xD devices for now.

Table 5. Summary of Read Functions

API Function	Description
Fapi_doVerify()	Verifies specified Flash memory range against supplied values
Fapi_doVerifyByByte()[1]	Verifies specified Flash memory range against supplied values by byte
Fapi_doBlankCheck()	Verifies specified Flash memory range against erased state
Fapi_doBlankCheckByByte()[1]	Verifies specified Flash memory range against erased state by byte
Fapi_doMarginRead()	Reads a specified Flash memory range using the specified read-margin mode
Fapi_doMarginReadByByte()[1]	Reads a specified Flash memory range using the specified read-margin mode by byte
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
Note: These functions are not supported for F2837xD ECC memory space.	

(1) Not applicable for C28x cores.

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()	Returns the information specific to the device the API library is being executed on
Fapi_getBankSectors()[1]	Returns the sector information for a bank

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

Table 7. Summary of User Defined Functions

API Function	Description
Fapi_serviceWatchdogTimer()	User modifiable function to service watchdog timer
Fapi_setupEepromSectorEnable()[1]	User modifiable function to enable the EEPROM bank sectors for programming and erase
Fapi_setupBankSectorEnable()[2]	User modifiable function to enable the non EEPROM banks sectors for programming and erase

(1) Not applicable for Concerto and F2837xD devices.
(2) Not applicable for Concerto 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 and64bit word
Fapi_waitDelay()	Creates a delay based on System frequency

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 device is first powered up, the function *Fapi_initializeAPI()* must be called before any other API function can be used except for the functions *Fapi_getLibraryInfo()* and *Fapi_getDeviceInfo()*. This initializes the API internal structures.

2.3.1.2 Bank Setup

Before performing a Flash operation for the first time or on a different Bank than is the current active Bank, the function *Fapi_setActiveFlashBank()* must be called.

2.3.1.3 On System Frequency Change

If the System operating frequency is changed after the initial call to the function 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

All ARM Cortex Flash API object files are distributed in the ARM standard EABI elf object format. For the CortexR4 cores, the library files are built using Thumb2 mode. All C28x Flash API object files are distributed in the standard TI COFF object format

NOTE: As of release of v1.51.0 of the API, compilation with the TI ARM codegen tools requires "Enable support for GCC extensions" option to be enabled.

2.3.2.2 Distribution Files

The following API files are distributed with the installer:

- Library Files
  - F021_API_CortexM3_LE.lib – This is the Flash API object file for Cortex M3 Little Endian devices (supports master subsystem in F28M35x/F28M36x).
  - F021_API_C28x.lib – This is the Flash API object file for C28x devices (supports control subsystem in F28M35x/F28M36x).
  - F021_API_C28x_FPU32.lib – This is the Flash API object file for C28x devices (supports control subsystem in F28M35x/F28M36x) that are using floating point unit
  - F021_API_F2837xD_C28x.lib – This is the Flash API object file for C28x devices (supports CPU1 subsystem and CPU2 subsystem in F2837xD).
  - F021_API_F2837xD_C28x_FPU32.lib – This is the Flash API object file for C28x devices (supports CPU1 subsystem and CPU2 subsystem in F2837xD) that are using 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.*
- Include Files
  - Only one of the following include files is used by the customer’s code based on the device that it is being developed for. These files set up compile-specific defines and then includes the F021.h master include file.
    - F021_Concerto_C28x.h – The master include file for Concerto C28x F021 devices.
    - F021_Concerto_Cortex.h – The master include file for Concerto Cortex M3 F021 devices.
    - F021_F2837xD_C28x.h – The master include file for F2837xD C28x F021 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 public API functions and includes all other include files.
  - Helpers.h – Set of helper defines.
  - Init.h – Defines the API initialization structure.
  - Registers_Concerto_C28x.h – Little Endian Flash memory controller registers structure for

Concerto C28x F021 devices.
- Registers_Concerto_Cortex.h – Little Endian Flash memory controller registers structure for Concerto Cortex M3 F021 devices.
- Registers_C28x.h – Little Endian Flash memory controller registers structure for C28x F021 devices (supports F2837xD).
- 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 all F021 devices.
- Constants/Concerto.h – Constant definitions for Concerto F021 devices.
- Constants/F2837xD.h – Constant definitions for F2837xD F021 devices.

2.3.3 Executing API From Flash

The F021 Flash API library cannot be executed from the same bank as the active bank selected for the API commands to operate on. On single bank devices, the F021 Flash API must be executed from RAM.

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 a different Flash Memory Controller is to be used on devices that have multiple Flash Memory Controllers, System frequency is changed, RWAIT or EWAIT values are changed.

NOTE: RWAIT and EWAIT register values must be set before calling this function.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include “F021.h”
#define HCLK_FREQUENCY 80 /* 80 MHz  System frequency */
int main(void)
{
 Fapi_StatusType oReturnCheck;

 oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS,HCLK_FREQUENCY);

 /* User code for 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. The value returned is not interpreted and it is up to the user's code to handle the return value.

Return Value

Table 9. FMSTAT Register

Bits 17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RV SUSP	RDVE R	RVF	ILA	DBT	PGV	PCV	EV	CV	Busy	ERS	PGM	INV DAT	CSTA T	Volt Stat	ESUS P	PS US P	SLOCK [1]

[1] SLOCK bit is not applicable for Concerto and F2837xD devices.

Table 10. FMSTAT Register Field Descriptions

Bit	Field	Description
17	RVSUSP	Read Verify Suspend. When set, this bit indicates that the flash module has received and processed a suspend command during a read-verify operation. This bit remains set until the read-verify-resume command has been issued or the Clear_More command is run.
16	RDVER	Read verify command currently underway. When set, this bit indicates that the flash module is actively performing a read-verify operation. This bit is set when read-verify starts and is cleared when it is complete. It is also cleared when the read-verify is suspended and set when the read-verify resumes.
15	RVF	Read Verify Failure When set, indicates that a read verify mismatch is detected using the Read Verify command. This bit remains set until clear_status or clear_more FSM commands are run.
14	ILA	Illegal Address When set, indicates that an illegal address is detected. Three conditions can set illegal address flag. • Writing to a hole (un-implemented logical address space) within a flash bank. • Writing to an address location to an un-implemented flash space. • Input address for write is decoded to select a different bank from the bank ID register. • The address range does not match the type of FSM command. For example, the erase_sector and erase_OTP commands must match the address regions. • TI-OTP address selected but CMD_EN in FSM_ST_MACHINE is not set.
13	DBF	Disturbance Test Fail. This bit sets during a Program Sector command when the FSM first reads an address and it is not all ones. Controlled by DIS_TST_EN of the FSM_ST_MACHINE register.
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	PCV	Precondition verify When set, indicates that a sector is not successfully preconditioned (pre-erased) after the maximum allowed number of program pulses are given for precondition operation for any applied command such as Erase Sector command. During Precondition verify command, this flag is set immediately if a flash bit is found to be 1. If Precondition Terminate Enable bit is cleared then PCV is not set when preconditioning fails. Setting Precondition Terminate Enable bit allows FSM to terminate the command immediately if precondition operation fails during any type of erase commands.

Table 10. FMSTAT Register Field Descriptions (continued)

Bit	Field	Description
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	CV	Compact Verify When set, indicates that a sector contains one or more bits in depletion after an erase operation with CMPV_ALLOWED set. During compact verify command, this flag is set immediately if a bit is found to be 1.
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, erase, or validate sector 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.Version 3.0.9, 3.1.0 Preliminary Flash Registers
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	SLOCK	Sector Lock Status. When set, this bit indicates that the operation was halted because the target sector was locked for erasing and the programming either by the sector protect bit or by OTP write protection disable bits. This bit is cleared by the Clear Status command. No SLOCK FSM error will occur if all sectors in a bank erase operation are set to 1. All the sectors will be checked but no LOCK will be set if no operation occurs due to the SECT_ERASED bits being set to all ones. A SLOCK error will occur if attempting to do a sector erase with either BSE is cleared or SECT_ERASED is set. For FLEE flash banks over 16 sectors, the BSE register must be set to all ones for a bank or sector erase.

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_connectFlashPumpToCpu()

Connects the Flash Pump to specified CPU FMC

Restrictions

This function is not applicable for Concerto and F2837xD devices. In Concerto and F2837xD devices, there is only one pump to be shared by two Flash banks. A Pump semaphore register is provided to gain access to pump. Refer to the Concerto and F2837xD device-specific technical reference manual.

Synopsis

```
Fapi_StatusType Fapi_connectFlashPumpToCpu(
         Fapi_CpuSelectorType oFlashCpu)
```

Parameters

oFlashCpu [in]	CPU to connect Flash Pump to

Description

This is a placeholder function for future devices that will have multiple Flash Memory Controllers and only one Flash pump.

Return Value

- **Fapi_Status_Success** (success)

3.2.4 Fapi_enableBanksForOtpWrite()

Enables programming of Customer OTP for specified banks

Restrictions

This function is not applicable for Concerto and F2837xD devices.

Synopsis

```
Fapi_StatusType Fapi_enableBanksForOtpWrite(
         uint8 u8Banks)
```

Parameters

u8Banks [in]	Bit mask indicating each bank to be enabled for OTP programming

Description

This function sets up the **OTPPRODIS** field in the **FBAC** register to enable Customer OTP programming for the banks specified in the bitfield mask u8Banks.

Return Value

- **Fapi_Status_Success** (success)

3.2.5 Fapi_disableBanksForOtpWrite()

Disables programming of Customer OTP

Restrictions

This function is not applicable for Concerto and F2837xD devices.

Synopsis

```
Fapi_StatusType Fapi_enableBanksForOtpWrite(void)
```

Parameters

None

Description

This function sets **OTPPRODIS** field in the **FBAC** register to disable Customer OTP programming for all banks.

Return Value

- **Fapi_Status_Success** (success)

3.2.6 Fapi_setActiveFlashBank()

Changes the active Flash Bank

Synopsis

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

Parameters

oNewFlashBank [in]	Bank number to set as active

Description

This function sets the active bank for further Flash operations to be performed on that bank and sets up the Flash Memory Controller for that 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.h"
int main(void)
{
 Fapi_StatusType oReturnCheck;

 oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);

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

3.2.7 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 following FSM commands Program Data, Erase Sector, and Erase Bank 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)

3.2.8 Fapi_writeEwaitValue()

Writes value to the EWAIT register location

Restrictions

This function is not applicable for Concerto and F2837xD devices.

NOTE: This function will not function correctly if called before Fapi_initializeAPI(). To write the EWAIT value before the API is initialized, use the macro FAPI_WRITE_LOCKED_FSM_REGISTER(<Register Address>, <Value>).

Synopsis

```
Fapi_StatusType Fapi_writeEwaitValue(
        uint32 u32Ewait)
```

Parameters

u32Ewait [in]	EWAIT value to write

Description

This function writes the supplied EWAIT value to the EEPROM_CONFIG register in the FMC.

Return Value

- **Fapi_Status_Success** (success)

3.2.9 Fapi_flushPipeline()

Flushes the FMC pipeline buffers

Synopsis

```
void Fapi_flushPipeline(void)
```

Parameters

None

Description

For non-C2000 devices this function flushes the FMC pipeline buffers and for C2000 devices, this function flushes the FMC data cache. The pipeline or data cache must be flushed before the first non-API Flash read after an operation that modifies the Flash contents (erasing and programming).

Return Value

None

3.2.10 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 Flash address for the given ECC Flash address.

Return Value

- **32-bit Main Flash Address**

3.2.11 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)

3.3 Asynchronous Functions

3.3.1 Fapi_issueAsyncCommandWithAddress()

Issues a command to the Flash State Machine with an address

Synopsis

```
Fapi_StatusType Fapi_issueAsyncCommandWithAddress(
         Fapi_FlashStateCommandsType oCommand,
         uint32 *pu32StartAddress)
```

Parameters

oCommand [in]	Command to issue to the FSM
pu32StartAddress [in]	Address for needed for Flash State Machine operation

Description

This function issues a command to the Flash State Machine for commands requiring an address to function correctly. Primary commands used with function are Erase Sector and Erase Bank.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include “F021.h”
#define HCLK_FREQUENCY 80 /* 80 MHz System frequency */
int main(void)
{
 Fapi_StatusType oReturnCheck;

  oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS,HCLK_FREQUENCY);
  oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
  oReturnCheck = Fapi_issueAsyncCommandWithAddress(Fapi_EraseSector,
                                                (uint32 *) 0);
 while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

 /* User code for 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.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include “F021.h”
#define HCLK_FREQUENCY 80 /* 80 MHz System frequency */
int main(void)
{
 Fapi_StatusType oReturnCheck;

  oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS,HCLK_FREQUENCY);
  oReturnCheck = Fapi_issueAsyncCommand(Fapi_ClearStatus);

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

3.4 Program Functions

3.4.1 Fapi_issueProgrammingCommand()

3.4.1.1 For ARM Cortex devices

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

Synopsis

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

Parameters

pu32StartAddress [in]	start address in Flash for the data and ECC to be programmed	
pu8DataBuffer [in]	pointer to the Data buffer address	
u8DataBufferSizeInBytes [in]	number of bytes in the Data buffer	
pu8EccBuffer [in]	pointer to the ECC buffer address	
u8EccBufferSizeInBytes [in]	number of bytes in the ECC buffer	
oMode [in]	Indicates the programming mode to use:	
	Fapi_DataOnly	Programs only the data buffer
	Fapi_AutoEccGeneration	Programs the data buffer and auto generates and programs the ECC.
	Fapi_DataAndEcc	Programs bot 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 and handles multiple bank widths automatically.

Programming modes:

Fapi_DataOnly – This mode will only program the data portion in Flash at the address specified. It can program from 1 byte up to the bank width (8,16,32) bytes based on the bank architecture. The supplied starting address to program at plus the data buffer length cannot exceed the bank data width. (Ex. Programming 14 bytes on a 16 byte wide bank starting at address 0x4 is not allowed.)

Fapi_AutoGeneration – This will program the supplied data portion in Flash along with automatically generated ECC. The ECC is calculated for the data width of the bank and data not supplied is treated as 0xFF. The data restrictions for Fapi_DataOnly also exist for this option.

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 64bit word and the length of data must correlate to the supplied ECC. (For example, data buffer length is 8 bytes, the ECC buffer must be 1 byte).

Fapi_EccOnly – This mode will only program the ECC portion in Flash at the address specified. It can program from 1 byte up to the bank ECC width (1, 2, 4) bytes based on the bank architecture. The supplied starting address to program at plus the ECC buffer length cannot exceed the bank ECC width. (Ex. Programming 3 bytes on a 2 byte ECC wide bank starting at address 0x0 is not allowed.)

NOTE: The length of pu8DataBuffer and pu8EccBuffer cannot exceed the bank width of the current active bank. Please refer to the device data sheet for the width of the banks.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (failure: Data buffer size specified exceeds Data bank width)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: ECC buffer size specified exceeds ECC bank width)
- **Fapi_Error_AsyncDataEccBufferLengthMismatch** (failure: Data buffer size either is not 64bit aligned or Data length exceeds amount ECC supplied)

Sample Implementation

```
#include “F021.h”
int main(void)
{
   uint8 au8DataBuffer[16] = {0x00, 0x01, 0x02, 0x03, 0x04,0x05, 0x06, 0x07,0x08, 0x09,
                                  0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};
   Fapi_StatusType oReturnCheck;
         Fapi_FlashBankType oActiveFlashBank = Fapi_FlashBank0;
   uint32 u32Index;

    oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS,80);
    if(oReturnCheck == Fapi_Status_Success)
    {
       oReturnCheck = Fapi_setActiveFlashBank(oActiveFlashBank);
       if(oReturnCheck == Fapi_Status_Success)
       {
             for(u32Index = 0;
             (u32Index < 0x40) && (oReturnCheck == Fapi_Status_Success);
             u32Index+= 0x10)
       {
         oReturnCheck = Fapi_issueProgrammingCommand(
                              (uint32 *) u32Index,
                              au8DataBuffer,
                              0x10,
                              0,
                              0,
                              Fapi_AutoEccGeneration);
              while(Fapi_checkFsmForReady() == Fapi_Status_FsmBusy);
              oReturnCheck = Fapi_getFsmStatus();
       }
        }
     }
 }
```

3.4.1.2 For C28x devices

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

Synopsis

```
Fapi_StatusType Fapi_issueProgrammingCommand(
         uint32 *pu32StartAddress,
         uint16 *pu16DataBuffer,
         uint16  u16DataBufferSizeInBytes,
         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	
u16DataBufferSizeInBytes [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 bot 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 and handles multiple bank widths automatically. 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.

Programming modes:

Fapi_DataOnly – This mode will only program the data portion in Flash at the address specified. It can program from 1 word up to the bank width (4,8,16) words based on the bank architecture. The supplied starting address to program at plus the data buffer length cannot exceed the bank data width. (Ex. Programming 7 words on an 8 word wide bank starting at address 0x100004 is not allowed.)

Fapi_AutoGeneration – This will program the supplied data portion in Flash along with automatically generated ECC. The ECC is calculated for the data width of the bank and data not supplied is treated as 0xFF. The data restrictions for Fapi_DataOnly also exist for this option.

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 64bit 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).

Fapi_EccOnly – This mode will only program the ECC portion in Flash at the address specified. It can program from 1 byte up to the bank ECC width (1, 2, 4) bytes based on the bank architecture. The supplied starting address to program at plus the ECC buffer length cannot exceed the bank ECC width. (Ex. Programming 3 bytes on a 2 byte ECC wide bank starting at address 0x100000 is not allowed.)

NOTE: The length of pu8DataBuffer and pu8EccBuffer cannot exceed the bank width of the current active bank. Please refer to the device data sheet for the width of the banks.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectDataBufferLength** (failure: Data buffer size specified exceeds Data bank width)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: ECC buffer size specified exceeds ECC bank width)
- **Fapi_Error_AsyncDataEccBufferLengthMismatch** (failure: Data buffer size either is not 64bit aligned or Data length exceeds amount ECC supplied)

Sample Implementation

```
#include “F021.h”
int main(void)
{
   uint16 au16DataBuffer[8] = {0x0001, 0x0203, 0x0405, 0x0607, 0x0809, 0x0A0B, 0x0C0D, 0x0E0F};
   Fapi_FlashBankType oActiveFlashBank = Fapi_FlashBank0;
   Fapi_StatusType oReturnCheck;
   uint32 u32Index;

    oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS,150);
    if(oReturnCheck == Fapi_Status_Success)
    {
       oReturnCheck = Fapi_setActiveFlashBank(oActiveFlashBank);
       if(oReturnCheck == Fapi_Status_Success)
       {
             for(u32Index = 0x100000;
             (u32Index < 0x100040) && (oReturnCheck == Fapi_Status_Success);
             u32Index+= 0x8)
       {
         oReturnCheck = Fapi_issueProgrammingCommand(
                                (uint32 *) u32Index,
                                au16DataBuffer,
                                0x8,
                                0,
                                0,
                                Fapi_AutoEccGeneration);
              while(Fapi_checkFsmForReady() == Fapi_Status_FsmBusy);
              oReturnCheck = Fapi_getFsmStatus();
       }
        }
      }
 }
```

3.4.2 Fapi_issueProgrammingCommandForEccAddress()

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

3.4.2.1 *For ARM Cortex devices*

Synopsis

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

Parameters

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

Description

This function will remap an address in the ECC memory space to the corresponding data address space and then call Fapi_issueProgrammingCommand() to program the supplied ECC data. The same limitations for Fapi_issueProgrammingCommand() using Fapi_EccOnly mode applies to this function.

NOTE: The length of pu8EccBuffer cannot exceed the bank width of the current active bank. Please refer to the device data sheet for the width of the banks.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: Data buffer size specified exceeds ECC bank width)

3.4.2.2 *For C28x devices*

Synopsis

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

Restrictions

This function is not supported for F2837xD devices for now.

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

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.

NOTE: The length of pu8EccBuffer cannot exceed the bank width of the current active bank. Please refer to the device data sheet for the width of the banks.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_AsyncIncorrectEccBufferLength** (failure: Data buffer size specified exceeds ECC bank width)

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)
```

Restrictions

This function is not supported for F2837xD ECC memory space.

Parameters

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

Description

This function checks the device for blank (erase state) starting at the specified address for the length of 32bit words specified. If a non-blank location is found, these results will be returned in the poFlashStatusWord parameter. This will use normal read, read margin 0 and read margin 1 modes checking for blank.

Restrictions

The region being blank checked cannot cross bank address boundary.

Return Value

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

3.5.2 Fapi_doBlankCheckByByte()

Verifies region specified is erased value by byte

Restrictions

This function is not applicable for C28x cores.

Synopsis

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

Parameters

pu8StartAddress [in]		start address for region to blank check
u32Length [in]		length of region in 32bit 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 0xFF)
	->au32StatusWord[3]	indicates read mode that failed blank check

Description

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

Restrictions

The region being blank checked cannot cross bank address boundary.

Return Value

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

3.5.3 Fapi_doVerify()

Verifies region specified against supplied data

Synopsis

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

Restrictions

This function is not supported for F2837xD ECC memory space.

Parameters

pu32StartAddress [in]		start address for region to verify
u32Length [in]		length of region in 32bit words to verify
pu32CheckValueBuffer [in]		address of buffer to verify region against
poFlashStatusWord [out]		returns the status of the operation if result is not Fapi_Status_Success
	->au32StatusWord[0]	address of first verify failure location
	->au32StatusWord[1]	data read at first verify failure location
	->au32StatusWord[2]	value of compare data
	->au32StatusWord[3]	indicates read mode that failed verify

Description

This function verifies the device against the supplied data starting at the specified address for the length of 32bit 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

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)

3.5.4 Fapi_doVerifyByByte()

Verifies region specified against supplied data by byte

Restrictions

This function is not applicable for C28x cores.

Synopsis

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

Parameters

pu8StartAddress [in]		start address for region to verify by byte
u32Length [in]		length of region in 8bit words to verify
pu8CheckValueBuffer [in]		address of buffer to verify region against by byte
poFlashStatusWord [out]		returns the status of the operation if result is not Fapi_Status_Success
	->au32StatusWord[0]	address of first verify failure location
	->au32StatusWord[1]	data read at first verify failure location
	->au32StatusWord[2]	value of compare data
	->au32StatusWord[3]	indicates read mode that failed verify

Description

This function verifies the device against the supplied data by byte starting at the specified address for the length of 8bit 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 checking for verifying the data.

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)

3.5.5 Fapi_doPsaVerify()

Verifies region specified against specified PSA value

Synopsis

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

Restrictions

This function is not supported for F2837xD ECC memory space.

Parameters

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

Description

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

Restrictions

The region being verified checked cannot cross bank address boundary.

Return Value

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

3.5.6 Fapi_calculatePsa()

Calculates the PSA for a specified region

Synopsis

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

Restrictions

This function is not supported for F2837xD ECC memory space.

Parameters

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

Description

This function calculates the PSA value for the region specified starting at pu32StartAddress for u32Length 32bit words using u32PsaSeed value in the margin mode specified.

Restrictions

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

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_InvalidReadMode** (failure: read mode specified is not valid)

3.5.7 Fapi_doMarginRead()

Reads a region of Flash Memory using specified margin mode

Synopsis

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

Restrictions

This function is not supported for F2837xD ECC memory space.

Parameters

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

Description

This function reads the region specified starting at pu32StartAddress for u32Length 32bit words using pu32ReadBuffer to store the read values.

NOTE: The region that is being read cannot cross bank address boundary.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_InvalidReadMode** (failure: read mode specified is not valid)

3.5.8 Fapi_doMarginReadByByte()

Reads a region of Flash Memory using specified margin mode by byte

Restrictions

This function is not applicable for C28x cores.

Synopsis

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

Parameters

pu8StartAddress [in]	start address for region to read by byte
pu8ReadBuffer [out]	address of buffer to return read data by byte u32
Length [in]	length of region in 32bit words to read
oReadMode [in]	indicates which margin mode (normal, RM0, RM1) to use

Description

This function reads the region specified starting at pu8StartAddress for u32Length 8bit words using pu8ReadBuffer to store the read values.

Restrictions

The region that is being read cannot cross bank address boundary.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_InvalidReadMode** (failure: read mode specified is not valid)

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 52 for F28M35x and F28M36x devices. Minor version is 53 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. F021 is tech type of 0x20
- u8ApiTechnologyRevision – Indicates the revision of the Technology supported by the API
- u8ApiEndianness – Indicates if this compile of the API is for Big Endian or Little Endian memory
- 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

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

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)
```

Restrictions

This function is not supported for F2837xD ECC memory space.

Parameters

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

Description

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

Return Value

- • 32bit Fletcher Checksum value

3.7.2 Fapi_calculateEcc()

Calculates the ECC for a 64bit value

Synopsis

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

Parameters

u32Address [in]	Address of the 64bit value to calculate the ECC
u64Data [in]	64bit value to calculate ECC on

Description

This function will calculate the ECC for a 64bit aligned word including address if device supports address in ECC calculation.

NOTE: As of version 1.51.0 of the API, this function expects the value in u64Data to be in the natural Endianness of the system the function is being called on.

Return Value

- 8bit calculated ECC

3.7.3 Fapi_waitDelay()

Generates a delay for amount specified

Synopsis

```
uint8 Fapi_waitDelay(
              uint32 u32WaitDelay)
```

Parameters

u32WaitDelay [in]	Number of <to be characterized> us increments to delay

Description

This function will generate a wait for the number of us increments.

Return Value

- **Fapi_Status_Success** (success)

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()

Services the watchdog timer

Synopsis

```
Fapi_StatusType Fapi_serviceWatchdogTimer(void)
```

Parameters

None

Description

This function allows the user to service their watchdog timer.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include “F021.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 EEPROM bank for erase and programming

Restrictions

This function is not applicable for Concerto and F2837xD devices.

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.

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include “F021.h”
Fapi_StatusType Fapi_setupEepromSectorEnable(void)
{
   /* Value must be 0xFFFF to enable erase and programming of the EEPROM bank, 0 to disable */
   Fapi_GlobalInit.m_poFlashControlRegisters->Fbse.u32Register = 0xFFFF;
   /* Enables/disables sectors  0-31 for bank and sector erase */
   Fapi_GlobalInit.m_poFlashControlRegisters->FsmSector1.u32Register = 0;
   /* Enables/disables sectors 32-63 for bank and sector erase */
   Fapi_GlobalInit.m_poFlashControlRegisters->FsmSector2.u32Register = 0;

   return(Fapi_Status_Success);
}
```

3.8.3 Fapi_setupBankSectorEnable()

Sets up the sectors available on non EEPROM banks for erase and programming

Synopsis

```
Fapi_StatusType Fapi_setupBankSectorEnable(void)
```

Parameters

None

Description

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

Return Value

- **Fapi_Status_Success** (success)

Sample Implementation

```
#include “F021.h”
Fapi_StatusType Fapi_setupEepromSectorEnable(void)
{
   /* Enables/disables sectors 0-15 for bank and sector erase */
 Fapi_GlobalInit.m_poFlashControlRegisters
        ->FsmSector.FSM_SECTOR_BITS.SECT_ERASED = 0;
   /* Enable sectors 0-15 for erase and programming */
   Fapi_GlobalInit.m_poFlashControlRegisters->Fbse.u32Register = 0xFFFF;
```

4 API Macros

The API includes a set of helper macros that may be used by the developer.

4.1 *FAPI_WRITE_LOCKED_FSM_REGISTER*

Allow easy writing to Flash Memory Controller registers that need to be unlocked first.

Synopsis

```
#define FAPI_WRITE_LOCKED_FSM_REGISTER(mRegister,mValue)                               \
    {                                                                                     \
        Fapi_GlobalInit.m_poFlashControlRegisters->FsmWrEna.FSM_WR_ENA_BITS.WR_ENA    = 0x5U; \
        mRegister = mValue;                                                                 \
        Fapi_GlobalInit.m_poFlashControlRegisters->FsmWrEna.FSM_WR_ENA_BITS.WR_ENA    = 0x2U; \
 }
```

Parameters

mRegister [in]	Address or bitfield of the locked register to be written to
mValue [in]	Value to be written to the locked register

Description

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

Return Value

None

5 Recommended FSM Flows

5.1 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.

5.2 Recommended Erase Flow

The following diagram describes the flow for erasing a sector(s) or bank(s) on a device. Please refer to Section 3.3.1 for further information.

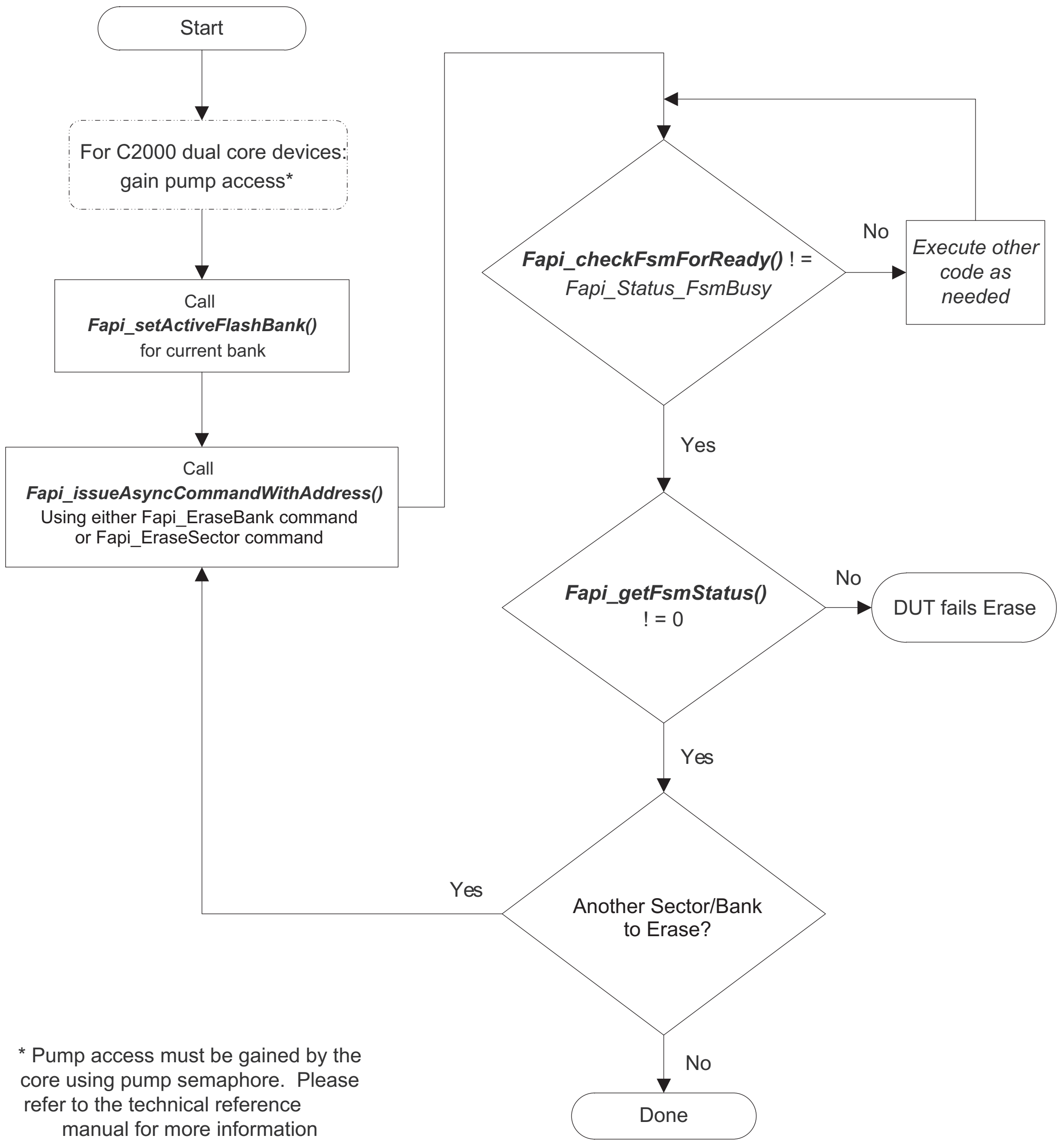
Start
For C2000 dual core devices: gain pump access*
Call *Fapi_setActiveFlashBank()* for current bank
Call *Fapi_issueAsyncCommandWithAddress()* Using either Fapi_EraseBank command or Fapi_EraseSector command
Fapi_checkFsmForReady() ! = *Fapi_Status_FsmBusy*
No
Execute other code as needed
Yes
Fapi_getFsmStatus() ! = 0
No
DUT fails Erase
Yes
Another Sector/Bank to Erase?
Yes
No
Done
* Pump access must be gained by the core using pump semaphore. Please refer to the technical reference manual for more information

Figure 1. Recommended Erase Flow

5.3 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 (see Section 5.2). Please refer to Section 3.4.1.1 for further information.

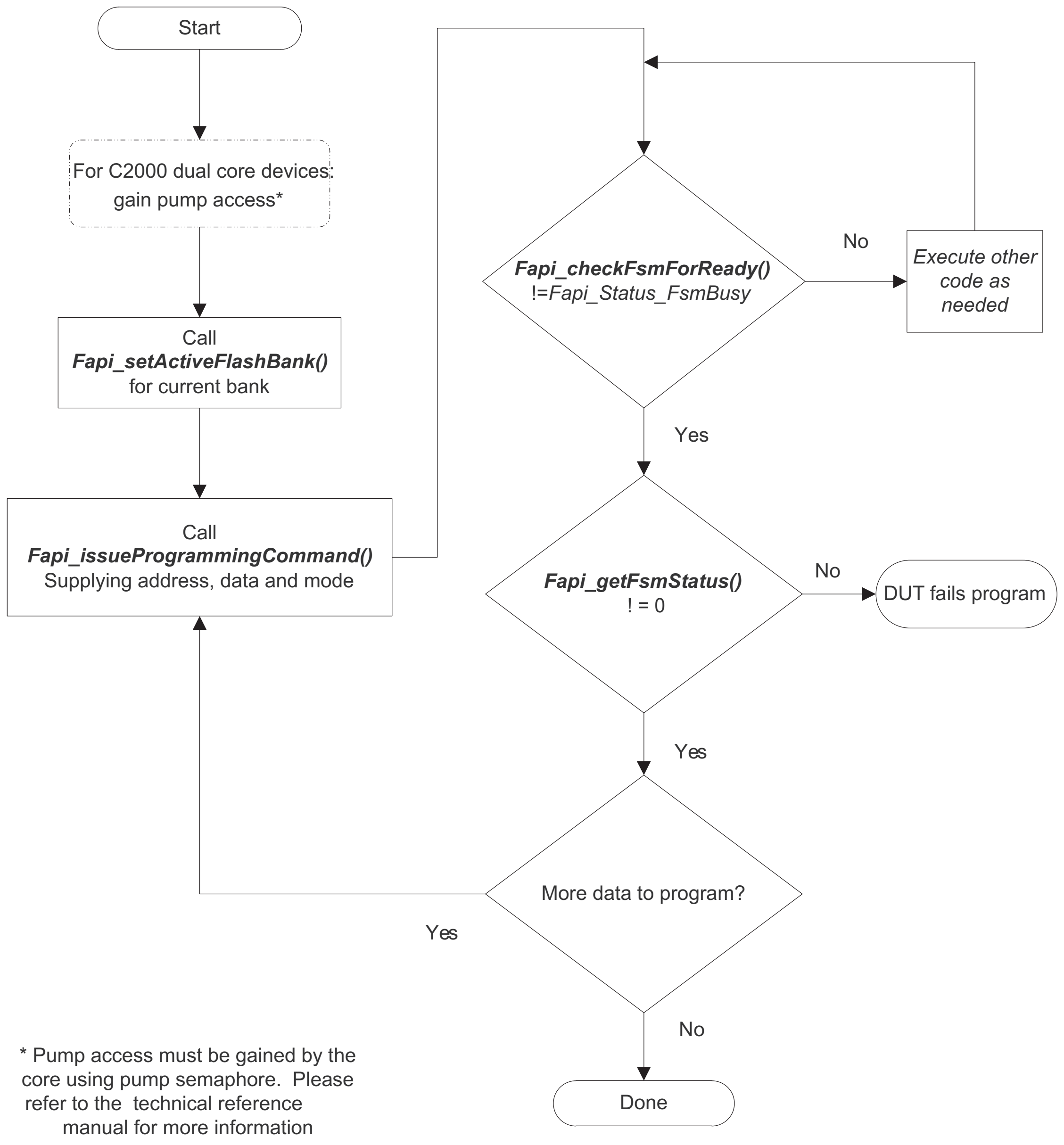
Start
For C2000 dual core devices: gain pump access*
Call *Fapi_setActiveFlashBank()* for current bank
Call *Fapi_issueProgrammingCommand()* Supplying address, data and mode
Fapi_checkFsmForReady() !=*Fapi_Status_FsmBusy*
No
Execute other code as needed
Yes
Fapi_getFsmStatus() ! = 0
No
DUT fails program
Yes
More data to program?
Yes
No
Done
* Pump access must be gained by the core using pump semaphore. Please refer to the technical reference manual for more information

Figure 2. Recommended Program Flow

Appendix A 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()
			Fapi_issueProgrammingCommandForEccAddress()

Erase Sector	Used to erase a Flash sector located by the specified address	Fapi_EraseSector	Fapi_issueAsyncCommandWithAddress()
Erase Bank[1]	Used to erase a Flash bank located by the specified address	Fapi_EraseBank	
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()

[1] Not supported on Concerto devices

Appendix B Object Library Function Information

B.1 ARM CortexR4 Big Endian Library

Table 12. ARM CortexR4 Big Endian Function Sizes and Stack Usage

Function Name	Size In Bytes	Worst Case Stack Usage
Fapi_calculateEcc	20	0
Fapi_calculateFletcherChecksum	42	8
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc • Fapi_serviceWatchdogTimer	316	64
Fapi_checkFsmForReady	18	0
Fapi_connectFlashPumpToCpu	4	0
Fapi_disableBanksForOtpWrite	32	0
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	432	80
Fapi_doBlankCheckByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	254	64
Fapi_doMarginRead *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	386	48
Fapi_doMarginReadByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	270	48
Fapi_doPsaVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	428	64
Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	586	104

Table 12. ARM CortexR4 Big Endian Function Sizes and Stack Usage (continued)

Fapi_doVerifyByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	424	96
Fapi_enableBanksForOtpWrite	38	0
Fapi_flushPipeline *Includes references to the following functions* • Fapi_waitDelay	78	16
Fapi_enableBanksForOtpWrite	38	0
Fapi_getBankSectors	154	16
Fapi_getDeviceInfo	78	32
Fapi_getFsmStatus	8	0
Fapi_getLibraryInfo	60	24
Fapi_initializeAPI	72	0
Fapi_isAddressEcc	62	0
Fapi_issueAsyncCommand	90	8
Fapi_issueAsyncCommandWithAddress *Includes references to the following functions* • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	170	24
Fapi_issueFsmSuspendCommand	56	0
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	450	64
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable • Fapi_remapEccAddress	568	88
Fapi_remapEccAddress	90	0
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	900	144
Fapi_waitDelay	44	8
Fapi_writeEwaitValue	54	0
Fapi_serviceWatchdogTimer(1)	?	?
Fapi_setupBankSectorEnable(1)	?	?
Fapi_setupEepromSectorEnable(1)	?	?

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

B.2 ARM CortexR4 Little Endian Library

Table 13. ARM CortexR4 Little Endian Function Sizes and Stack Usage

Function Name	Size In Bytes	Worst Case Stack Usage
Fapi_calculateEcc	20	0

Table 13. ARM CortexR4 Little Endian Function Sizes and Stack Usage (continued)

Fapi_calculateFletcherChecksum	42	16
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc • Fapi_serviceWatchdogTimer	316	64
Fapi_checkFsmForReady	18	0
Fapi_connectFlashPumpToCpu	4	0
Fapi_disableBanksForOtpWrite	32	0
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	432	80
Fapi_doBlankCheckByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	254	64
Fapi_doMarginRead *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	392	48
Fapi_doMarginReadByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	270	48
Fapi_doPsaVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	428	64
Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	586	104
Fapi_doVerifyByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	424	96
Fapi_enableBanksForOtpWrite	38	0
Fapi_flushPipeline *Includes references to the following functions* • Fapi_waitDelay	78	16
Fapi_enableBanksForOtpWrite	38	0
Fapi_getBankSectors	154	16
Fapi_getDeviceInfo	78	32
Fapi_getFsmStatus	8	0

Table 13. ARM CortexR4 Little Endian Function Sizes and Stack Usage (continued)

Fapi_getLibraryInfo	60	24
Fapi_initializeAPI	72	0
Fapi_isAddressEcc	62	0
Fapi_issueAsyncCommand	90	8
Fapi_issueAsyncCommandWithAddress *Includes references to the following functions* • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	170	24
Fapi_issueFsmSuspendCommand	56	0
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	448	64
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable • Fapi_remapEccAddress	566	88
Fapi_remapEccAddress	90	0
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	916	144
Fapi_waitDelay	44	8
Fapi_writeEwaitValue	54	0
Fapi_serviceWatchdogTimer(1)	?	?
Fapi_setupBankSectorEnable(1)	?	?
Fapi_setupEepromSectorEnable(2)	?	?

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

B.3 ARM CortexM3 Little Endian Library

Table 14. ARM CortexM3 Little Endian Function Sizes and Stack Usage

Function Name	Size In Bytes	Worst Case Stack Usage
Fapi_calculateEcc	20	0
Fapi_calculateFletcherChecksum	52	16
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc • Fapi_serviceWatchdogTimer	284	64
Fapi_checkFsmForReady	18	0
Fapi_connectFlashPumpToCpu	4	0
Fapi_disableBanksForOtpWrite	32	0

Table 14. ARM CortexM3 Little Endian Function Sizes and Stack Usage (continued)

Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	402	80
Fapi_doBlankCheckByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	252	64
Fapi_doMarginRead *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	352	48
Fapi_doMarginReadByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	268	48
Fapi_doPsaVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	390	64
Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	550	104
Fapi_doVerifyByByte *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay	392	96
Fapi_enableBanksForOtpWrite	32	0
Fapi_flushPipeline *Includes references to the following functions* • Fapi_waitDelay	78	16
Fapi_enableBanksForOtpWrite	38	0
Fapi_getBankSectors	138	16
Fapi_getDeviceInfo	78	32
Fapi_getFsmStatus	8	0
Fapi_getLibraryInfo	60	24
Fapi_initializeAPI	72	0
Fapi_isAddressEcc	58	0
Fapi_issueAsyncCommand	90	8
Fapi_issueAsyncCommandWithAddress *Includes references to the following functions* • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	172	24

Table 14. ARM CortexM3 Little Endian Function Sizes and Stack Usage (continued)

Fapi_issueFsmSuspendCommand	56	0
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	680	64
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable • Fapi_remapEccAddress	754	88
Fapi_remapEccAddress	46	0
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	918	144
Fapi_waitDelay	44	8
Fapi_writeEwaitValue	54	0
Fapi_serviceWatchdogTimer(1)	?	?
Fapi_setupBankSectorEnable(1)	?	?
Fapi_setupEepromSectorEnable(1)	?	?

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

B.4 C28x Library

Table 15. C28x Function Sizes and Stack Usage

Function Name	Size In Words	Worst Case Stack Usage
Fapi_calculateEcc	19	0
Fapi_calculateFletcherChecksum	38	2
Fapi_calculatePsa *Includes references to the following functions* • Fapi_isAddressEcc • Fapi_serviceWatchdogTimer	185	26
Fapi_checkFsmForReady	10	2
Fapi_connectFlashPumpToCpu	2	2
Fapi_disableBanksForOtpWrite	2	16
Fapi_doBlankCheck *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	145	32
Fapi_doMarginRead *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	201	24

Table 15. C28x Function Sizes and Stack Usage (continued)

Fapi_doVerify *Includes references to the following functions* • Fapi_flushPipeline • Fapi_serviceWatchdogTimer • Fapi_waitDelay • Fapi_isAddressEcc	263	42
Fapi_enableBanksForOtpWrite	19	4
Fapi_flushPipeline *Includes references to the following functions* • Fapi_waitDelay	45	8
Fapi_enableBanksForOtpWrite	38	0
Fapi_getBankSectors	151	10
Fapi_getDeviceInfo	37	14
Fapi_getFsmStatus	6	2
Fapi_getLibraryInfo	29	14
Fapi_initializeAPI	35	2
Fapi_isAddressEcc	25	2
Fapi_issueAsyncCommand	41	6
Fapi_issueAsyncCommandWithAddress *Includes references to the following functions* • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	96	8
Fapi_issueFsmSuspendCommand	23	2
Fapi_issueProgrammingCommand *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable	442	30
Fapi_issueProgrammingCommandForEccAddresses *Includes references to the following functions* • Fapi_calculateEcc • Fapi_setupBankSectorEnable • Fapi_setupEepromSectorEnable • Fapi_remapEccAddress	497	40
Fapi_remapEccAddress	35	2
Fapi_setActiveFlashBank *Includes references to the following functions* • Fapi_calculateFletcherChecksum	594	118
Fapi_waitDelay	23	4
Fapi_writeEwaitValue	24	2
Fapi_serviceWatchdogTimer(1)	?	?
Fapi_setupBankSectorEnable(1)	?	?
Fapi_setupEepromSectorEnable(2)	?	?

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

Appendix C Typedefs, defines, enumerations and structures

C.1 Type Definitions

```
#if defined(__TMS320C28XX__)

typedef unsigned char       boolean;

typedef unsigned int           uint8; //This is 16bits 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

```
#define false             FALSE;
#define true                 TRUE;

#if defined(__ICCARM__)            /* IAR EWARM Compiler */
#define ATTRIBUTE_PACKED  __packed
#elif defined(__TMS320C28XX__)     /* TI CGT C28xx compilers */
#define ATTRIBUTE_PACKED
#else                              /* all other compilers */
#define ATTRIBUTE_PACKED    __attribute__((packed))
#endif
```

C.3 Enumerations

C.3.1 Fapi_CpuSelectorType

This is used to indicate which CPU is being used.

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

C.3.2 Fapi_CpuType

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

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

C.3.3 Fapi_FamilyType

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

```
typedef enum
{
   Family_FMC       = 0x00,
   Family_L2FMC     = 0x10,
   Family_Sonata    = 0x20,
   Family_Stellaris = 0x30,
   Family_Future    = 0x40
} ATTRIBUTE_PACKED Fapi_FamilyType;
```

C.3.4 Fapi_AddressMemoryType

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

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

C.3.5 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.3.6 Fapi_FlashBankType

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

```
typedef enum
{
   Fapi_FlashBank0,
   Fapi_FlashBank1,
   Fapi_FlashBank2,
   Fapi_FlashBank3,
   Fapi_FlashBank4,
   Fapi_FlashBank5,
   Fapi_FlashBank6,
   Fapi_FlashBank7
}  ATTRIBUTE_PACKED Fapi_FlashBankType;
```

C.3.7 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.3.8 Fapi_FlashSectorType

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

```
typedef enum
{
   Fapi_FlashSector0,
   Fapi_FlashSector1,
   Fapi_FlashSector2,
   Fapi_FlashSector3,
   Fapi_FlashSector4,
   Fapi_FlashSector5,
   Fapi_FlashSector6,
   Fapi_FlashSector7,
   Fapi_FlashSector8,
   Fapi_FlashSector9,
   Fapi_FlashSector10,
   Fapi_FlashSector11,
   Fapi_FlashSector12,
   Fapi_FlashSector13,
   Fapi_FlashSector14,
   Fapi_FlashSector15,
   Fapi_FlashSector16,
   Fapi_FlashSector17,
   Fapi_FlashSector18,
   Fapi_FlashSector19,
   Fapi_FlashSector20,
   Fapi_FlashSector21,
   Fapi_FlashSector22,
   Fapi_FlashSector23,
   Fapi_FlashSector24,
   Fapi_FlashSector25,
   Fapi_FlashSector26,
   Fapi_FlashSector27,
   Fapi_FlashSector28,
   Fapi_FlashSector29,
   Fapi_FlashSector30,
   Fapi_FlashSector31,
   Fapi_FlashSector32,
   Fapi_FlashSector33,
   Fapi_FlashSector34,
   Fapi_FlashSector35,
   Fapi_FlashSector36,
   Fapi_FlashSector37,
   Fapi_FlashSector38,
   Fapi_FlashSector39,
   Fapi_FlashSector40,
   Fapi_FlashSector41,
   Fapi_FlashSector42,
   Fapi_FlashSector43,
   Fapi_FlashSector44,
   Fapi_FlashSector45,
   Fapi_FlashSector46,
   Fapi_FlashSector47,
   Fapi_FlashSector48,
   Fapi_FlashSector49,
   Fapi_FlashSector50,
   Fapi_FlashSector51,
   Fapi_FlashSector52,
   Fapi_FlashSector53,
   Fapi_FlashSector54,
   Fapi_FlashSector55,
   Fapi_FlashSector56,
   Fapi_FlashSector57,
   Fapi_FlashSector58,
   Fapi_FlashSector59,
   Fapi_FlashSector60,
   Fapi_FlashSector61,
   Fapi_FlashSector62,
   Fapi_FlashSector63
}  ATTRIBUTE_PACKED Fapi_FlashSectorType;
```

C.3.9 Fapi_FlashStateCommandsType

This contains all the possible Flash State Machine commands.

```
typedef enum
{
   Fapi_ProgramData    = 0x0002,
   Fapi_EraseSector    = 0x0006,
   Fapi_EraseBank      = 0x0008,
   Fapi_ValidateSector = 0x000E,
   Fapi_ClearStatus    = 0x0010,
   Fapi_ProgramResume  = 0x0014,
   Fapi_EraseResume    = 0x0016,
   Fapi_ClearMore      = 0x0018
}  ATTRIBUTE_PACKED Fapi_FlashStateCommandsType;
```

C.3.10 Fapi_FlashReadMarginModeType

This contains all the possible Flash State Machine commands.

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

C.3.11 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.3.12 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_TiOtpBytesType

This is used to define an accessor to the TI OTP values

```
#if defined(_LITTLE_ENDIAN)
typedef volatile union
{
   struct
   {
#if defined (_C28X)
      uint16 ChecksumLength:16;  /* 0x150 bits 15:0 */
      uint16 OtpVersion:16;      /* 0x150 bits 31:16 */
      uint32 OtpChecksum;        /* 0x154 bits 31:0 */
      uint16 NumberOfBanks:16;   /* 0x158 bits 15:0 */
      uint16 NumberOfSectors:16; /* 0x158 bits 31:16 */
      uint16 MemorySize:16;      /* 0x15C bits 15:0 */
      uint16 Package:16;         /* 0x15C bits 31:16 */
      uint16 SiliconRevision:8;  /* 0x160 bits 7:0 */
      uint16 AsicNumber_23_8:8; /* 0x160 bits 31:8 */
      uint16 AsicNumber_31_24:16; /* 0x160 bits 31:8 */
      uint32 LotNumber;          /* 0x164 bits 31:0 */
      uint16 WaferNumber:16;     /* 0x168 bits 15:0 */
      uint16 Flowbits:16;        /* 0x168 bits 31:16 */
      uint16 YCoordinate:16;     /* 0x16C bits 15:0 */
      uint16 XCoordinate:16;     /* 0x16C bits 31:16 */
      uint16 EVSU:8;             /* 0x170 bits 7:0 */
      uint16 PVSU:8;             /* 0x170 bits 15:8 */
      uint16 ESU:8;              /* 0x170 bits 23:16 */
      uint16 PSU:8;              /* 0x170 bits 31:24 */
      uint16 CVSU:12;            /* 0x174 bits 11:0 */
      uint16 Add_EXEZSU:4;       /* 0x174 bits 15:12 */
      uint16 PVAcc:8;            /* 0x174 bits 23:16 */
      uint16 RVSU:8;             /* 0x174 bits 31:24 */
      uint16 PVH2:8;             /* 0x178 bits 7:0 */
      uint16 PVH:8;              /* 0x178 bits 15:8 */
      uint16 RH:8;               /* 0x178 bits 23:16 */
      uint16 PH:8;               /* 0x178 bits 31:24 */
      uint16 SmFrequency:12;     /* 0x17C bits 11:0 */
      uint16 VSTAT:4;            /* 0x17C bits 15:12 */
      uint16 Sequence:8;         /* 0x17C bits 23:16 */
      uint16 EH:8;               /* 0x17C bits 31:24 */
      uint16 VHV_EStep:16;       /* 0x180 bits 15:0 */
      uint16 VHV_EStart:16;      /* 0x180 bits 31:16 */
      uint16 MAX_PP:16;          /* 0x184 bits 15:0 */
      uint16 OtpReserved1:16;    /* 0x184 bits 31:16 */
      uint16 PROG_PW:16;         /* 0x188 bits 15:0 */
      uint16 MAX_EP:16;          /* 0x188 bits 31:16 */
      uint32 ERA_PW;             /* 0x18C bits 31:0 */
      uint16 VHV_E:16;           /* 0x190 bits 15:0 */
      uint16 VHV_P:16;           /* 0x190 bits 31:16 */
      uint16 VINH:8;             /* 0x194 bits 7:0 */
      uint16 VCG:8;              /* 0x194 bits 15:8 */
      uint16 VHV_PV:16;          /* 0x194 bits 31:16 */
      uint16 OtpReserved2:8;     /* 0x198 bits 7:0 */
      uint16 VRead:8;            /* 0x198 bits 15:8 */
      uint16 VWL_P:8;            /* 0x198 bits 23:16 */
      uint16 VSL_P:8;            /* 0x198 bits 31:24 */
      uint32 ApiChecksum;        /* 0x19C bits 15:0 */
      uint32 OtpReserved3;    /* 0x1A0 bits 31:0 */
      uint32 OtpReserved4;    /* 0x1A4 bits 31:0 */
      uint32 OtpReserved5;    /* 0x1A8 bits 31:0 */
      uint32 OtpReserved6;    /* 0x1AC bits 31:0 */
#else
      uint32 ChecksumLength:16;  /* 0x150 bits 15:0 */
      uint32 OtpVersion:16;      /* 0x150 bits 31:16 */
      uint32 OtpChecksum;        /* 0x154 bits 31:0 */
      uint32 NumberOfBanks:16;   /* 0x158 bits 15:0 */
```

```
            uint32 NumberOfSectors:16; /* 0x158 bits 31:16 */
            uint32 MemorySize:16;      /* 0x15C bits 15:0 */
            uint32 Package:16;         /* 0x15C bits 31:16 */
            uint32 SiliconRevision:8;  /* 0x160 bits 7:0 */
            uint32 AsicNumber:24;       /* 0x160 bits 31:8 */
            uint32 LotNumber;          /* 0x164 bits 31:0 */
            uint32 WaferNumber:16;     /* 0x168 bits 15:0 */
            uint32 Flowbits:16;        /* 0x168 bits 31:16 */
            uint32 YCoordinate:16;     /* 0x16C bits 15:0 */
            uint32 XCoordinate:16;     /* 0x16C bits 31:16 */
            uint32 EVSU:8;             /* 0x170 bits 7:0 */
            uint32 PVSU:8;             /* 0x170 bits 15:8 */
            uint32 ESU:8;              /* 0x170 bits 23:16 */
            uint32 PSU:8;              /* 0x170 bits 31:24 */
            uint32 CVSU:12;            /* 0x174 bits 11:0 */
            uint32 Add_EXEZSU:4;       /* 0x174 bits 15:12 */
            uint32 PVAcc:8;            /* 0x174 bits 23:16 */
            uint32 RVSU:8;             /* 0x174 bits 31:24 */
            uint32 PVH2:8;             /* 0x178 bits 7:0 */
            uint32 PVH:8;              /* 0x178 bits 15:8 */
            uint32 RH:8;               /* 0x178 bits 23:16 */
            uint32 PH:8;               /* 0x178 bits 31:24 */
            uint32 SmFrequency:12;     /* 0x17C bits 11:0 */
            uint32 VSTAT:4;            /* 0x17C bits 15:12 */
            uint32 Sequence:8;         /* 0x17C bits 23:16 */
            uint32 EH:8;               /* 0x17C bits 31:24 */
            uint32 VHV_EStep:16;       /* 0x180 bits 15:0 */
            uint32 VHV_EStart:16;      /* 0x180 bits 31:16 */
            uint32 MAX_PP:16;          /* 0x184 bits 15:0 */
            uint32 OtpReserved1:16;    /* 0x184 bits 31:16 */
            uint32 PROG_PW:16;         /* 0x188 bits 15:0 */
            uint32 MAX_EP:16;          /* 0x188 bits 31:16 */
            uint32 ERA_PW;             /* 0x18C bits 31:0 */
            uint32 VHV_E:16;           /* 0x190 bits 15:0 */
            uint32 VHV_P:16;           /* 0x190 bits 31:16 */
            uint32 VINH:8;             /* 0x194 bits 7:0 */
            uint32 VCG:8;              /* 0x194 bits 15:8 */
            uint32 VHV_PV:16;          /* 0x194 bits 31:16 */
            uint32 OtpReserved2:8;     /* 0x198 bits 7:0 */
            uint32 VRead:8;            /* 0x198 bits 15:8 */
            uint32 VWL_P:8;            /* 0x198 bits 23:16 */
            uint32 VSL_P:8;            /* 0x198 bits 31:24 */
            uint32 ApiChecksum:32;     /* 0x19C bits 31:0 */
            uint32 OtpReserved3:32;    /* 0x1A0 bits 31:0 */
            uint32 OtpReserved4:32;    /* 0x1A4 bits 31:0 */
            uint32 OtpReserved5:32;    /* 0x1A8 bits 31:0 */
            uint32 OtpReserved6:32;    /* 0x1AC bits 31:0 */
      #endif
         } OTP_VALUE;
         uint8  au8OtpWord[0x60];
         uint16 au16OtpWord[0x30];
         uint32 au32OtpWord[0x18];
      }Fapi_TiOtpBytesType;
      #else
      typedef volatile union
      {
         struct
         {
            uint32 OtpVersion:16;      /* 0x150 bits 31:16 */
            uint32 ChecksumLength:16;  /* 0x150 bits 15:0 */
            uint32 OtpChecksum;        /* 0x154 bits 31:0 */
            uint32 NumberOfSectors:16; /* 0x158 bits 31:16 */
            uint32 NumberOfBanks:16;   /* 0x158 bits 15:0 */
            uint32 Package:16;         /* 0x15C bits 31:16 */
            uint32 MemorySize:16;      /* 0x15C bits 15:0 */
            uint32 AsicNumber:24;      /* 0x160 bits 31:8 */
            uint32 SiliconRevision:8;  /* 0x160 bits 7:0 */
            uint32 LotNumber;          /* 0x164 bits 31:0 */
            uint32 Flowbits:16;        /* 0x168 bits 31:16 */
            uint32 WaferNumber:16;     /* 0x168 bits 15:0 */
            uint32 XCoordinate:16;     /* 0x16C bits 31:16 */
            uint32 YCoordinate:16;     /* 0x16C bits 15:0 */
            uint32 PSU:8;              /* 0x170 bits 31:24 */
            uint32 ESU:8;              /* 0x170 bits 23:16 */
            uint32 PVSU:8;             /* 0x170 bits 15:8 */
            uint32 EVSU:8;             /* 0x170 bits 7:0 */
            uint32 RVSU:8;             /* 0x174 bits 31:24 */
            uint32 PVAcc:8;            /* 0x174 bits 23:16 */
            uint32 Add_EXEZSU:4;       /* 0x174 bits 15:12 */
            uint32 CVSU:12;            /* 0x174 bits 11:0 */
            uint32 PH:8;               /* 0x178 bits 31:24 */
            uint32 RH:8;               /* 0x178 bits 23:16 */
            uint32 PVH:8;              /* 0x178 bits 15:8 */
            uint32 PVH2:8;             /* 0x178 bits 7:0 */
            uint32 EH:8;               /* 0x17C bits 31:24 */
            uint32 Sequence:8;         /* 0x17C bits 23:16 */
            uint32 VSTAT:4;            /* 0x17C bits 15:12 */
            uint32 SmFrequency:12;     /* 0x17C bits 11:0 */
            uint32 VHV_EStart:16;      /* 0x180 bits 31:16 */
            uint32 VHV_EStep:16;       /* 0x180 bits 15:0 */
            uint32 OtpReserved1:16;    /* 0x184 bits 31:16 */
            uint32 MAX_PP:16;          /* 0x184 bits 15:0 */
            uint32 MAX_EP:16;          /* 0x188 bits 31:16 */
            uint32 PROG_PW:16;         /* 0x188 bits 15:0 */
            uint32 ERA_PW;             /* 0x18C bits 31:0 */
            uint32 VHV_P:16;           /* 0x190 bits 31:16 */
            uint32 VHV_E:16;           /* 0x190 bits 15:0 */
            uint32 VHV_PV:16;          /* 0x194 bits 31:16 */
            uint32 VCG:8;              /* 0x194 bits 15:8 */
            uint32 VINH:8;             /* 0x194 bits 7:0 */
            uint32 VSL_P:8;            /* 0x198 bits 31:24 */
            uint32 VWL_P:8;            /* 0x198 bits 23:16 */
            uint32 VRead:8;            /* 0x198 bits 15:8 */
            uint32 OtpReserved2:8;     /* 0x198 bits 7:0 */
            uint32 ApiChecksum:32;     /* 0x19C bits 31:0 */
            uint32 OtpReserved3:32;    /* 0x1A0 bits 31:0 */
            uint32 OtpReserved4:32;    /* 0x1A4 bits 31:0 */
            uint32 OtpReserved5:32;    /* 0x1A8 bits 31:0 */
            uint32 OtpReserved6:32;    /* 0x1AC bits 31:0 */
         } OTP_VALUE;
         uint8  au8OtpWord[0x60];
         uint16 au16OtpWord[0x30];
         uint32 au32OtpWord[0x18];
      }Fapi_TiOtpBytesType;
      #endif
```

C.4.4 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.4.5 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.4.6 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;
```

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		