F021 Flash API

Version 2.01.01

Reference Guide

Literature Number: SPNU501H
December 2012–Revised April 2015

Contents

List of Figures

List of Tables

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 Introduction

1.1 Reference Material

Use this guide in conjunction with the *F021 Flash Module* chapter in the device-specific technical reference manual and data sheet that is being used. For additional options for programming and erasing the Flash, see the *Advanced F021 Flash API Erase/Program Usage* (SPNA148).

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 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 (65nm) process. On ARM Cortex devices, these routines must be run in a *privileged mode* (a mode other than user) to allow access to the Flash memory controller registers. The API verifies for the selected bank, that the appropriate RWAIT or EWAIT value is set for the specified system frequency.

2.2 API Overview

Table 1. Summary of Flash State Machine Functions

API Function	Description
Fapi_disableAutoEccCalculation()[(1)]	Disables auto generation of ECC when data is written into an FWPWRITEx register.
Fapi_disableBanksForOtpWrite()	Disables all banks from programming customer OTP
Fapi_disableFsmDoneEvent()	Disables the generation of an FSM_Done event at the end of a program or erase operation.
Fapi_enableAutoEccCalculation()[(1)]	Enables auto generation of ECC when data is written into an FWPWRITEx register.
Fapi_enableBanksForOtpWrite()	Enables banks to allow programming of customer OTP
Fapi_enableEepromBankSectors()	Enables the sectors in EEPROM bank for program and erase operations
Fapi_enableFsmDoneEvent()	Enables the generation of an FSM_Done event at the end of a program or erase operation.
Fapi_enableMainBankSectors()	Enables the sectors in Main banks for program and erase operations
Fapi_initializeFlashBanks()	Required Bank initialization before any erase, program, or verify API function.
Fapi_isAddressEcc()	Determines if address falls in Flash memory controller ECC ranges
Fapi_remapEccAddress()	Remaps an ECC address to corresponding main address
Fapi_remapMainAddress()	Remaps an Main address to corresponding ECC address
Fapi_setActiveFlashBank()	Sets the active bank for a erase or program command

(1) This function is only available on devices with the L2FMC Flash Controller.

Table 2. Summary of Asynchronous Command Functions

API Function	Description
Fapi_issueAsyncCommand()	Issues a command to FSM for operations that do not require an address
Fapi_issueAsyncCommandWithAddress()	Issues a command to FSM for operations that require an address

Table 3. Summary of Program Functions

API Function	Description
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()

Table 4. Summary of Read Functions

API Function	Description
Fapi_doVerify()	Verifies specified Flash memory range against supplied values
Fapi_doVerifyByByte()	Verifies specified Flash memory range against supplied values by byte
Fapi_doBlankCheck()[1]	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()	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
Fapi_flushPipeline()	Flushes the pipeline buffers in the Flash memory controller

(1) For devices with L2FMC memory controller, these functions have to be executed from RAM.

Table 5. 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()	Returns the sector information for a bank

Table 6. Summary of User Defined Functions

API Function	Description
Fapi_serviceWatchdogTimer()	User modifiable function to service watchdog timer

Table 7. Summary of Utility Functions

API Function	Description
Fapi_calculateFletcherChecksum()	Function calculates a Fletcher checksum for the memory range specified
Fapi_calculateEcc()	Calculates the ECC for the supplied address and 64-bit word
Fapi_waitDelay()[1]	Creates a delay

(1) This function is deprecated and should not be used in new projects

2.3 Using API

This section describes how to use the various API functions and any relevant flows.

2.3.1 Initialization Flow

For proper initialization of the device prior to any Flash operations, see the device-specific initialization document. Additionally, *all API functions require execution in privilege mode*.

2.3.1.1 Before Using Any Erase, Program or Read Flash API Function

Before using any *asynchronous command* Table 2 , *program* Table 3 or *read* Table 4 functions, the function *Fapi_initializeFlashBanks()* must be called to correctly initialize the Flash Memory controller.

2.3.1.2 Bank Setup

Before performing a Flash erase or program operation for the first time or on a different Bank than is the current active Bank, the function *Fapi_setActiveFlashBank()* must be called. Additionally, *Fapi_enableMainBankSectors()* (for banks 0-6) or *Fapi_enableEepromBankSectors()* (for bank 7) must be called before the first sector erase or program operation and always before a bank erase operation.

2.3.1.3 On System Frequency Change

If the System operating frequency (HCLK) is changed after the initial call to *Fapi_initializeFlashBanks()*, this function along with *Fapi_setActiveFlashBank()* must be called again before any *asynchronous command* Table 2 , *program* Table 3 or *read* Table 4 function. This will update the Flash Memory controller to the new system frequency.

2.3.2 Flash Addressing

For program and erase operations on Bank 0 to Bank 6, the API and the FMC requires the standard Flash memory map where Bank 0, Sector 0 begins at address 0. For the read functions on the Flash memory in Bank 0 to Bank 6, they need either current address mapping (Flash addresses starting at either 0 {Power On State} or 0x0800_0000 {RAM-Flash swap) or mirrored addresses starting at 0x2000_0000.

2.3.3 Building With the API

The macro *_L2FMC* must be defined before the inclusion of the header file F021.h on devices with the L2FMC Flash controller.

```
#define _L2FMC
```

2.3.3.1 Object Library Files

All ARM Cortex Flash API object files are distributed in the ARM standard EABI ELF object format. For the CortexR4/R5 cores, the library files are built using Thumb2 mode.

2.3.3.2 Distribution Files

The following API files are distributed with the installer:

- Library Files - *(All library files were built using TI's code generation tools for ARM v5.1.3 with the following compile options: -mv7R4 --abi=eabi --strict_ansi -O3 --diag_warning=225 --gen_func_subsections=on --enum_type=packed --code_state=16)*
  - F021_API_CortexR4_BE.lib – This is the Flash API object file for Cortex R4 Big Endian devices. *(In addition to the general build options, this library was built using : -g --symdebug:dwarf_version=3)*
  - F021_API_CortexR4_BE_v3D16.lib – This is the Flash API object file for Cortex R4 Big Endian devices that are using floating point unit. *(In addition to the general build options, this library was built using : -g --symdebug:dwarf_version=3 --float_support=VFPv3D16)*
  - F021_API_CortexR4_BE_L2FMC.lib – This is the Flash API object file for Cortex R4 Big Endian devices using the L2FMC memory controller. *(In addition to the general build options, this library was built using : -g --symdebug:dwarf_version=3)*
  - F021_API_CortexR4_LE.lib – This is the Flash API object file for Cortex R4 Little Endian devices. *(In addition to the general build options, this library was built using : -me -g --symdebug:dwarf_version=3)*
  - F021_API_CortexR4_LE_v3D16.lib – This is the Flash API object file for Cortex R4 Little Endian devices that are using floating point unit. *(In addition to the general build options, this library was built using : -me -g --symdebug:dwarf_version=3 --float_support=VFPv3D16)*
  - F021_API_CortexR4_LE_L2FMC.lib – This is the Flash API object file for Cortex R4 Little Endian devices using the L2FMC memory controller. *(In addition to the general build options, this library was built using : -me -g --symdebug:dwarf_version=3)*
  - F021_API_CortexR4_BE_L2FMC_v3D16.lib – This is the Flash API object file for Cortex R4/R5 Big Endian devices using the L2FMC memory controller and float point unit. *(In addition to the general build options, this library was built using : -g --symdebug:dwarf_version=3 --float_support=VFPv3D16)*
  - F021_API_CortexR4_LE_L2FMCv3D16.lib – This is the Flash API object file for Cortex R4/R5 Little Endian devices using the L2FMC memory controller and float point unit. *(In addition to the general build options, this library was built using : -me -g --symdebug:dwarf_version=3 --float_support=VFPv3D16)*

  - F021_API_CortexR4_BE_NDS.lib – This is the Flash API object file without debug symbols for Cortex R4 Big Endian devices . *(In addition to the general build options, this library was built using : --symdebug:none)*
  - F021_API_CortexR4_BE_v3D16_NDS.lib – This is the Flash API object file without debug symbols for Cortex R4 Big Endian devices that are using floating point unit. *(In addition to the general build options, this library was built using : --symdebug:none --float_support=VFPv3D16)*
  - F021_API_CortexR4_BE_L2FMC_NDS.lib – This is the Flash API object file without debug symbols for Cortex R4 Big Endian devices using the L2FMC memory controller. *(In addition to the general build options, this library was built using : --symdebug:none)*
  - F021_API_CortexR4_LE_NDS.lib – This is the Flash API object file without debug symbols for Cortex R4 Little Endian devices. *(In addition to the general build options, this library was built using : -me --symdebug:none)*
  - F021_API_CortexR4_LE_v3D16_NDS.lib – This is the Flash API object file without debug symbols for Cortex R4 Little Endian devices that are using floating point unit. *(In addition to the general build options, this library was built using : -me --symdebug:none --float_support=VFPv3D16)*
  - F021_API_CortexR4_LE_L2FMC_NDS.lib – This is the Flash API object file without debug symbols for Cortex R4 Little Endian devices using the L2FMC memory controller. *(In addition to the general build options, this library was built using : -me --symdebug:none)*
  - F021_API_CortexR4_BE_L2FMC_v3D16_NDS.lib – This is the Flash API object without debug symbols file for Cortex R4/R5 Big Endian devices using the L2FMC memory controller and float point unit. *(In addition to the general build options, this library was built using : --symdebug:none --float_support=VFPv3D16)*
  - F021_API_CortexR4_LE_L2FMCv3D16_NDS.lib – This is the Flash API object file without debug symbols for Cortex R4/R5 Little Endian devices using the L2FMC memory controller and float point unit. *(In addition to the general build options, this library was built using : -me --symdebug:none --float_support=VFPv3D16)*
- Source Files
  - Fapi_UserDefinedFunctions.c – This is file that contains the user definable functions.
- Include Files
  - F021.h – This is the master include file and includes all other include files. This should be the only include file added to the users's code.
- The following include files should not be included directly by the user’s code, but are listed here for user reference:
  - Compatibility.h - A set of macros to be used for backwards compatibility for 1.x.x versions of the API.
  - Constants.h – Constant definitions used by the API.
  - FapiFunctions.h - Contains all the Fapi function prototypes.
  - Helpers.h – Set of helper defines
  - Registers.h – Definitions common to all register implementations and includes the appropriate register include file for the selected device type.
  - Registers_FMC_BE.h – Big Endian Flash memory controller registers structure for TMS570/RM4 devices.
  - Registers_FMC_LE.h – Little Endian Flash memory controller registers structure for TMS570/RM4 devices.
  - Types.h – Contains all the enumerations and structures used by the API
  - Below are a set of compiler specific support header files:
    - CGT.ARM.h - Contains a set of definitions used by the ARM compiler
    - CGT.CCS.h - Contains a set of definitions used by the TI CCS compiler
    - CGT.gcc.h - Contains a set of definitions used by the gcc compiler
    - CGT.GHS.h - Contains a set of definitions used by the GreenHills compiler
    - CGT.IAR.h - Contains a set of definitions used by the IAR EWARM compiler

- Library information files
  - build_information.txt - This file contains function callgraphs, worst case stack usage for each function, function size in bytes and MD5 and SHA1 checksums for all files delivered in the installer package.
  - License_Agreement.pdf - This file contains the libraries license agreement.
  - readme.txt - This file contains build specific information.
  - Release_Notes.pdf - This file contains release specific information.
  - spna148.pdf - This is the application note, Advanced F021 Flash API Erase/Program Usage.
  - spnu501x.pdf - This file.
  - spnz210.pdf - This is the library errata document.

2.3.4 Executing API From Flash

NOTE: 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.

2.3.5 Memory Regions Required to be Readable

NOTE: The F021 Flash API library must be able to read from addresses 0x000, 0x100, 0x200, 0x300, and the TI OTP region, 0xF008_0000 - 0xF00B_0000, for the API to operate correctly.

3 API Functions

3.1 Flash State Machine Functions

3.1.1 Fapi_disableAutoEccCalculation()

Disables auto generation of ECC on L2FMC devices

Synopsis

```
Fapi_StatusType Fapi_disableAutoEccCalculation(void)
```

Parameters

None

Description

NOTE: This function is only available on devices using the L2FMC Flash Memory Controller.

This function disables the auto generation of ECC when data is written to the FWPWRITEx registers on L2FMC devices. The use of this function is primarily intended for those users writing their own programming functions based on the application note, *Advanced F021 Flash API Erase/Program Usage* (SPNA148). The function *Fapi_issueProgrammingCommand()* Section 3.2.2 will automatically set this bit depending on the programming mode used (enabled for Fapi_AutoEccGeneration, disabled for all other modes) and will stay that way when this function returns.

Return Value

- *Fapi_Status_Success* (success)

3.1.2 Fapi_disableBanksForOtpWrite()

Disables ability to program the Customer OTP on all Flash Banks.

Synopsis

```
Fapi_StatusType Fapi_disableBanksForOtpWrite(void)
```

Parameters

None

Description

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

Return Value

- **Fapi_Status_Success** (success)

3.1.3 Fapi_disableFsmDoneEvent()

Disables generation of the FSM_DONE event.

Synopsis

```
Fapi_StatusType Fapi_disableFsmDoneEvent(void)
```

Parameters

None

Description

This function disables the generation of the FSM_DONE event. For more information on the FSM_DONE event, see the device-specific technical reference manual.

Return Value

- **Fapi_Status_Success** (success)

3.1.4 Fapi_enableAutoEccCalculation()

Enables auto generation of ECC on L2FMC devices

Synopsis

```
Fapi_StatusType Fapi_enableAutoEccCalculation(void)
```

Parameters

None

Description

NOTE: This function is only available on devices using the L2FMC Flash Memory Controller.

This function enables the auto generation of ECC when data is written to the FWPWRITEx registers on L2FMC devices. The use of this function is primarily intended for those users writing their own programming functions based on *Advanced F021 Flash API Erase/Program Usage* (SPNA148). The function *Fapi_issueProgrammingCommand()* Section 3.2.2 will automatically set this bit depending on the programming mode used (enabled for Fapi_AutoEccGeneration, disabled for all other modes) and will stay that way when this function returns.

Return Value

- **Fapi_Status_Success** (success)

3.1.5 Fapi_enableBanksForOtpWrite()

Enables ability to program the Customer OTP region for the specified banks

Synopsis

```
Fapi_StatusType Fapi_enableBanksForOtpWrite(
        uint8_t 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 the ability to program Customer OTP for the banks specified in the bitfield mask u8Banks. The bitfield mask has Bank 0 as bit 1 up to Bank 7 as bit 7.

Return Value

- **Fapi_Status_Success** (success)

3.1.6 Fapi_enableEepromBankSectors()

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

Synopsis

```
Fapi_StatusType Fapi_enableEepromBankSectors(
        uint32_t u32SectorsEnables_31_0,
        uint32_t u32SectorsEnables_63_32)
```

Parameters

u32SectorsEnables_31_0 [in]	Bit mask indicating which of sectors 0-31 are enabled for erase and programming.
u32SectorsEnables_63_32 [in]	Bit mask indicating which of sectors 32-63 are enabled for erase and programming.

Description

NOTE: On devices using the L2FMC Flash Memory Controller, this function should be called only when the Flash State Machine is Idle .

This function sets up the sectors in the EEPROM banks that are available for erase and programming operations. This function must be called with the EEPROM bank (Flash Bank 7) as the active bank. Additionally, the function must be called once before performing program and sector erase operations and always before a bank erase operation. Each bit refers to a single sector with Sector 0 is bit 0 in u32SectorEnables_31_0 to Sector 31 is bit 31 in u32SectorEnables_31_0 and Sector 32 is bit 0 in u32SectorEnables_63_32 to Sector 63 is bit 31 in u32SectorEnables_63_32. This function will check the OTP to see if the requested sector exists on the device and will enable it only if exists.

Return Value

- **Fapi_Status_Success** (success)

3.1.7 Fapi_enableFsmDoneEvent()

Enables the generation of the FSM_DONE event

Synopsis

```
Fapi_StatusType Fapi_enableBanksForOtpWrite(void)
```

Parameters

None

Description

This function enables the generation of the FSM_DONE event. This event is generated when the FSM finishes a program or erase operation. The FSM_DONE event flag resides in the FEDACSTATUS register at bit 24. For a complete description of the FEDACSTATUS register and FSM_DONE event, see the *F021 Flash Module* chapter in the device-specific technical reference manual. This FSM_DONE flag must be cleared by writing a one to this bit in order for the event signal to stop. There is a helper macro CLEAR_FSM_DONE_EVENT defined to accomplish this.

Return Value

- **Fapi_Status_Success** (success)

3.1.8 Fapi_enableMainBankSectors()

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

Synopsis

```
Fapi_StatusType Fapi_enableMainBankSectors(
        uint16_t u16SectorsEnables)
```

Parameters

u16SectorsEnables [in]	Bit mask indicating which of sectors 0-15 are enabled for erase and programming.

Description

NOTE: On devices using the L2FMC Flash Memory Controller, this function should be called only when the Flash State Machine is Idle .

This function sets up the sectors in the non-EEPROM banks that are available for erase and programming operations. This function must be called with the bank intended for the erase or program operation as the active bank. Additionally, the function must be called once before performing program and sector erase operations and always before a bank erase operation. Each bit refers to a single sector where Sector 0 is bit 0 to Sector 15 is bit 15 in u16SectorEnables.This function will check the OTP to see if the requested sector exists on the device and will enable it only if exists.

Return Value

- **Fapi_Status_Success** (success)

3.1.9 Fapi_isAddressEcc()

Indicates if an address is in the Flash Memory Controller ECC space

Synopsis

```
boolean_t Fapi_isAddressEcc(
          uint32_t u32Address)
```

Parameters

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

Description

This function returns True if the address in u32Address 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)

3.1.10 Fapi_initializeFlashBanks()

Initializes the Flash Banks for API operations

Synopsis

```
Fapi_StatusType Fapi_initializeFlashBanks(
         uint32_t u32HclkFrequency)
```

Parameters

u32HclkFrequency [in]	System clock frequency in MHz. The value should be rounded up to the next integer. For example, if the system clock frequency is 133.3MHz, then the u32HclkFrequency value should be 134.

Description

This function is required to initialize the Flash Banks before using any *asynchronous command* Table 2 , *program* Table 3 or *read* Table 4 functions. This function must also be called if the system frequency is changed and/or RWAIT/EWAIT values are changed.

NOTE: This function requires that Bank 7 be enabled and powered. Once this function has completed, Bank 7 may be disabled and powered off.

RWAIT and EWAIT register values must be set before calling this function. There is a helper macro FAPI_WRITE_EWAIT(_mEwait) is provided to make writing the EWAIT value easier due to the need for the EEPROM_CONFIG register to be unlocked before you can write the EWAIT value.

Return Value

- **Fapi_Status_Success** (success)
- **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)

3.1.11 Fapi_remapEccAddress()

Takes ECC address and remaps it to main address space

Synopsis

```
uint32_t Fapi_remapEccAddress(
         uint32_t u32EccAddress)
```

Parameters

u32EccAddress [in]	ECC address to remap

Description

This function returns the main Flash address for the given ECC Flash address passed in u32EccAddress.

Return Value

- **32-bit Main Flash Address**

3.1.12 Fapi_remapMainAddress()

Takes Flash Main address and remaps it to its corresponding ECC address

Synopsis

```
uint32_t Fapi_remapMainAddress(
         uint32_t u32MainAddress)
```

Parameters

u32MainAddress [in]	Main address to remap

Description

This function returns the ECC Flash address for the given Main Flash address passed in u32MainAddress.

Return Value

- **32-bit ECC Flash Address**

3.1.13 Fapi_setActiveFlashBank()

Sets 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 (passed in oNewFlashBank) for any Flash operation issued after calling this function.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_InvalidBank** (failure: Bank specified does not exist on device)

3.2 Asynchronous Functions

3.2.1 Fapi_issueAsyncCommand()

Issues a command to the Flash State Machine that only requires a command

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)
- **Fapi_Error_InvalidCommand** (failure: Command specified requires an address to be specified)

3.2.2 Fapi_issueAsyncCommandWithAddress()

Issues a command to the Flash State Machine with an address

Synopsis

```
Fapi_StatusType Fapi_issueAsyncCommandWithAddress(
        Fapi_FlashStateCommandsType oCommand,
        uint32_t *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.

NOTE: Reading a Flash memory location from the bank that an erase command (sector or bank) is currently being performed will stall the CPU until the erase command finishes and the FMSTAT register indicates the FSM is not busy.

The Bank Erase command is not a suspendable operation.

Return Value

- **Fapi_Status_Success** (success)
- **Fapi_Error_InvalidCommand** (failure: Command specified does not require an address to be specified or is a program command)

3.3 Program Functions

3.3.1 Fapi_issueProgrammingCommand()

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

Synopsis

```
Fapi_StatusType Fapi_issueProgrammingCommand(
        uint32_t *pu32StartAddress,
        uint8_t  *pu8DataBuffer,
        uint8_t  u8DataBufferSizeInBytes,
        uint8_t  *pu8EccBuffer,
        uint8_t  u8EccBufferSizeInBytes,
        Fapi_FlashProgrammingCommandType oMode)
```

Parameters

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

Description

This function sets up the programming registers of the Flash State Machine based on the supplied parameters. It offers four different programming modes to the user 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_AutoEccGeneration – This will program the supplied data portion in Flash along with automatically generated ECC. ECC is calculated on 64-bit aligned addresses up to the data width of the bank. Data not supplied is treated as 0xFF. For example, on a device with a 144-bit wide bank width, if data is written only to bytes 0x0-0x7 (or 0x8-0xF), then the ECC will only be calculated for those 64 bits. If the data supplied crosses a 64-bit boundary, ECC will be calculated for both 64-bit words. For example, on a device with a 144-bit wide bank width, data is written to bytes 0x4 - 0xB, the 2 bytes of ECC data will be calculated. 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 64-bit 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 (programming 3 bytes on a 2 byte ECC wide bank starting at address 0x0 is not allowed).

NOTE: Reading a Flash memory location from the bank that an program command is currently being performed will stall the CPU until the program command finishes and the FMSTAT register indicates the FSM is not busy.

The length of pu8DataBuffer and pu8EccBuffer cannot exceed the bank width of the current active bank. For bank width information, see the *F021 Flash Module* chapter in the device-specific technical reference manual.

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 64-bit aligned or Data length exceeds amount ECC supplied)

3.3.2 Fapi_issueProgrammingCommandForEccAddress()

Synopsis

```
Fapi_StatusType Fapi_issueProgrammingCommandForEccAddress(
        uint32_t *pu32StartAddress,
        uint8_t  *pu8EccBuffer,
        uint8_t   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: Reading a Flash memory location from the bank that a program command is currently being performed will stall the CPU until the program command finishes and the FMSTAT register indicates the FSM is not busy.

The length of pu8EccBuffer cannot exceed the bank width of the current active bank. For bank width information, see the *F021 Flash Module* chapter in the device-specific technical reference manual.

Return Value

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

3.4 Read Functions

The read functions do not combine main and ECC ranges. Because this, if you need a read operation performed on the data and corresponding ECC, you must call the function for both. For example, if you have erased Bank 0, Sector 0, you would need to perform a blank check on Sector 0 main address range and ECC address range.

3.4.1 Fapi_doBlankCheck()

Verifies region specified is erased value

Synopsis

```
Fapi_StatusType Fapi_doBlankCheck(
        uint32_t *pu32StartAddress,
        uint32_t  u32Length,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

Parameters

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

Description

This function checks the device for blank (erase state) starting at the specified address for the length of 32-bit words specified. If a non-blank location is found, these results will be returned in the poFlashStatusWord parameter. This will use read margin 1 mode for this check. As the erase state of the Flash is not a valid ECC condition, on Banks 0 - 6 ECC correction at a minimum must be disabled. On Bank 7, ECC can either be temporarily disabled by writing 0101 to the EE_EDACEN bits or setting the EE_ALL1_OK bit EE_CTRL1 register.

This function assumes the value passed in *pu32StartAddress* is a 32bit aligned address.

NOTE: For devices with L2FMC memory controller, this function has to be executed from RAM.

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)

3.4.2 Fapi_doBlankCheckByByte()

Verifies region specified is erased value by byte

Synopsis

```
Fapi_StatusType Fapi_doBlankCheckByByte(
        uint8_t *pu8StartAddress,
        uint32_t  u32Length,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

Parameters

pu8StartAddress [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 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 8-bit words specified. If a non-blank location is found, these results will be returned in the poFlashStatusWord parameter. This will use read margin 1 mode for this check. As the erase state of the Flash is not a valid ECC condition, on Banks 0 - 6 ECC correction at a minimum must be disabled. On Bank 7, ECC can either be temporarily disabled by writing 0101 to the EE_EDACEN bits or setting the EE_ALL1_OK bit EE_CTRL1 register.

NOTE: For devices with L2FMC memory controller, this function has to be executed from RAM.

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)

3.4.3 Fapi_doVerify()

Verifies region specified against supplied data

Synopsis

```
Fapi_StatusType Fapi_doVerify(
        uint32_t *pu32StartAddress,
        uint32_t  u32Length,
        uint32_t *pu32CheckValueBuffer,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

Parameters

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

Description

This function verifies the device against the supplied data starting at the specified address for the length of 32-bit words specified. If a location fails to compare, these results will be returned in the poFlashStatusWord parameter. This will use normal read, read margin 0 and read margin 1 modes for verifying the data.

This function assumes the value passed in *pu32StartAddress* is a 32bit aligned address.

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)

3.4.4 Fapi_doVerifyByByte()

Verifies region specified against supplied data by byte

Synopsis

```
Fapi_StatusType Fapi_doVerifyByByte(
        uint8_t *pu8StartAddress,
        uint32_t  u32Length,
        uint8_t *pu8CheckValueBuffer,
        Fapi_FlashStatusWordType *poFlashStatusWord)
```

Parameters

pu8StartAddress [in]		start address for region to verify by byte
u32Length [in]		length of region in 8-bit 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 the byte starting at the specified address for the length of 8-bit words specified. If a location fails to compare, these results will be returned in the poFlashStatusWord parameter. This will use normal read, read margin 0 and read margin 1 modes 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)

3.4.5 Fapi_doPsaVerify()

Verifies region specified against specified PSA value

Synopsis

```
Fapi_StatusType Fapi_doPsaVerify(
        uint32_t *pu32StartAddress,
        uint32_t  u32Length,
        uint32_t  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[1]	Actual PSA for read-margin 0
	->au32StatusWord[2]	Actual PSA for read-margin 1
	->au32StatusWord[3]	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.

This function assumes the value passed in *pu32StartAddress* is a 32bit aligned address.

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)

3.4.6 Fapi_calculatePsa()

Calculates the PSA for a specified region

Synopsis

```
uint32_t Fapi_calculatePsa(
         uint32_t *pu32StartAddress,
         uint32_t  u32Length,
         uint32_t  u32PsaSeed,
         Fapi_FlashReadMarginModeType oReadMode)
```

Parameters

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

Description

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

This function assumes the value passed in *pu32StartAddress* is a 32bit aligned address.

Restrictions

The region that the PSA is being calculated must be 32-bit aligned.

Return Value

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

3.4.7 Fapi_doMarginRead()

Reads a region of Flash Memory using specified margin mode

Synopsis

```
Fapi_StatusType Fapi_doMarginRead(
        uint32_t *pu32StartAddress,
        uint32_t *pu32ReadBuffer,
        uint32_t  u32Length,
        Fapi_FlashReadMarginModeType oReadMode)
```

Parameters

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

Description

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

This function assumes the value passed in *pu32StartAddress* is a 32bit aligned address.

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)
- **Fapi_Error_NullPointer** (failure: pu32ReadBuffer is a NULL pointer)

3.4.8 Fapi_doMarginReadByByte()

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

Synopsis

```
Fapi_StatusType Fapi_doMarginReadByByte(
        uint8_t *pu8StartAddress,
        uint8_t *pu8ReadBuffer,
        uint32_t  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 32-bit words to read
oReadMode [in]	indicates which margin mode (normal, RM0, RM1) to use

Description

This function reads the region specified starting at pu8StartAddress for u32Length 8-bit 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)
- **Fapi_Error_NullPointer** (failure: pu8ReadBuffer is a NULL pointer)

3.4.9 Fapi_flushPipeline()

Flushes the FMC pipeline buffers

Synopsis

```
void Fapi_flushPipeline(void)
```

Parameters

None

Description

This function flushes the FMC pipeline buffers.

NOTE: The pipeline must be flushed before the first non-API Flash read after an operation that modifies the Flash contents (Erasing and Programming).

This function makes the assumption that is can read from Flash Addresses 0, 0x100, 0x200, 0x300.

If the Flash and RAM memory regions are swapped, you will need to do 2 dummy reads 128 bytes apart from the remapped Flash memory addresses after an operation that modifies the Flash contents (Erasing and Programming) before performing any API read functions.

Return Value

None

3.5 Informational Functions

3.5.1 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
- 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 0x04
- 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)

3.5.2 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)

3.5.3 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 in kilobytes, 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
- au16SectorSizes[] – An array of sectors sizes for each sector in the bank (*As all sectors on FLEE banks are the same size, only au16SectorSizes[0] is returned with a sector size*)

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)
- **Fapi_Error_NullPointer** (failure: poFlashBankSectors is a NULL pointer)

3.6 Utility Functions

3.6.1 Fapi_calculateFletcherChecksum()

Calculates the Fletcher checksum from the given address and length

Synopsis

```
uint32_t Fapi_calculateFletcherChecksum(
              uint16_t *pu16Data,
              uint16_t 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.

This function assumes the value passed in *pu16Data* is a 16bit aligned address.

Return Value

- • 32-bit Fletcher Checksum value

3.6.2 Fapi_calculateEcc()

Calculates the ECC for a 64-bit value

Synopsis

```
uint8_t Fapi_calculateEcc(
              uint32_t 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

Description

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

This function assumes the value passed in *u32Address* is a 64bit aligned address.

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

- 8-bit calculated ECC

3.6.3 Fapi_waitDelay() -- Deprecated

Generates a delay roughly proportional to the value passed. The delay is created with a simple software loop. The time is not adjusted to account for operating frequency, CPU type or memory access time. This function will not be supported in future versions of this API.

Synopsis

```
uint8_t Fapi_waitDelay(
             uint32_t u32WaitDelay)
```

Parameters

u32WaitDelay [in]	Number of arbitrary units to delay

Description

This function will generate a wait for a time rougnly proportional to the value of u32WaitDelay.

Return Value

- **Fapi_Status_Success** (success)

3.7 User Definable Functions

These callback functions are distributed in the file Fapi_UserDefinedFunctions.c and used by the *read* *Table 4* functions. 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.

3.7.1 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 in the *read* *Table 4* functions. It is called when the address being read crosses the 256 byte aligned address boundaries.

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);
}
```

4 API Macros

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

4.1 FAPI_CHECK_FSM_READY_BUSY

Returns state of FSM.

Synopsis

```
#define FAPI_CHECK_FSM_READY_BUSY (FLASH_CONTROL_REGISTER-
>FmStat.FMSTAT_BITS.BUSY ? Fapi_Status_FsmBusy : Fapi_Status_FsmReady)
```

Parameters

None

Description

This macro returns the state of the FSM.

Return Value

- **Fapi_Status_FsmReady** (FSM is ready to accept a new command)
- **Fapi_Status_FsmBusy** (FSM is busy and may only accept a suspend command if operation is program or erase sector)

4.2 FAPI_CLEAR_FSM_DONE_EVENT

Helper macro to clear FSM_DONE event

Synopsis

```
#define FAPI_CLEAR_FSM_DONE_EVENT (FLASH_CONTROL_REGISTER-
>FedAcStatus.FEDACSTATUS_BITS.FSM_DONE = 1U)
```

Parameters

None

Description

This macro writes a 1 to the FSM_DONE bit in the FEDACSTATUS register to clear the FSM_DONE event.

Return Value

None

4.3 FAPI_GET_FSM_STATUS

Returns the value of the FMSTAT register

Synopsis

```
#define FAPI_GET_FSM_STATUS (FLASH_CONTROL_REGISTER->FmStat.u32Register)
```

Parameters

None

Description

This macro is used to get the value of the FMSTAT register.

FMSTAT Register Value

Figure 1. FMSTAT Register

31							24
Reserved							
23							16
Reserved							
15	14	13	12	11	10	9	8
Reserved	ILA	Reserved	PGV	Reserved	EV	Reserved	BUSY
7	6	5	4	3	2	1	0
ERS	PGM	INV-DAT	CSTAT	VOLTSTAT	ESUSP	PSUSP	SLOCK

Table 8. FMSTAT Register Field Descriptions

Bit	Field	Description
15	Reserved	Read returns 0. Writes have no effect.
14	ILA	Illegal Address When set, indicates that an illegal address is detected. Three conditions can set illegal address flags. • 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	Reserved	Read returns 0. Writes have no effect.
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	Reserved	Read returns 0. Writes have no effect.
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	Reserved	Read returns 0. Writes have no effect.
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.

Table 8. FMSTAT Register Field Descriptions (continued)

Bit	Field	Description
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.

4.4 FAPI_SUSPEND_FSM

Issues FSM suspend command

Synopsis

```
#define FAPI_SUSPEND_FSM  FAPI_WRITE_LOCKED_FSM_REGISTER(FLASH_CONTROL_REGISTER-
>FsmExecute.FSM_EXECUTE_BITS.SUSPEND_NOW, 0x5U)
```

Parameters

None

Description

This macro will issue a FSM suspend command. Only program and erase sector operations are valid suspendable operations.

Return Value

None

4.5 FAPI_WRITE_EWAIT

Helper macro to write EWAIT value

Synopsis

```
#define FAPI_WRITE_EWAIT(_mEwait) FAPI_WRITE_LOCKED_FSM_REGISTER(FLASH_CONTROL_REGISTER-
>EepromConfig.EEPROM_CONFIG_BITS.EWAIT,_mEwait)
```

Parameters

_mEwait [in]	EWAIT value to be written

Description

This macro writes _mEwait to the EWAIT bits in the EEPROM_CONFIG register.

Return Value

None

4.6 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)                        \
    do {                                                                        \
        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; \
 }while(0)
```

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 Flows

Figure 2 and Figure 3 describe the flow for erasing a sector(s) or bank(s) on a device. For further information, see Section 3.2.2.

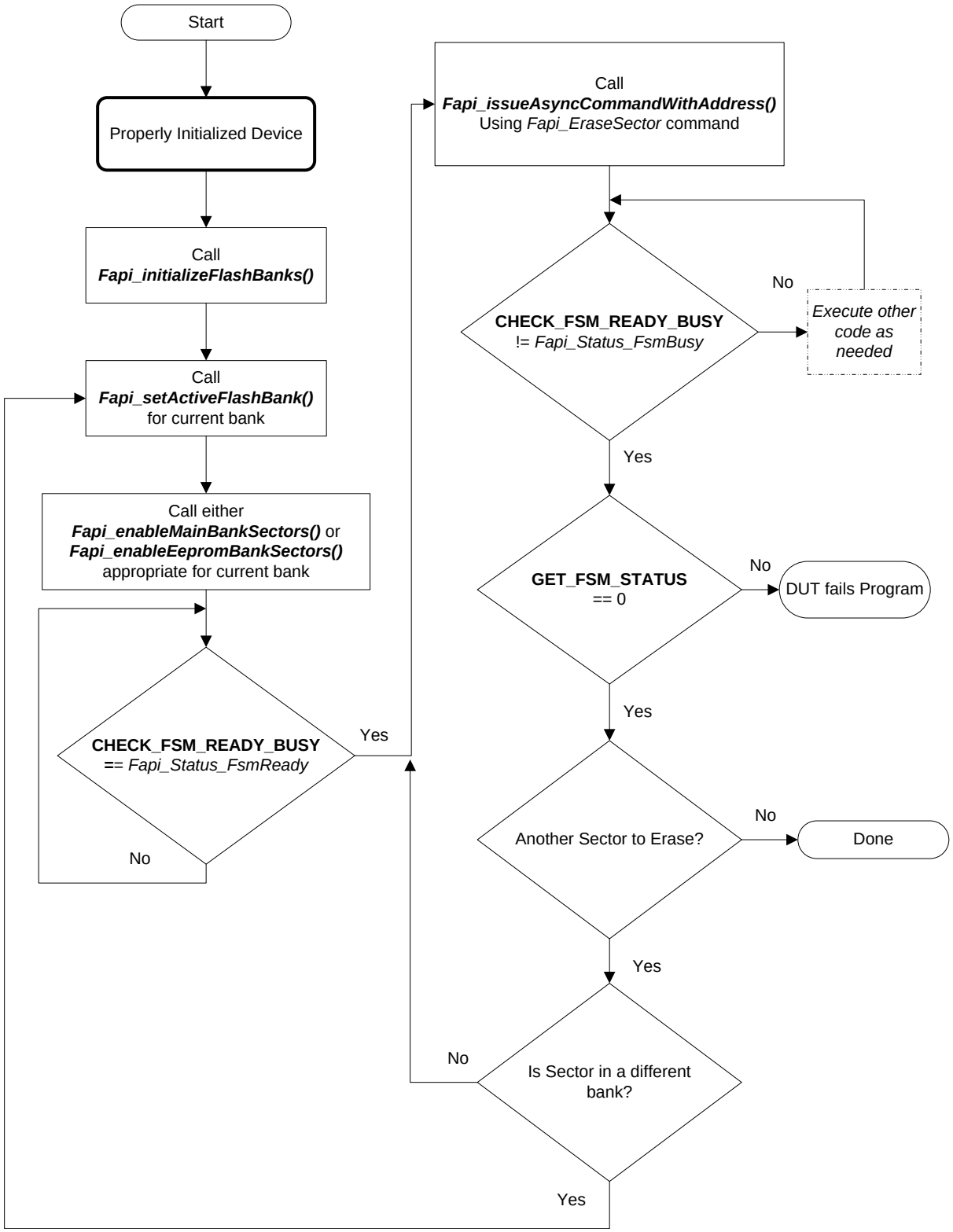

Figure 2. Recommended Sector Erase Flow

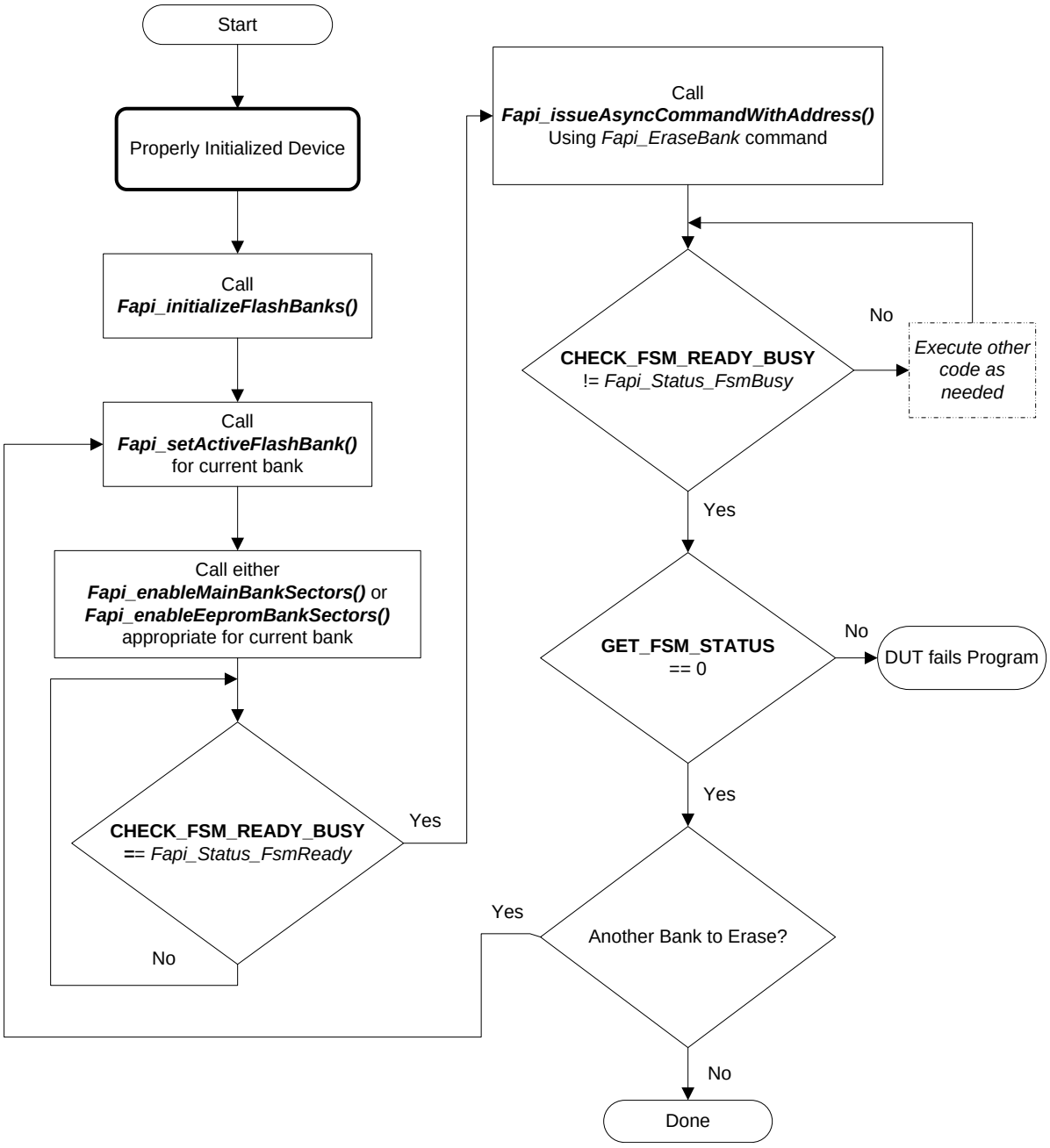

Figure 3. Recommended Bank Erase Flow

5.3 Recommended Program Flow

Figure 4 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). For further information, see Section 3.3.1.

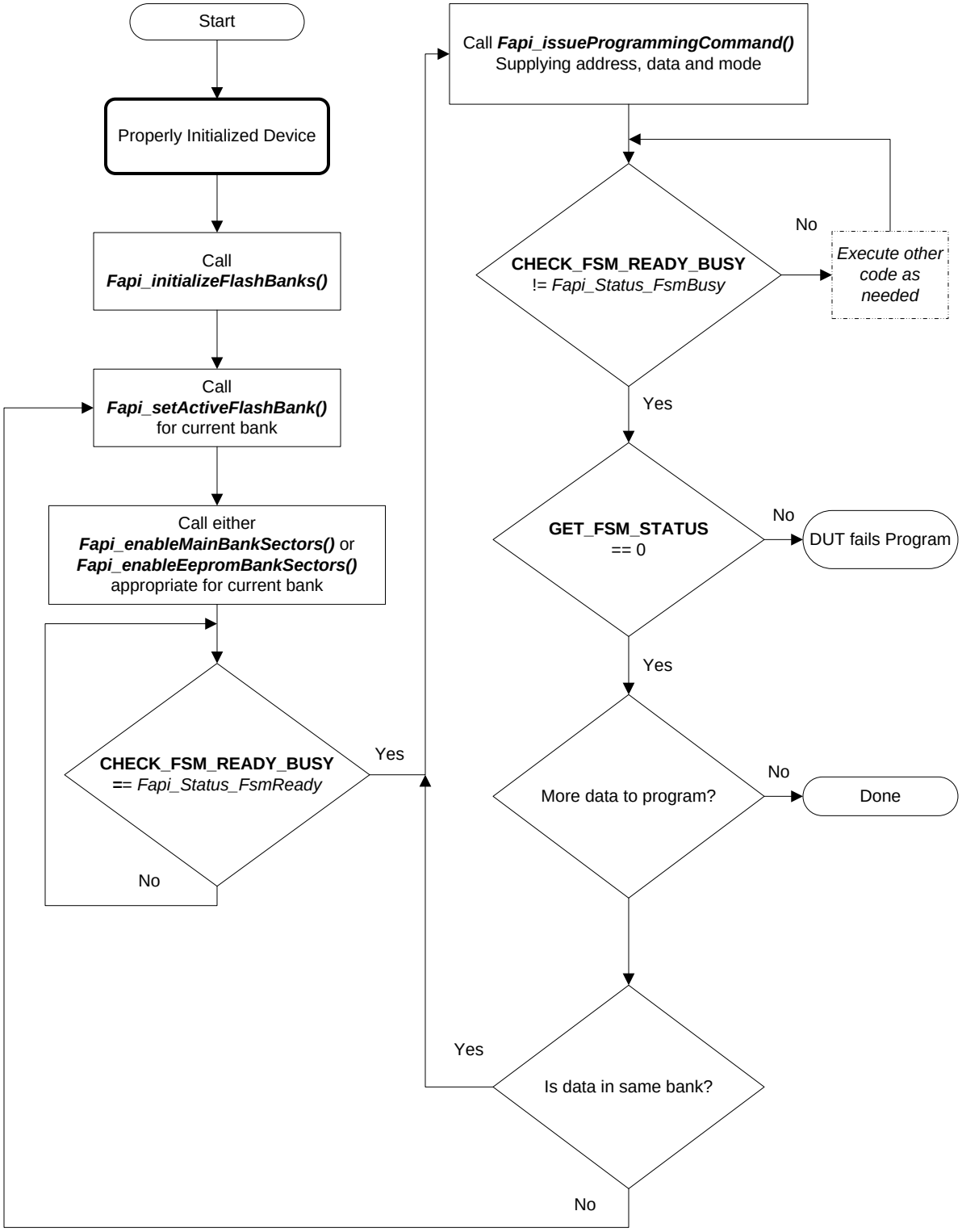

Figure 4. Recommended Program Flow

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_issueProgrammingCommandForEccAddress()
Erase Sector	Used to erase a Flash sector located by the specified address	Fapi_EraseSector	Fapi_issueAsyncCommandWithAddress()
Erase Bank	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()

Typedefs and Enumerations

B.1 Type Definitions

```
typedef unsigned char boolean_t;
```

B.2 Enumerations

B.2.1 Fapi_CpuSelectorType

This is used to indicate which CPU is being used.

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

B.2.2 Fapi_CpuType

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

```
typedef enum
{
   ARM7 = 0U,        /* ARM7 core, Legacy placeholder */
   M3   = 1U,        /* ARM Cortex M3 core */
   R4   = 2U,        /* ARM Cortex R4 core without ECC logic */
   R4F  = 3U,        /* ARM Cortex R4, R4F, and R5 cores with ECC logic*/
   C28  = 4U,        /* TI C28x core */
   Undefined1 = 5U,  /* To Be Determined.  Future core placeholder */
   Undefined2 = 6U,  /* To Be Determined.  Future core placeholder */
   Undefined3 = 7U   /* To Be Determined.  Future core placeholder */
}  ATTRIBUTE_PACKED Fapi_CpuType;
```

B.2.3 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;
```

B.2.4 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;
```

B.2.5 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;
```

B.2.6 Fapi_FlashBankType

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

```
typedef enum
{
   Fapi_FlashBank0=0,
   Fapi_FlashBank1=1,
   Fapi_FlashBank2=2,
   Fapi_FlashBank3=3,
   Fapi_FlashBank4=4,
   Fapi_FlashBank5=5,
   Fapi_FlashBank6=6,
   Fapi_FlashBank7=7
}  ATTRIBUTE_PACKED Fapi_FlashBankType;
```

B.2.7 Fapi_FlashBankTechType

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

```
typedef enum
{
    Fapi_FLEP=0,
    Fapi_FLEE=1,
    Fapi_FLES=2,
    Fapi_FLHV=3
}  ATTRIBUTE_PACKED Fapi_FlashBankTechType;
```

B.2.8 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;
```

B.2.9 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;
```

B.2.10 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;
```

B.2.11 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_Error_Fail,                 /* Generic Function Fail code */
   Fapi_Error_NullPointer,          /* One of the pointer parameters is a null pointer */
   Fapi_Error_InvalidCommand,       /* Command used is invalid for the function called */
   Fapi_Error_InvalidEccAddress,    /* Returned if the ECC Address given to a function is
invalid for that function */
   Fapi_Error_OtpChecksumMismatch,  /* Returned if OTP checksum does not match expected value */
   Fapi_Error_InvalidHclkValue,     /* Returned if FClk is above max FClk value -
 FClk is a calculated from HClk and
                                       RWAIT/EWAIT */
   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,   /* Returned if Data buffer size specified
exceeds Data bank width */
   Fapi_Error_AsyncIncorrectEccBufferLength,    /* Returned if ECC buffer size specified
exceeds ECC bank width */
   Fapi_Error_AsyncDataEccBufferLengthMismatch, /* Returned if Data buffer size either is not
64bit aligned or Data
                                                   length exceeds amount ECC supplied */
   Fapi_Error_FeatureNotAvailable  /* FMC feature is not available on this device */
}  ATTRIBUTE_PACKED Fapi_StatusType;
```

B.2.12 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;
```

Flash Validation Procedure

TI distributes the F021 Flash API as pre-compiled object libraries which has been fully qualified for use on Hercules devices. This ensures that the object code for programming will be the same as was qualified by TI. However it might be still possible that the application program does not use the API functions correctly by either passing incorrect values or using the API functions in a manner not in accordance with the API documentation. In order to ensure proper usage of the Flash API, the in application and 3rd programming tools must be validated.

1. Erase and program at least 4 devices with customer's object code using the method selected by the customer for programming.
2. Using the TI profiling tool (contact local TI representative to obtain), generate profiles for each device programmed in 1.
3. Erase and program these devices again using the same data pattern used in step 1 with the TI tool, UniFlash.
4. Repeat step 2 on the devices programmed with UniFlash.

The profiles generated in steps 2 and 4 should correlate within 100mV or 18 indices on the 0's Vt and be within 3 indices on the 1's BCC.

NOTE: It is also highly recommended that the F021 Flash API usage be thouroughly reviewed by someone who was not involved with writing the code for Flash usage.

Parallel Signature Analysis (PSA) Algorithm

The functions Section 3.4.6 and Section 3.4.5 make use of the Parallel Signature Analysis (PSA) algoritm. 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_t calculatePSA (uint32_t* pu32StartAddress,
                           uint32_t  u32Length,          /* Number of 32bit words */
                       uint32_t  u32InitialSeed)
{
   uint32_t 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;
}
```

Revision History

Table 10 lists the API versions.

Table 10. API Version History

Version	Additions/Modifications/Deletions
1.00.0	Initial Revision
1.00.1	Added missing extern reference for Fapi_getBankSectors()
1.50.0	• Added support for Concerto devices. • Corrected issue with with Fapi_doBlankCheck() and Fapi_doVerify that could cause an illegal address abort. • Added ECC read support for 72-bit wide FLEE banks
1.51.0	• SDOCM00086276 - Changed the u64Data parameter passed to Fapi_calculateEcc() to be consistent behavior on Big and Little Endian Systems. • SDOCM00087304 - Updated Fapi_doBlankCheck to use Read Margin mode 0 only. • SDOCM00086404 - Added macro FAPI_WRITE_LOCKED_FSM_REGISTER() to allow easier writes to locked register in the Flash Memory Controller. • SDOCM00086405 - Changed the functions Fapi_setupEepromSectorEnable() and Fapi_setupBankSectorEnable() in Fapi_UserDefinedFunctions.c to use the macro FAPI_WRITE_LOCKED_FSM_REGISTER() as the register that were being written to are locked register in the Flash Memory Controller. • SDOCM00086402 - Corrected the issue in Fapi_doMarginRead() that did not return all the requested data in the ECC memory regions. • SDOCM00088256 - Changed the check that insures max FCLK value for the device is not exceeded from a defined constant in the API to a check from the device OTP value. • SDOCM00087302 - Updated the structures in Types.h to support linking in gcc toolchain.
2.00.00	• Replaced the user defined callback functions Fapi_setupEepromSectorEnable() and Fapi_setupBankSectorEnable() with the functions Fapi_enableEepromBankSectors() and Fapi_enableMainBankSectors. • Deprecated the function Fapi_waitDelay(). • Removed the header files F021_FMC_BE.h and F021_FMC_LE.h as F021.h has been updated to automatically determine compile endianness. • Replaced the Fapi_initializeAPI() function with Fapi_initializeFlashBanks(). With this change, all global variables have been removed from the API. • Added the Compatibility.h header file. This file contains some backwards compatibility macros to work with projects that were previously built with v1.51 of the API. The list of functions and global variables with compatibility defines are: – Fapi_initializeAPI() – Fapi_getFsmStatus() – Fapi_issueFsmSuspendCommand() – Fapi_writeEwaitValue(mEwait) – Fapi_checkFsmForReady() – Fapi_GlobalInit.m_poFlashControlRegisters • Fapi_getBankSectors() was updated to return sector sizes in kilobytes and to support 256kB sectors, au8SectorSizes which was an array uint8_t was changed to an array of uint16_t and renamed au16SectorSizes.

Table 10. API Version History (continued)

Version	Additions/Modifications/Deletions
2.00.00	• Added Fapi_remapMainAddress() to give an easy method to determine ECC address for a main flash address • Removed unused status' from Fapi_StatusType – Fapi_Status_AsyncBusy – Fapi_Status_AsyncComplete – Fapi_Error_StateMachineTimeout – Fapi_Error_InvalidDelayValue – Fapi_Error_InvalidCpu • Removed the listing of structures. Please refer to the installed F021 Flash API headers files for these. • Changed from the use of defined typedefs uint64, uint32, uint16, and uint8 to the standard definitions in stdint.h, uint64_t, uint32_t, uint16_t, and uint8_t. Also changed boolean to boolean_t • Added #if defined guardbanding around the defines in Types.h that can conflict with Autosar Platform_Types.h defines. • Added appendix describing the PSA calculation
2.00.01	• Corrected the function description for Fapi_enableMainBanks(). • Added additional files that are distributed with the library.
2.01.00	• Added additional library files for L2FMC with floating point support. • SDOCM00102756 - Remove FLOCK register from register include file. • SDOCM00103134 - Sector size returned for FLEE banks by Fapi_getBankSectors() is double the actual size.
2.01.01	• Added Blank Check routine for devices with L2FMC controller. • OTP check for Fapi_enableEepromBankSectors and Fapi_enableMainBankSectors. • SDOCM00102084 - Typo in CGT.CCS.H in GNU attribute check . • SDOCM00102399 - FEDACSDIS and FEDACSDIS2 are missing from Fapi_FmcRegistersType definition. • SDOCM00100674 - Add Blank check function for Conqueror devices. • SDOCM00107638 - Macros FAPI_SUSPEND_FSM and FAPI_WRITE_EWAIT are malformed. • SDOCM00108528 - In F021, dot operator usage in macro causes compiler warning. • All libraries compiled without debug symbols included in the distributed files.

Table 11 lists the API changes made since the previous revision of this document.

Table 11. Document Revision History

Reference	Additions/Modifications/Deletions
-	Initial revision
A	Updated for v1.50.0
B	Updated the example code for Fapi_issueProgrammingCommand()
C	Updated for v1.51.0
D	Updated for v2.00.00
E	Updated for v2.00.01
F	• Added the Validation procedure • Corrected the register field name to disable ECC on bank 7 before calling a Blank Check function.
G	Updated for v2.01.01
H	Updated Section 3.4.5

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

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		

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