TMS320F2803x Piccolo™ Flash API using CCS4.0v

For creating custom programming solutions for the TMS320F2803x DSPs.

This document applies to the following API:

TMS320F2803x Flash API V1.00 using CCS4.0v
(2803x_FlashAPI_BootROMSymbols.lib)

Release version: 2010.05.03
Release date: May 03, 2010

Flash API Disclaimer

This documentation applies to the following Flash Application Program Interface (Flash API) libraries:

- **2803x_FlashAPI_BootROMSymbols.lib**

Texas Instruments Inc. (TI) reserves the right to update or change any material included with this release. This includes:

- **The API functional behavior based on continued TMS320F2803x testing.**
- **Improvements in algorithm performance and functionality.**

It is the user's responsibility to check for future updates to these APIs and to use the latest version available for their TMS320F2803x silicon.

Should functional changes occur to the APIs, it is the user's responsibility to update any application that uses the API (programmers, embedded software, etc) for proper long-term operation of the flash.

Updates to the API will be posted on the Texas Instruments Inc website (www.ti.com) and can also be obtained by contacting a local TI representative or the TI Product Information Center.

In this document, the terms F2803x, and TMS320F2803x are used to refer to all flash devices within the family.

The device datasheet has precedence over this document pertaining to all specifications and device behavior.

Contents:

1. Release Notes

These release notes apply to all of the TMS320F2803x devices.

a) The Flash API has been compiled with the large memory model (-ml) enabled. The small memory model is not supported. Any application that uses the Flash API should also be compiled for the large memory model. For information on the large memory model refer to the *TMS320C28x Optimizing C/C++ Compiler User's Guide* (literature #SPRU514).

b) Some traditional programming utilities have separate operations for "Clear" and "Erase". These two operations have been combined into one operation referred to only as "Erase".

c) Note: The CSM will be <u>permanently</u> locked if the CSM password locations are loaded with all 0x0000 and the device is secured. During the combined "Erase" function, a sector clear is immediately followed by an erase operation without resetting the device. This will help avoid permanently locking the CSM. Do not program the CSM passwords with all 0x0000.

d) The intended use of the Flash API software is for development of custom flash programming methods. The Flash API is used with ROM boot loading options such as parallel load 16/8, SCI, eCAN, I2C and SPI modes, to get the flash programming code into the DSP.

e) For programming the device through the JTAG port, use the SDFlash programmer from TI 3rd Party vendor Spectrum Digital Inc. (www.spectrumdigital.com) or the Code Composer Studio™ (CCS) Plug-in. Check the TI website, and CCS update advisor for updates to the plug-in to incorporate this version of the API.

f) TMS320F2803x API is embedded into the Boot ROM on the device. This differs from other 28x devices where the API is wholly software. As such both a software library (2803x_FlashAPI_BootROMSymbols.lib) are provided. This library file should be included in the application with the symbol library given preference.

2. Software Library Revision vs. Silicon Revision

The silicon revision can be determined by the lot trace code marked on the top of the package. Refer to the device specific errata for information on how to determine your device's silicon revision from the device symbolization. In addition, the silicon revision can be read from the REVID register located at address 0x883 as shown in the tables below.

Revision information for the F2803x API

Second Letter In Prefix of Trace Lot Code	REVID (Addr 0x883)	ROM API (Symbol Library) Revision	Obsolete – Software Library These Software Libraries are Obsolete and No Longer Recommended	Recommended Software Library
Blank	= 0x0000	1.00	None	**V1.00**

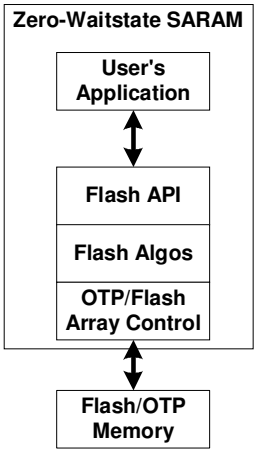
Note:
For future silicon revisions, TI anticipates that no functional changes will be required to these APIs (and hence no changes should be required to your DSP software).

Should API changes occur that affect the programming of the flash, it is the user's responsibility to update any application that uses the API (programmers, embedded software, etc) to insure proper long-term operation of the flash. TI will test these APIs on future silicon revisions as soon as possible when such devices become available. Updates that only add features are not required.

3. Introduction: Flash API Programming Fundamentals

The Flash Application Program Interface (Flash API) consists of well-documented functions that the client application calls to perform flash specific operations. The flash array and One Time Programmable (OTP) block on the device are managed via CPU execution of algorithms in the Flash API library. Texas Instruments Inc (TI) provides API functions to erase, program and verify the flash array as briefly described here:

Erase:
Erase operates on the flash array only. The One Time Programmable (OTP) block cannot be erased once it has been programmed. The Erase function is used to set the flash array contents to all 1's (0xFFFF). The erase operation includes the following steps:



- Pre-compact all sectors. This step is to make sure no bits are in an over-erased or "depleted" state before attempting the sector erase. Depletion can occur as a result of stopping the erase function before its post-condition or compaction step can complete. Even with this step, halting the erase function before it completes is not recommended.
- Pre-condition or "clear" the sector to be erased. This step programs all of the bits in the sector to 0x0000 to allow for an even erase across the sector.
- Erase the sector. This step removes charge from the bits in the sector until all of the bits within the sector are erased.
- Post-condition or compact the sector that was erased. This step makes sure no bits are left in an over-erased (or depleted) state.

The smallest amount of memory that can be erased at a time is a single sector. Some traditional algorithms, such as those for the 240x family, require that the flash be pre-conditioned or "cleared" before it is erased. The Flash API erase function includes the flash pre-conditioning and a separate "clear" step is not required.

The flash array and OTP block are in an erased state (all 0xFFFF) when the device is shipped from the factory.

Program:
The program function operates on both the flash array and the OTP block. This function is used to put application code and data into the flash array or OTP. The program function can only change bits from a 1 to a 0. Bits cannot be moved from a 0 back to a 1 by the programming function. For this reason, flash is typically in an erased state (all 0xFFFF) before calling the programming function. The programming function operates on a single 16-bit word at a time.

To protect the flash or OTP and allow for user flexibility, the program operation will not attempt to program any bit that has previously been programmed. For example, a flash or OTP location can be programmed with 0xFFFE and later the same location can be programmed with 0xFFFC without going through an erase cycle. During the second programming call, the program operation will detect that bit 0 was already programmed and will only program bit 1.

Verify:
The erase and program functions perform verification with voltage margin as they execute. The verify function provides a second check via a CPU read that can be run to verify the flash contents against the reference value. The verify function operates on both the flash array and OTP blocks.

To integrate one of the Flash APIs into your application you will need to follow the steps described within this document. The checklist provided in section 5 gives an overview of the required steps and can be used to guide you through the process. While integrating the API, keep the following Do's and Don'ts in mind:

API Do's:

- Execute the Flash API code from zero-wait state internal SARAM or ROM memory.
- Configure the API for the correct CPU frequency of operation.
- Follow the Flash API checklist in section 5 to integrate the API into an application.
- Initialize the PLL control register (PLLCR) and wait for the PLL to lock before calling an API function.
- Initialize the API callback function pointer (Flash_CallbackPtr). If the callback function is not going to be used then it is best to explicitly set the function pointer to NULL as described in section 13. Failure to initialize the callback function pointer can cause the code to branch to an undefined location.
- Carefully review the API restrictions for the callback function, interrupts, and watchdog described in section 15.

API Don'ts:

- Don't execute the Flash APIs from the flash or OTP. If any API functions are stored in flash or OTP memory, they must first be copied to internal zero-wait state SARAM before they are executed.
- Don't execute any interrupt service routines (ISRs) that can occur during an erase, program or depletion recovery API function from the flash or OTP memory blocks. Until the API function completes and exits the flash and OTP are not available for program execution or data storage.
- Don't execute the API callback function from flash or OTP. When the callback function is invoked by the API during the erase, program or depletion recovery routine the flash and OTP are not available for program execution or data storage. Only after the API function completes and exits will the flash and OTP be available.
- Don't stop the erase, program or depletion recovery functions while they are executing (for example, don't stop the debugger within API code, don't reset the part, etc).
- Do not execute code or fetch data from the flash array or OTP while the flash and/or OTP is being erased, programmed or during depletion recovery.

4. Example Programs

An example program is included with the API. In a typical installation, the examples are located in the following directories:

- F28035: C:\tidcs\c28\Flash28_API\Flash28035_API_V100\example

These examples demonstrate the following:

- Required software setup before calling the API (setting the PLL, checking for limp mode etc.)
- How to copy the API from flash into SARAM for execution.
- How to call the API functions.

The steps taken by the example are outlined in the API checklist in section 5 and described in detail in the remainder of this document. The example program is setup to be stored in the flash and the appropriate code and constants are copied to SARAM for execution. You must first build the application and then program it into the flash using an available programmer such as SDFlash available from Spectrum Digital Inc. (www.spectrumdigital.com) or the Code Composer Plug-in. Additional programmers from 3rd parties may be available, please contact TI for details.

To step through the code, use Code Composer Studio to load the symbols from the .out file (File->Load Symbols).

5. Flash API Checklist

Integration of the Flash API into user software requires that the system designer implement operations to satisfy several key requirements. The following checklist gives an overview of the steps required to use the API. These steps are further discussed in detail in the reference section indicated.

Before using the API, do the following:

	Step	Description	Reference
	1	Modify Flash2803x_API_Config.h for your target operating conditions.	Section 6
	2	Include Flash2803x_API_Library.h in your source code.	Section 7
	3	Add the proper Flash API library to your project. When using the Flash API, build your code with the large memory model. The API Library is built in 28x Object code (OBJMODE = 1, AMODE = 1)	Section 8

In your application, before calling any Flash API functions do the following:

	Step	Description	Reference
	4	Initialize the PLL control register (PLLCR) and wait for the PLL to lock.	Section 9
	5	Make sure the PLL is not running in limp mode. If the PLL is in limp mode, do not call any of the API functions as the device will not be running at the proper frequency.	Section 10
	6	Optional: The API must execute from zero-wait state internal SARAM or ROM. If the API is to be copied from flash/OTP into internal SARAM memory then follow the instructions in this section.	Section 11
	7	Initialize the 32-bit global variable Flash_CPUScaleFactor	Section 12
	8	Initialize the global function pointer Flash_CallbackPtr to point to the application's callback function. Alternatively set the pointer to NULL.	Section 13
	9	Optional: Disable global interrupts before calling an API function.	Section 14
	10	Understand the API restrictions detailed in this section before making any API calls.	Section 15
	11	Optional: Run the frequency toggle test to confirm proper frequency configuration of the Flash API. Note: The ToggleTest function will execute forever. You must halt the processor to stop this test.	Section 16
	12	Optional: Unlock the code security module (CSM).	Section 17
	13	Call the Flash API Functions described in the API Reference	Section 18

The called flash API function will do the following:

- The watchdog timer is disabled. (Section 15).
- Performs the called operation and:
  - Disables and restores global interrupts (via INTM, DBGM, XNMICR) around time critical code segments. (Section 14).
  - Invokes the callback function if Flash_CallbackPtr is not NULL.
- Returns success or an error code. These are defined in F2803x_API_Library.h (Section 18.11)

The user's code should then do the following:

	Step	Description	Reference
	14	Check the return status against the error codes.	Section 18.11
	15	Optional: Re-enable the watchdog timer.	

6. Step 1: Modify Flash2803x_API_Config.h

Modify Flash2803x_API_Config.h file, found in the include directory of each API, to match your specific target operating conditions.

6.1. Specify the clock rate of the CPU (SYSCLKOUT) in nanoseconds.

Uncomment the line corresponding to the CPU Clock rate (SYSCLKOUT) in nanoseconds at which the API functions will be run at. This is done by removing the leading // in front of the correct line. Only one line should be uncommented. The file lists a number of commonly occurring clock rates. If your CPU clock rate is not listed, then provide your own definition using the examples as a guideline.

For example: Suppose the final CPU clock rate will be 55 MHz. This corresponds to a 18.182 nS cycle time. There is no line present for this clock speed, so you should insert your own entry and comment out all other entries:

```
/*-------------------------------------------------------------------
*  Flash2803x_API_Config.h
*-------------------------------------------------------------------*/
 ...
//#define CPU_RATE   16.667L   // for a 60MHz CPU clock speed (SYSCLKOUT)
#define CPU_RATE   18.182L   // for a 55MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE   20.000L   // for a 50MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE   25.333L   // for a 40MHz CPU clock speed (SYSCLKOUT)
 ...
```

The CPU clock rate is used during the compile to calculate a scale factor for your operating frequency. This scale factor will be used by the Flash API functions to properly scale software delays that are VITAL to the proper operation of the API.

The formula, found at the bottom of the Flash2803x_API_Config.h, file for this calculation is:

```
/*-------------------------------------------------------------------
*  Flash2803x_API_Config.h
*-------------------------------------------------------------------*/
 ...
 #define SCALE_FACTOR  1048576.0L*( (200L/CPU_RATE) )
 ...
```

CAUTION

For flash integrity at operating frequencies, the device should always be programmed at the fastest possible CPU frequency. For example, if the CLKIN frequency is 10 MHz program the device at 60 MHz rather than 10 MHz.

The flash API is not designed to function properly below 10 MHz.

7. Step 2: Include Flash2803x_API_Library.h

Flash2803x_API_Library.h is the main include file for the Flash API and should be included in any application source file that interfaces to the Flash API.

Flash2803x_API_Library.h contains the following:

- Error code definitions. Refer to section 18.11.
- Sector bit mask definitions that can be used when calling the erase function.
- Flash status structure (FLASH_ST) definition used by the API functions to return information back to the calling routine.
- Function prototypes for each API library. Refer to section 18.
- Frequency scale factor definition: Flash_CPUScaleFactor. Refer to section 12.
- Pointer to callback function definition: Flash_CallbackPtr. Refer to section 13.
- Macros to enable easy porting between the API libraries. Note: depending on which API is being used; some 2803x devices may not be included in this file since they were released to market at a later time. The header file for the newer device can be used if required. Refer to section 18.1.

8. Step 3: Include the proper Flash API libraries

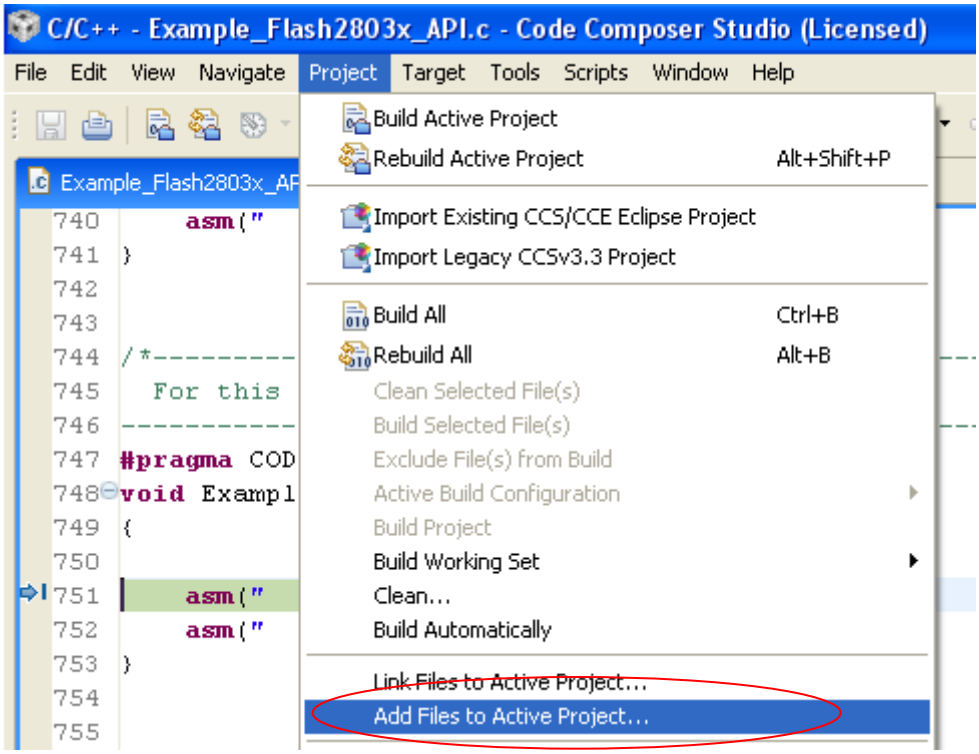
The proper Flash API libraries must also be included in your project. In F2803x, a symbol library called "2803x_FlashAPI_BootROMSymbols.lib" needs to be used. This library contains the addresses of the various API functions that are embedded into the device Boot ROM. Since all of the functions reside in ROM, adding the API to an application only takes up a small amount of extra RAM and Flash space.

By default `<> = C:\tidcs\c28`

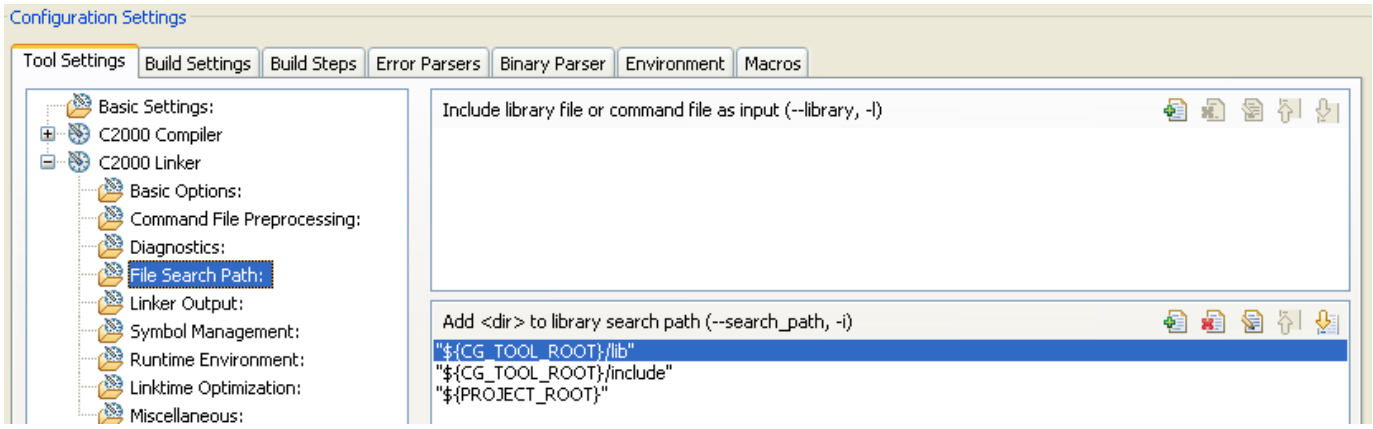
`<>\Flash28_API\Flash2803x_API_V100\lib\2803x_FlashAPI_BootROMSymbols.lib`

The build order should be set so that the symbol library is linked in first. This way, if the function resides in ROM, it will be taken. To do this perform the following actions:

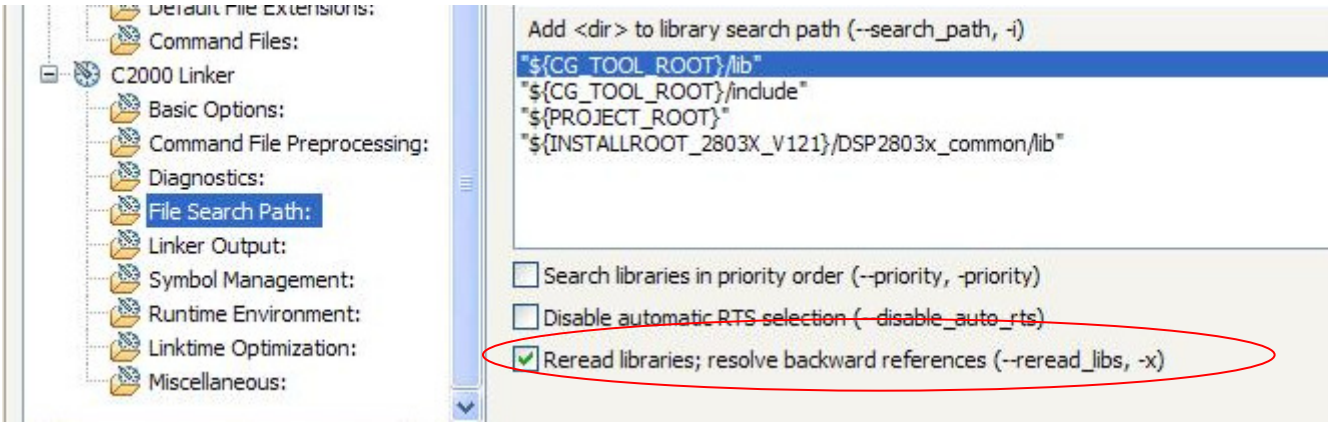
Step 1: Add the 2803x_FlashAPI_BootROMSymbols.lib libraries to the project.

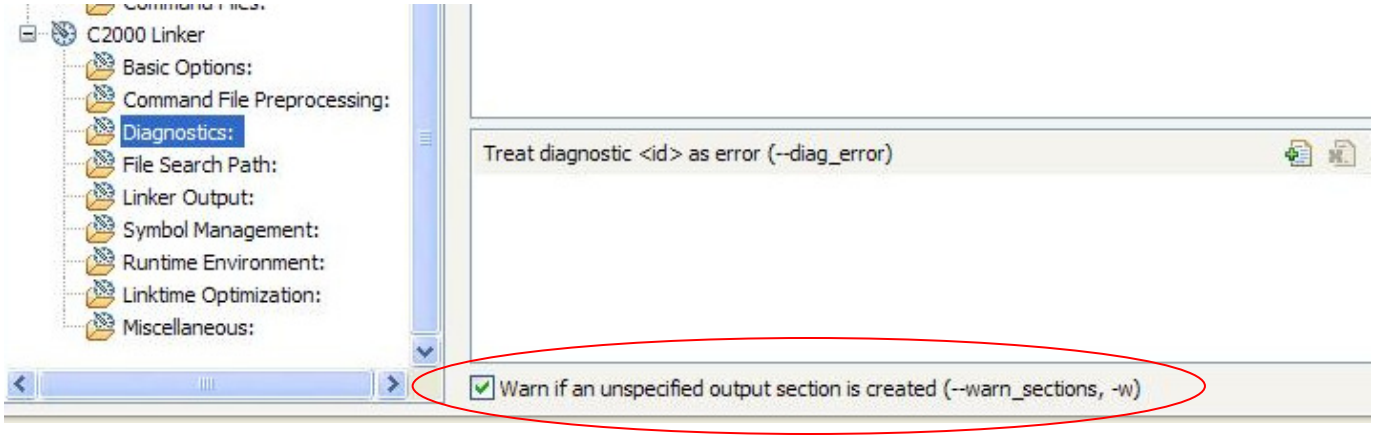


Step 2: Make sure the "Incl. Library file (-l):" field under the File Search Path in C/C++ build → C2000 Linker is empty.

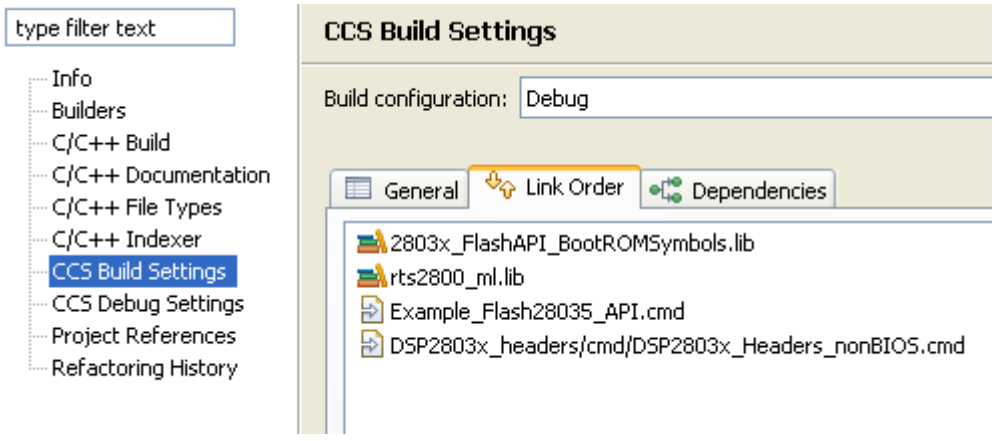


Step 3: Check the “Resolve Symbols To First Library (-priority)” box under the Linker/Advanced tab in the Build Options menu.





Step 4: Place the symbol library first in the Link Order tab in the Build Options menu.



The Flash APIs have been compiled with the large memory model (-ml) option. The small memory model option is not supported. For information on the large memory model refer to the *TMS320C28x Optimizing C/C++ Compiler User's Guide* (literature #SPRU514).

9. Step 4: Initialize PLL Control Register (PLLCR)

It is vital that the API functions be run at the proper operating frequency. To achieve this, the calling application must initialize the PLLCR register before calling any of the API functions. To change the PLLCR follow the flow outlined in the *TMS320x2803x System Control and Interrupts Reference Guide*. Following this flow is important in order to make sure that the PLL is not operating in limp mode before changing the PLLCR register.

As part of this initialization, the calling application must guarantee that the PLL has had enough time to lock at the new frequency before making API calls. To do this the application can monitor the PLLLOCKS bit in the PLLSTS register. When this bit is set it indicates that the PLL has completed locking and the CPU is running at the specified frequency.

10. Step 5: Check PLL Status for Limp Mode Operation

The API functions contain time critical code with software delay loops that must execute to meet specific timing requirements. For this reason, the device must be operating at the correct CPU frequency before the Flash API functions are called. If the input clock to the device has gone missing, the PLL will enter what is called limp mode operation and the CPU will be clocked at a much lower frequency. When this happens the device is reset and the MCLKSTS bit will be set in the PLLSTS register. If this bit is set, the API functions should not be called.

Refer to *TMS320x2803x System Control and Interrupts Reference Guide* for more information on the missing clock detection logic of the F2803x devices.

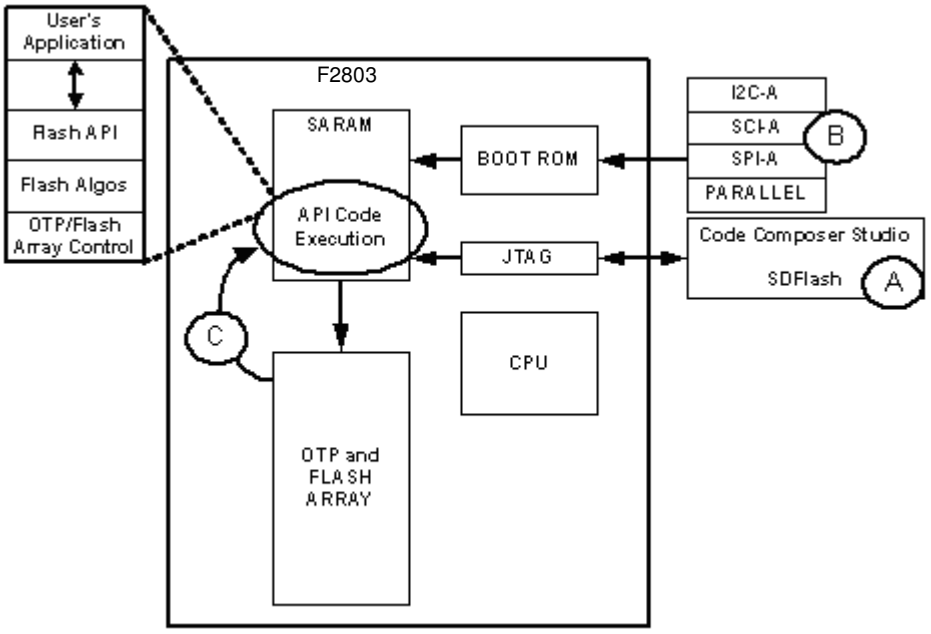
11. Step 6: Copy the Flash API functions to Internal SARAM

There are two factors that restrict the type of memory that the Flash API functions can be executed from:

- On these devices, there is only one flash array. The flash architecture imposes the restriction that the flash can perform only one operation at a time. Due to this restriction, when erasing, programming, or verifying the flash, code itself cannot execute and data cannot be fetched from the flash.

- There are required delays within the flash API functions that are vital to their proper operation. These delays are implemented via cycle-sensitive software delays. To be accurate, these delays must execute from zero-waitstate memory.

To satisfy these two restrictions, all flash API functions must be executed from on-chip, zero-wait state SARAM or ROM memory. The figure below illustrates three different methods that can be used to load the API code into the device.



Method A: The software API code is loaded directly into on-chip SARAM via the JTAG port. This is the method used by Code Composer Studio and the SDFlash utility.

Method B: The software API code is loaded directly into on-chip SARAM by one of the ROM boot loaders (SCI, SPI or Parallel). This is the method used by custom programmers such as SCI programmers. This method can also be extended to other peripherals by programming a custom loader into the OTP block. Note: all boot modes shown may not be available on a specific device.

Method C: The software API code is embedded within an application that is stored in the flash or OTP. In this case the API code must first be copied out of flash into on-chip SARAM before it is executed.

If the software API code is loaded directly into on-chip, zero-waitstate SARAM as shown in method A or B then this step can be skipped. If, however, the software API code is stored in flash or OTP, then the calling application must first copy the required code into SARAM before making any calls into the API. The following describes how to accomplish this copy.

Steps to copy the API functions from flash to SARAM:

In the linker command (.cmd) file, create a group section called Flash28_API as shown.

TMS320F2803x Example:

```
/*---------------------------------------------------------------------
*  User's .cmd file: TMS320F2803x API Library Group Section Example
*---------------------------------------------------------------------*/
...
   SECTIONS
   {
   ramfuncs             : LOAD = FLASHA,
                          RUN = PRAML0,
                          LOAD_START(_RamfuncsLoadStart),
                          LOAD_END(_RamfuncsLoadEnd),
                          RUN_START(_RamfuncsRunStart),
                          PAGE = 0

...
   }
```

This group section defines symbols that the linker will assign to the load start, load end, and run start addresses of the section.

For the example shown, the linker will assign the following symbols:

- Load address start: RamfuncsLoadStart
- Load address end: RamfuncsLoadEnd
- Run address start: RamfuncsRunStart

These symbols are already declared in the main library include file, Flash2803x_API_Library.h

```
/*---------------------------------------------------------------------
*  Flash2803x_API_Library.h
*---------------------------------------------------------------------*/
...
extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsLoadEnd;
extern Uint16 RamfuncsRunStart;
...
```

These three symbols can then used to copy the Flash API functions from the flash memory to the SARAM as shown in the included sample programs.

```
/*-----------------------------------------------------------------
*  User's application .c file: Example call to a memory copy routine
*-----------------------------------------------------------------*/

#include Flash2803x_API_Library.h
...

   Example_MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);

...
```

```
/*-----------------------------------------------------------------
*  User's application .c file: Example memory copy routine
*-----------------------------------------------------------------*/
void Example_MemCopy(Uint16 *SourceAddr, Uint16* SourceEndAddr, Uint16* DestAddr)
{
    while(SourceAddr < SourceEndAddr)  { *DestAddr++ = *SourceAddr++; }
    return;
}
```

This same method and copy routine can be used to copy any additional code and data that is needed during the programming operation.

12. Step 7: Initialize Flash_CPUScaleFactor

Flash_CPUScaleFactor is a global 32-bit variable defined by the Flash API functions. The Flash API functions contain several delays that are implemented as software delays. The correct timing of these software delays is vital to the proper operation of the API functions. The 32-bit global variable Flash_CPUScaleFactor is used by the API functions to properly scale these software delays for a particular CPU operating frequency (SYSCLKOUT).

First, make sure the proper CPU rate in nanoseconds is defined in the library configuration file Flash2803x_API_Config.h. This step is described in section 6.

The corresponding Flash_CPUScaleFactor value for the defined CPU rate is calculated during the compile by the following formula:

```
/*-------------------------------------------------------------------
*  Flash2803x_API_Config.h
*-------------------------------------------------------------------*/
...
 #define SCALE_FACTOR  1048576.0L*( (200L/CPU_RATE) )
...
```

CAUTION

The SCALE_FACTOR formula is already defined in the file Flash2803x_API_Config.h. This formula must not be modified. Doing so will cause improper operation of the flash API functions.

The calling application must then initialize the global variable Flash_CPUScaleFactor as follows before calling any API function. Additionally, since the majority of the API is embedded into the ROM with hard addresses the location of the Flash_CPUScaleFactor variable must be at address 0x000D04. To ensure this use the following compiler pragma directive in combination with the DSP2803x_Headers_nonBIOS.cmd linker command file:

```
/*-------------------------------------------------------------------
*  Flash2803x_API_Library.h
*-------------------------------------------------------------------*/
...
extern Uint32 Flash_CPUScaleFactor;
```

```
/*-----------------------------------------------------------------
*  User's application .c file
*-----------------------------------------------------------------*/
#include Flash2803x_API_Library.h
...
#pragma DATA_SECTION(Flash_CPUScaleFactor, "FlashScalingVar");
Uint32 Flash_CPUScaleFactor;
...
EALLOW;
Flash_CPUScaleFactor = SCALE_FACTOR;
EDIS;
```

NOTE

The Flash_CPUScaleFactor variable is EALLOW protected.

CAUTION

It is strongly recommended that you test the CPU frequency and PLL configuration using the configuration toggle test described in section 16 before erasing or programming any parts.

If this test fails, DO NOT PROCEED to erase or program the flash until the problem is corrected, or flash damage can occur.

13. Step 8: Initialize the Callback Function Pointer

A callback function is one that is not invoked explicitly by the user's application; rather the responsibility for its invocation is delegated to the API function by way of the callback function's address. The callback function can be used whenever the application must process certain information itself at some time in the middle of the execution of an API function. For example, if the system has an external watchdog that must be serviced or if status needs to be sent by way of a communications port, this can be done by the user inserting code within the callback function.

Flash_CallbackPtr is global function pointer used to specify the callback function to be used by the Flash API. The Flash API functions will call the callback function at safe times during the program, erase, verify and depletion recovery algorithms. Refer to section 15 for rules that must be followed when using the callback function.

To use the callback function, the calling application must first initialize the function pointer Flash_CallbackPtr before calling any API function. If the callback feature is not going to be used, then set the pointer to NULL. When Flash_CallbackPtr is NULL the API will not make a call to any function.

Since the majority of the API is embedded into the ROM with hard addresses the location of the Flash_CallbackPtr variable must be at address 0x000D02. To ensure this use the following compiler pragma directive in combination with the DSP2803x_Headers_nonBIOS.cmd linker command file:

```
/*-----------------------------------------------------------------
*  User's Application with a Callback Function
*-----------------------------------------------------------------*/
#include Flash2803x_API_Library.h
...
void MyCallbackFunction(void);       // My Callback function prototype
...
#pragma DATA_SECTION(Flash_CallbackPtr, "FlashCallbackVar");
void (*Flash_CallbackPtr) (void);
...
EALLOW;
Flash_CallbackPtr = &MyCallbackFunction;
EDIS;
...
...
void myCallbackFunction(void)
{
  // User's application code to execute during the callback function
  // This must not execute from flash/OTP or read data from flash/OTP
}
```

```
/*-----------------------------------------------------------------
*  User's Application without a Callback Function
*-----------------------------------------------------------------*/
#include Flash2803x_API_Library.h
#include <stdio.h>                   // NULL is defined here
...
#pragma DATA_SECTION(Flash_CallbackPtr, "FlashCallbackVar");
void (*Flash_CallbackPtr) (void);
...
EALLOW;
Flash_CallbackPtr = NULL;
EDIS;
```

NOTE

The Flash_CallbackPtr variable is EALLOW protected.

CAUTION

If the callback function feature is not used, then explicitly set the function pointer to NULL in the application code. Failure to explicitly initialize the callback function pointer can cause the code to branch to an undefined location.

By default, the Flash_CallbackPtr is initialized to NULL as part of the device boot code. If the boot code is bypassed during emulation, and the application does not explicitly set the Flash_CallbackPtr to either NULL or a callback function, code can branch to an undefined location. For this reason it is recommended that Flash_CallbackPtr always be explicitly initialized before calling the Flash API.

During the callback function, the flash and OTP are in a safe state such that the callback function can take as much time as required to send status, service an external watchdog or perform other operations. However, flash and OTP are not available for execution of code or reading of data during this time. Refer to section 15 for rules that must be followed during the callback function.

CAUTION

During the callback function, the flash and OTP are not available for use by the application. Code cannot be executed from the flash/OTP nor can data be read from the flash/OTP. Flash and OTP will only again become available after the API function exits. Thus, the callback function must be executed completely outside of the flash/OTP and must not expect to read data from the flash/OTP.

Attempting to execute from the flash will result in unknown opcode fetch and likely result in an illegal trap (ITRAP). Data fetched will be an unknown value.

14. Step 9: Optional: Disable Global Interrupts

The Flash API saves the state of INTM and DBGM before and restores them after time critical code segments. The following two assembly routines are used to enable this save and restore feature:

```
/*-------------------------------------------------------------------
* API Save INTM and DBGM / Set INTM, DBGM to Disable Interrupts
* This code is internal to the API and is called by the API routines
*-------------------------------------------------------------------*/
_Fl28x_DisableInt:
    PUSH  ST1           ;Save the state of INTM DBGM (in ST1)
    SETC  INTM,DBGM     ;Disable interrupts
    MOV   AL, *--SP     ;Return ST1 to the calling routine
    LRETR
```

```
/*-------------------------------------------------------------------
* API Restore INTM, DBGM
* This code is internal to the API and is called by the API routines
*-------------------------------------------------------------------*/
_Fl28x_RestoreInt:
    MOV   *SP++, AL     ;Pushed the saved ST1 onto the stack
    POP   ST1           ;Restore INTM and DBGM (in ST1)
    LRETR
```

In addition to global interrupts, the XNMI interrupt is also disabled during time critical code. That is, the state of the XNMICR control register is saved and XNMI is disabled. After the time critical code, XNMICR is restored.

```
/*-------------------------------------------------------------------
* API Save/Restore XNMICR
* This code is internal to the API routines
*-------------------------------------------------------------------*/
#define XNMICR (volatile Uint16*)0x00007077   // XNMI Control
Uint16 XnmiCr;
...
XnmiCr = *XNMICR;                             // Save XNMICR
*XNMICR = *XNMICR & 0xFFFE                    // Disable XNMI
...
// Time critical API code
...
*XNMICR = XnmiCR;                             // Restore XNMICR
```

During the time that interrupts are enabled, the flash and OTP are in a safe state such that an interrupt service routine (ISR) can take as much time as required to service the interrupt. Flash and OTP are not available for execution of code or reading of data during this time. Refer to section 15 for rules that must be followed when interrupts are allowed during API function calls.

CAUTION

During the time that interrupts are enabled, the flash and OTP are not available for use by the application. Code cannot be executed from the flash/OTP nor can data be read from the flash/OTP.

Flash and OTP will only again become available after the API function exits. Any ISR routines that are executed during an API function call must completely reside outside of the flash/OTP and must not expect to read data from the flash/OTP.

Attempting to execute from the flash/OTP will result in undefined opcode fetch and likely result in an illegal trap (ITRAP). Data fetched will be an unknown value.

15. Step 10: Rules for Callback, Interrupts, and Watchdog

The following API restrictions should be understood:

- If an interrupt is taken during the erase, program, or depletion recovery routines, then the flash/OTP will not be accessible. Thus no code or data used by the interrupt can be stored in the flash or OTP.
- When the callback function is invoked during the erase, program, or depletion recovery routines, the flash/OTP will not be accessible. Thus no code or data used by the callback function can be stored in the flash or OTP.
- The flash/OTP is left in a safe state when interrupts are enabled and during the callback function. There is no restriction on the amount of time that can be spent in the ISR or callback function.
- The API functions are not reentrant.
- Only one API function should be called at any particular time. If one API function has been called but has not yet completed, do not call another API function from within the interrupt service routine or callback function. Let the API function complete and exit before calling any other API function.
- The ToggleTest API function disables interrupts and runs until the processor is halted by the user.
- The ToggleTest API function does not invoke the callback function.
- The Erase function disables interrupts for up to 3-4ms. This is the longest duration interrupts are disabled in any of any of the API functions, except for the ToggleTest function.
- The erase function will invoke the callback function every 3-4ms. This is the longest duration between callback function invocations, except for the ToggleTest function.
- The API disables the watchdog using the following code such that the watchdog counter pre-scaler is not modified:

```
/*-------------------------------------------------------------------
* API Watchdog Disable
* This code is internal to the API routines
*-------------------------------------------------------------------*/
#define WDCR (volatile Uint16*)0x00007029   // Watchdog control register
asm("   EALLOW);
*WDCR = (*WDCR | 0x0068);                   // Disable the watchdog
asm("   EDIS");
```

CAUTION

If you clear the WD_OVERIDE bit in your code so that the watchdog cannot be disabled by the API, you will run the risk of resetting the DSP during an API operation. This can leave the flash in an unknown state and possibly lock the Code Security Module with an unknown password. If you must clear the WD_OVERIDE bit, then it is recommended that you re-assign the watchdog output to its interrupt before calling an API function. This will keep the watchdog from resetting the device during this time.

16. Step 11: Optional: Frequency and PLL configuration toggle test

This test is used to confirm that the algorithms are properly configured for the CPU frequency (Refer to section 12) and PLL multiplier (Refer to section 9). During this test, a specified GPI/O pin will toggle at a known frequency. If this frequency is not correct then the API functions are not configured correctly.

This test is started by calling the API ToggleTest function documented in section 18.6. This function allows you to specify which GPI/O pin will be toggled by passing a pointer to its corresponding GPIOMUX and a pointer to its GPIOTOGGLE register. Finally you can specify exactly which pin on the specified port will be toggled by a Mask value.

CAUTION

Choose an appropriate pin for your system. Check your board design and board connections to be certain that the pin you have selected for toggling is not being driven by a source other than the DSP, or voltage contention can occur. Also, be certain that whatever the toggling pin is connected to in your system will not encounter difficulty when the pin is toggling (e.g, the device the pin is connected to should be powered-down, held in reset, etc.).

While the test runs, monitor the selected pin using an oscilloscope.

- **If the algorithms are configured correctly for your CPU rate then the pin will toggle near 10kHz (100µS +/- 10µS cycle time).**
- **If the pin is toggling at a different rate, then the algorithms are <u>not</u> configured correctly. If this is the case, review steps 1-6 in the checklist shown in section 5 to ensure the proper Flash API setup.**

Note: The toggle test runs forever and does not return. Interrupts are disabled during the frequency toggle test. The device can be halted anytime during this test to stop execution. This test is only used during development to confirm the configuration of the Flash API. If this function is not referenced in your code it will not be linked in.

17. Step 12: Optional: Unlock the Code Security Module (CSM)

The Code Security Module (CSM) protects the contents of the F2803x flash and OTP memory blocks as well as L0-L3 SARAM blocks. The Flash API functions must be able to access the flash while performing any erase, program, or verify operations. There are two possible scenarios to consider:

- The Flash API functions are executed from within memory protected by the CSM. Since the API functions are executing from within CSM protected memory they will be able to access any other secure memory location including the flash and OTP. In this case the CSM can remain locked and no action is required.

- The Flash API Functions are executed from memory not protected by the CSM. In this case, the API will not be able to access any secure memory location and thus cannot access the flash or OTP. In this case, the calling application must first unlock the CSM before making any calls to the Flash API.

Refer to *TMS320x2803x System Control and Interrupts Reference Guide* for details on the proper operation of the CSM.

18. Step 13: API Reference

18.1. Data Type Conventions

The following data type definitions are defined in Flash2803x_API_Library.h and are used within this document:

```
#ifndef DSP28_DATA_TYPES
#define DSP28_DATA_TYPES
typedef int                   int16;
typedef long                  int32;
typedef long long             int64;
typedef unsigned int          Uint16;
typedef unsigned long         Uint32;
typedef unsigned long long    Uint64;
typedef float                 float32;
typedef long double           float64;
#endif
```

18.2. API Function Naming Conventions and Function list

The F2803x API function names are of the following form:

```
Flash2803x_<operation>(args)
```

Where *<operation>* is the operation being performed such as Erase, Program, Verify

For example: `Flash2803x_Program(args)` is the F2803x Program function.

The API function definitions for the F2803x API libraries are compatible with other 28x devices. For this reason the file Flash2803x_API_Library.h includes macro definitions that allow a generic function call to be used in place of the device specific function call.

```
Flash_<operation>(args)
```

Use of these macros is optional. They have been provided to allow easy porting of code between the devices. All of the examples shown in this document use the generic function call.

TMS320F2803x API Compatibility Macros:

Generic Function	F2803x API Function
Flash_ToggleTest	Flash2803x_ToggleTest
Flash_Erase	Flash2803x_Erase
Flash_Program	Flash2803x_Program
Flash_Verify	Flash2803x_Verify
Flash_DepRecover	Flash2803x_DepRecover
Flash_APIVersion	Flash2803x_APIVersion
Flash_APIVersionHex	Flash2803x_APIVersionHex

For users porting from the F281x family of DSPs:

Except for the ToggleTest function, the F2803x API function calls are all compatible with the F281x API function calls. This makes moving from the F281x to the F2803x straight forward.

For users porting other 28x families of DSPs:

The F2803x API function calls are all compatible with the other 28x family API function calls. This makes moving from those devices to the F2803x straight forward.

18.3. Flash status structure (FLASH_ST)

This structure is used to pass information back to the calling routine by the Program, Erase and Verify API functions. This structure is defined in Flash2803x_API_Library.h:

```
typedef struct {
    Uint32  FirstFailAddr;
    Uint16  ExpectedData;
    Uint16  ActualData;
}FLASH_ST;
```

18.4. API Version (in float) Function

Description: The API Version function returns the version number of the API. .

Note: The Flash_APIVersionHex() function can be used in place of this function to avoid issues associated with processing floating point values.

Function Prototype (Defined in Flash2803x_API_Library.h)

```
TMS320F2803x:
  extern float32 Flash2803x_APIVersion(void);
```

Return Value:

- The Flash_APIVersion function returns the current version of the API as a float value. Note: The Flash_APIVersionHex function can be used in place of this function to avoid issues associated with floating point values.

Notes:

- By using the API compatibility macros provided in Flash2803x_API_Library, this function can be called as Flash_APIVersion. All of the examples shown use this generic function call. Refer to section 18.2.

Example:

```
/*-----------------------------------------------------------------
* Example: Get the version of the API as a floating point value
*-----------------------------------------------------------------*/
#include Flash2803x_API_Library.h
#define VALID_API_VERSION (float32)1.00   // Version 1.00
...
float32 ApiVerison;
ApiVersion = Flash_APIVersion();
if(ApiVersion == VALID_API_VERSION)
{
     // Code for valid API
}
else
{
     // Code for invalid API
}
```

18.5. API Version (in Hex) Function

Description: This function returns the current API version number in decimal encoded hex..

Function Prototypes (Defined in Flash2803x_API_Library.h)

```
TMS320F2803x:
   extern Uint16 Flash2803x_APIVersionHex(void);
```

Return Value:

- If the function returns the current version of the API in a 16-bit decimal encoded hex value. This function can be used in place of the Flash_APIVersion() function to avoid issues associated with floating point values. The value is divided such that the upper 8 bits are the major release and the lower 8 bits are the minor release. The version uses values 0-9, but does not use A-F.

  For example:

  If the API Version is 1.00, Flash_APIVersionHex would return 0x0100
  If the API Version is 3.10 Flash_APIVersionHex would return 0x0310

Notes:

- By using the API compatibility macros provided in Flash2803x_API_Library, this function can be called as Flash_APIVersionHex. All of the examples shown use this generic function call. Refer to section 18.2.

Example:

```
/*-----------------------------------------------------------------
* Example: Read the version of the API in Hex
*-----------------------------------------------------------------*/
#include Flash2803x_API_Library.h
#define VALID_API_VERSION 0x0100   // Version 1.00
...
Uint16 ApiVerison;
ApiVersion = Flash_APIVersionHex();
if(ApiVersion == VALID_API_VERSION)
{
     // Code for valid API
}
else
{
     // Code for invalid API
}
```

18.6. ToggleTest Function

Description: The **ToggleTest** function toggles a specified GPIO port pin at a 10 kHz rate. This test can be run to test the frequency configuration of the flash API. If the toggle rate of the specified I/O pin is not correct then the API is not configured properly.

Note that due to different device requirements, this function has changed from the F281x API. If you are porting from the F281x, make sure to setup the GPIO prior to the call to the ToggleTest function and to change the function call appropriately.

IMPORTANT: Before calling this function you must first configure the GPIO pin you wish to use as a GPIO output pin by writing to the appropriate GPxMUX1/2 register and the GPxDIR register.

Function Prototype (Defined in Flash2803x_API_Library.h)

```
TMS320F2803x:
  extern void Flash2803x_ToggleTest(
     volatile Uint32 *ToggleReg, // Pointer to I/O port TOGGLE register
     Uint32 Mask                 // Pin Mask
  );
```

Parameter:	Description:
`volatile Uint32 *ToggleReg`	Pointer to the desired 32-bit GPxTOGGLE Register. This would be the address of the GPxTOGGLE register.
`Uint32 Mask`	Mask value specifying which pin to toggle on the specified I/O port. If the bit is set, the pin will be toggled, if it is clear then the pin will not be toggled. For example: Toggle Pin GPIO0: Use Mask: 0x00000001 Toggle Pin GPIO1: Use Mask: 0x00000002 etc..

Return Values: None. This function runs “forever” and never returns.

Notes:

- **BEFORE** calling this function you must first configure the GPIO pin you wish to use as a GPIO output pin by writing to the appropriate GPxMUX1/2 register and the GPxDIR register. This function assumes that the pin is already configured as a GPIO output.
- By using the F2803x API compatibility macros provided in Flash2803x_API_Library, this function could be called as Flash_ToggleTest. All of the examples shown use this generic function call. Refer to section 18.2.
- Choose an appropriate pin for your system. Check your board design and board connections to be certain that the pin you have selected for toggling is not being driven by a source other than the DSP, or voltage contention can occur. Also, be certain that whatever the toggling pin is connected to in your system will not encounter difficulty when the pin is toggling (e.g, the device the pin is connected to should be powered-down, held in reset, etc.).

Example: Toggle Test on GPIO32

```
/*-----------------------------------------------------------------
* User's application .c code:  Example Toggle GPIO32
*-----------------------------------------------------------------*/
#include Flash2803x_API_Library.h
...
#pragma DATA_SECTION(Flash_CPUScaleFactor, "FlashScalingVar");
Uint32 Flash_CPUScaleFactor;

#pragma DATA_SECTION(Flash_CallbackPtr, "FlashCallbackVar");
void (*Flash_CallbackPtr) (void);
...
#define GPBMUX1          (volatile Uint32*)0x00006F98   // GPIO B mux 1
#define GPBTOGGLE        (volatile Uint32*)0x00006FCE   // GPIO B toggle
#define GPBDIR           (volatile Uint32*)0x00006F9A   // GPIO B dir
#define GPIO32_MASK      (Uint32)0x00000001             // GPIO32 mask
...
Flash_CPUScaleFactor = SCALE_FACTOR;
Flash_CallbackPtr = NULL;
...
// Code to set PLLCR and wait for PLL lock
...
// BEFORE calling the toggle test, configure the desired pin to be a GPIO
// output pin. For this example, configure GPIO32 as a GPIO output

   EALLOW;                          // Macro for asm("    EALLOW);
   *GPBMUX1 = 0x00000000;           // All GPIO B pins are GPIO
   *GPBDIR = 0x00000001;            // All GPIO B pins are outputs
   EDIS;

// Call the Toggle Test function.  This function will run forever
// and will not exit.

   Flash_ToggleTest(GPBTOGGLE,GPIO32_MASK);
...
```

18.7. Erase Function

Description: The Erase function will erase the specified flash sectors. The remaining sectors will not be changed.

Function Prototypes (Defined in Flash2803x_API_Library.h)

```
TMS320F2803x:
   extern Uint16 Flash2803x_Erase(
        Uint16 SectorMask,          // Sector mask
        FLASH_ST *FEraseStat        // Pointer to the status structure
   );
```

Parameter: `Uint16 SectorMask`

Description:
Sector Mask value: Set bits indicate which sectors will be erased.

Bit	F2803x
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
14:8	ignored

Parameter: `FLASH_ST *FEraseStat`

Pointer to a flash status structure.
This structure is defined in Flash2803x_API_Library.h:

```
typedef struct {
    Uint32  FirstFailAddr;
    Uint16  ExpectedData;
    Uint16  ActualData;
}FLASH_ST;
```

Note: For erase, only the FirstFailAddr of this structure is currently used.

Return Value:

- If the function succeeds STATUS_SUCCESS is returned.
- If the function fails a status value indicating the reason for the failure is returned.

Notes:

- By using the F2803x API compatibility macros provided in Flash2803x_API_Library, this function can be called as Flash_Erase. All of the examples shown use this generic function call. Refer to section 18.2.
- After a sector is erased, all of its memory locations will read back 0xFFFF.
- The Erase function includes a preconditioning (“clear”) step as well as a post-conditioning (“compaction”) of the specified sectors. A separate step to precondition the flash is not required.
- On the F2803x the specified sectors are erased in order Sector H – Sector A.
- The minimum amount of flash memory that can be erased on a F2803x device is a single sector. A word or bit cannot be erased by itself.
- The OTP block cannot be erased.
- The following sector mask #defines are included in Flash2803x_API_Library.h

```
/*-------------------------------------------------------------------
* Flash2803x_API_Library.h
*-------------------------------------------------------------------*/
#define SECTORA   (Uint16)0x0001
#define SECTORB   (Uint16)0x0002
#define SECTORC   (Uint16)0x0004
#define SECTORD   (Uint16)0x0008
#define SECTORE   (Uint16)0x0010
#define SECTORF   (Uint16)0x0020
#define SECTORG   (Uint16)0x0040
#define SECTORH   (Uint16)0x0080


// All sectors on an F2803x - Sectors A - H
#define SECTOR_F2803x
(SECTORA|SECTORB|SECTORC|SECTORD|SECTORE|SECTORF|SECTORG|SECTORH)
```

Erase function continued…

Examples:

```
/*-----------------------------------------------------------------
* User's Application
* Example: Erase Sector D then erase sector C and B
*-----------------------------------------------------------------*/
#include "Flash2803x_API_Library.h"
...
Uint16 Status;
FLASH_ST EraseStatus;

#pragma DATA_SECTION(Flash_CPUScaleFactor, "FlashScalingVar");
Uint32 Flash_CPUScaleFactor;

#pragma DATA_SECTION(Flash_CallbackPtr, "FlashCallbackVar");
void (*Flash_CallbackPtr) (void);
...

Flash_CPUScaleFactor = SCALE_FACTOR;
Flash_CallbackPtr = NULL;
...

// Code to set PLLCR and wait for PLL lock
...

// Erase Sector D
// Following is defined in Flash2803x_API_Library.h
// #define SECTORD   (Uint16)0x0008

// User's Code:
Status = Flash_Erase(SECTORD,&EraseStatus);
if(Status != STATUS_SUCCESS) Error(Status);
...

// Erase Sector C and Sector B
// Following is defined in Flash2803x_API_Library.h
// #define SECTORC   (Uint16)0x0004
// #define SECTORB   (Uint16)0x0002

// User's Code:
Status = Flash_Erase((SECTORC|SECTORB),&EraseStatus);
if(Status != STATUS_SUCCESS) Error(Status);

...
```

18.8. Program Function

Description: The program function will program a buffer of 16-bit values into the flash or OTP.

Function Prototypes (Defined in Flash2803x_API_Library.h)

```
TMS320F2803x:
   extern Uint16 Flash2803x_Program(
         Uint16 *FlashAddr,           // Pointer to the first flash/OTP loc
         Uint16 *BufAddr,             // Pointer to the buffer
         Uint32 Length,               // Number of 16-bit values to program
         FLASH_ST *FProgStatus        // Pointer to the status structure
   );
```

Parameter:	**Description:**
`Uint16 *FlashAddr`	Pointer to the first 16-bit location in flash or OTP to be programmed.
`Uint16 *BufAddr`	Pointer to the buffer of 16-bit data or code to be programmed into flash or OTP.
`Uint32 Length`	Number of 16-bit values to be programmed into the flash or OTP

`FLASH_ST *FProgStatus`	Pointer to a flash status structure.
	This structure is defined in Flash2803x_API_Library.h:
	`typedef struct {`
	`  Uint32  FirstFailAddr;`
	`  Uint16  ExpectedData;`
	`  Uint16  ActualData;`
	`}FLASH_ST;`

Return Value:

- If the function succeeds STATUS_SUCCESS is returned.
- If the function fails a status value indicating the reason for the failure is returned.

Notes:

- By using the F2803x API compatibility macros provided in Flash2803x_API_Library, this function can be called as Flash_Program. All of the examples shown use this generic function call.Refer to section 18.2.
- Program operates on a 16-bit word at a time until all the data in the buffer is programmed or an error is detected.
- Program moves bits from a value of 1 to a value of 0 in order to match the data to be programmed.
- Typically a sector will be erased prior to being programming. However, to protect the flash and allow for user flexibility, the Program operation will not attempt to program any bit that has previously been programmed. For example, a location can be programmed with 0xFFFE and later the same location can be programmed with 0xFFFC without going through an erase cycle. During the second programming call, the Program operation will detect that bit 0 was already programmed and will only program bit 1.
- If the data to be programmed has a 1 in any bit that has previously been programmed the function will stop and return STATUS_FAIL_ZERO_BIT_ERROR. For example, if you program a location with 0x0001 and then try to program the same location with 0x0002 the function will return this failure. This is because no single bit can be erased (i.e. moved from a 0 to a 1). If this happens, the function will not attempt to program any other bits.

Example:

```
/*-------------------------------------------------------------------
* Example: Program 0x400 values into the flash starting at 0x3F0000
*-------------------------------------------------------------------*/
#include Flash2803x_API_Library.h
#define  WORDS_IN_FLASH_BUFFER 0x400
...

volatile Uint16  Buffer[WORDS_IN_FLASH_BUFFER];
Uint16 *Flash_ptr;     // Pointer to a location in flash
Uint32 Length;         // Number of 16-bit values to be programmed
FLASH_ST ProgStatus;   // Status structure
Uint16 Status;         // Return status

#pragma DATA_SECTION(Flash_CPUScaleFactor, "FlashScalingVar");
Uint32 Flash_CPUScaleFactor;

#pragma DATA_SECTION(Flash_CallbackPtr, "FlashCallbackVar");
void (*Flash_CallbackPtr) (void);
...

Flash_CPUScaleFactor = SCALE_FACTOR;
Flash_CallbackPtr = NULL;
...

// Code to set PLLCR and wait for PLL lock
...

// Fill the buffer with some data to program into the flash
for(i=0; i<0x400; i++) Buffer[i] = 0x8000+i;
...

Flash_ptr = (Uint16 *)0x003F0000;
Length = 0x400;
...

// Call the program API function
Status = Flash_Program(Flash_ptr,Buffer,Length,&ProgStatus);
if(Status != STATUS_SUCCESS) Error(Status);
```

18.9. Verify Function

Description: Verify the contents of flash or OTP against a buffer. While the program operation itself does verification as it programs this verification is an additional step that can be taken after programming is complete.

Function Prototypes (Defined in Flash2803x_API_Library.h)

```
TMS320F2803x:
   extern Uint16 Flash2803x_Verify(
        Uint16 *FlashAddr,            // Pointer to the first flash/OTP loc
        Uint16 *BufAddr,              // Pointer to the buffer
        Uint32 Length,                // Number of 16-bit values to verify
        FLASH_ST *FVerifyStat         // Pointers to the status structure
);
```

Parameter:	**Description:**
Uint16 *FlashAddr	Pointer to the first location in flash or OTP to be programmed.
Uint16 *BufAddr	Pointer to the buffer of data or code to be programmed into flash or OTP.
Uint32 Length	Number of 16-bit values to be programmed into the flash or OTP
FLASH_ST *FProgStatus	Pointer to a flash status structure. This structure is defined in Flash2803x_API_Library.h:

```
typedef struct {
    Uint32  FirstFailAddr;
    Uint16  ExpectedData;
    Uint16  ActualData;
}FLASH_ST;
```

Return Value:

- If the function succeeds STATUS_SUCCESS is returned.
- If the function fails a status value indicating the reason for the failure is returned.

Notes:

- By using the F2803x API compatibility macros provided in Flash2803x_API_Library, this function can be called as Flash_Verify. The examples use this generic function call. Refer to section 18.2.

```
/*-----------------------------------------------------------------
* Example: Verify 0x400 values in the flash starting at 0x3F0000
*-----------------------------------------------------------------*/
#include Flash2803x_API_Library.h
#define  WORDS_IN_FLASH_BUFFER 0x400
...
volatile Uint16  Buffer[WORDS_IN_FLASH_BUFFER];
Uint16 *Flash_ptr;     // Pointer to a location in flash
Uint32 Length;         // Number of 16-bit values to be programmed
FLASH_ST VerifyStatus; // Status structure
Uint16 Status;         // Return status

#pragma DATA_SECTION(Flash_CPUScaleFactor, "FlashScalingVar");
Uint32 Flash_CPUScaleFactor;

#pragma DATA_SECTION(Flash_CallbackPtr, "FlashCallbackVar");
void (*Flash_CallbackPtr) (void);
...
Flash_CPUScaleFactor = SCALE_FACTOR;
Flash_CallbackPtr = NULL;
...
// Code to set PLLCR and wait for PLL lock
...
// Fill the buffer with some data to verify against
for(i=0; i<0x400; i++) Buffer[i] = 0x8000+i;
...
Flash_ptr = (Uint16 *)0x003F0000;
Length = 0x400;
...
// Call the verify API function
Status = Flash_Verify(Flash_ptr,Buffer,Length,&VerifyStatus);
if(Status != STATUS_SUCCESS) Error(Status);
```

18.10. Depletion Recovery Function

Description: The depletion recovery algo looks for sectors that are in depletion and attempts to recover them. All sectors on the device are checked

Function Prototypes (Defined in Flash2803x_API_Library.h)

```
TMS320F2803x:
   extern Uint16 Flash2803x_DepRecover(void);
```

Return Value:

- If the function succeeds STATUS_SUCCESS is returned.
- If the function fails STATUS_FAIL_PRECONDITION is returned.

Notes:

How does depletion occur?

If the erase operation is halted and not allowed to complete, bits in the sector may be left in an over erased or depleted state. When this happens, the device may then begin to fail to erase. The depletion recovery algo looks for sectors that are in depletion and attempts to recover them. All sectors on the device are checked.

To avoid depletion, all efforts should be taken to not stop the erase algorithm before it completes. In addition to depletion, halting erase can also affect the CSM passwords. If the passwords are left in an unknown state then the device cannot be erased, programmed or recovered as the flash cannot be accessed unless the API function can be executed within CSM protected SARAM memory. If, however, the CSM passwords are known and the device can be unlocked, then the depletion recovery algorithm can be run to try and recover the part.

The current maximum timeout for the algorithm is approximately 35 seconds for each sector that is in depletion. Typically only one sector would be in depletion unless erase has been called multiple times on multiple sectors without running to completion. If a longer timeout can be tolerated, the depletion recovery can be used multiple times.

There is no guarantee that this algorithm will be able to bring a sector out of depletion within a reasonable amount of time. The deeper in depletion the part is, the longer it will take to recover. The Flash API erase function has been implemented to erase the flash in such a manner that it is not put into deep depletion. However, if the CPU is halted during an erase pulse for a long period of time the part can be put into a deep depletion that may not be recoverable in a time period that is acceptable.

This algorithm cannot recover the part if the flash passwords are unknown. For example if power is lost during the erase of sector A, where the CSM passwords are located, then the device may be permanently locked and the recovery algorithm cannot operate on the flash.

18.11. Step 14: Return Status Values

To communicate back to the calling application, the API returns the following status messages. These status values are defined in the Flash2803x_API_Library.h file for use within your application.

Status	Definition	Notes
0	STATUS_SUCCESS	Operation was successful.
10	STATUS_FAIL_CSM_LOCKED	The API function is unable to access the flash array due to a locked Code Security Module. The API must either execute from secure SARAM memory (L0-L3) or the CSM must be unlocked before calling the API function. Refer to section 17.
11	STATUS_FAIL_REVID_INVALID	Reserved – Not used in the F2803x API This error was used in early F281x APIs but is **not** used in the F2803x API.
12	STATUS_FAIL_ADDR_INVALID	An invalid address (outside of the flash or OTP bank) was passed to the Flash API. This could be caused by the first address being outside of flash/User OTP or the length being such that the last address will be outside of flash/User OTP. None of the values will be programmed. In this case the flash status structure (FLASH_ST) is not updated with any information. This error can be returned by the Erase and depletion recovery functions if an invalid address is used for the pre-conditioning of the flash. In this case check that the .const section of the API is located in SARAM that can be accessed by the API. This section contains important information that is used by the flash Erase and DepRecovery functions.
13	STATUS_FAIL_INCORRECT_PARTID	Reserved – Not used in the F2803x API

14	STATUS_FAIL_API_SILICON_MISMATCH	Reserved – Not used in the F2803x API
Erase Specific Status Messages		
20	STATUS_FAIL_NO_SECTOR_SPECIFIED	Erase had nothing to do because no valid sectors were specified.
21	STATUS_FAIL_PRECONDITION	Erase operation failed because the pre-condition (i.e. program all 0's) operation failed.
22	STATUS_FAIL_ERASE	Erase operation failed because the sector could not be erased with the maximum allowed number of pulses.
23	STATUS_FAIL_COMPACT	Erase operation failed because the post-conditioning (i.e. compaction) failed.
24	STATUS_FAIL_PRECOMPACT	Erase operation failed because the pre- compaction portion failed. The pre-compaction is applied to all sectors on the device. The FLASH_ST structure will return a fail address corresponding to the first sector fails this step.
Program Specific Status Messages		
30	STATUS_FAIL_PROGRAM	Program operation failed because one or more bits could not be programmed.
31	STATUS_FAIL_ZERO_BIT_ERROR	Program operation failed because one or more bits were already programmed (read back 0) that should have been erased (read back 1). If this happens it could be because the sector was not erased before attempting to program.
Verify Specific Status Messages		
40	STATUS_FAIL_VERIFY	The verify operation failed because one or more bits did not match the reference data. Try increasing the flash or OTP wait states.

19. F2803x API Revision Information

V1.00: Initial Release. May 2009.

20. Files included with the API

Each device has an API library. Include the proper library for your device in your project.

Since the API function call definitions are compatible between the F2803x devices, the include files and documentation is the same for all of the F2803x devices. They have been duplicated under both sub-directories for convenience and clarity.

In a typical install, `<base> = c:\tidcs\c28\Flash28_API`

TMS320F2803x

Boot ROM Symbol API Library:

`<>\Flash2803x_API_V100\lib\2803x_FlashAPI_BootROMSymbols.lib`

API Include Files:

`<>\Flash2803x_API_V100\include\Flash2803x_API_Library.h`
`<>\Flash2803x_API_V100\include\Flash2803x_API_Config.h`

Documentation:

`<>\Flash2803x_API_V100\doc`

Example:

`<>\Flash2803x_API_V100\example`