TMX320F281x Flash API

Version 1.00

December 10, 2003
Flash API Disclaimer

The following Flash Application Program Interface (Flash API) libraries are included in this release:

- Flash2810_API_V100.lib
- Flash2811_API_V100.lib and
- Flash2812_API_V100.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 TMS320F281x 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 F281x silicon.

Should functional changes occur to the API’s, 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.

Updates to the API will be posted on the Texas Instruments Inc website (www.ti.com) and can also be obtained by contacting Texas Instruments’ support.
Contents:

1. Release Notes ............................................................................................................................... 4
2. Revision Identification ................................................................................................................... 5
3. Introduction: Flash API Programming Fundamentals .............................................................. 6
4. Example Program ........................................................................................................................... 6
5. Flash API Checklist ....................................................................................................................... 7
6. Step 1: Modify Flash281x_API_Config.h .............................................................................. 8
7. Step 2: Include Flash281x_API_Library.h ............................................................................. 9
8. Step 3: Include the proper Flash API library .............................................................................. 9
9. Step 4: Initialize PLL Control Register (PLLCR) ..................................................................... 10
10. Step 5: Copy the Flash API functions to Internal SARAM .................................................. 10
11. Step 6: Initialize Flash_CPUScaleFactor ............................................................................... 13
12. Step 7: Optional: Frequency and PLL configuration toggle test .......................................... 14
13. Step 8: Optional: Unlock the Code Security Module (CSM) ................................................ 14
14. Step 9: API Reference ............................................................................................................. 15
14.1. Data Type Conventions ........................................................................................................... 15
14.2. API Function Naming Conventions and Function list ........................................................ 15
14.3. Flash status structure (FLASH_ST) ...................................................................................... 15
14.4. ToggleTest Function ............................................................................................................. 16
14.5. Erase Function ....................................................................................................................... 18
14.6. Program Function .................................................................................................................. 21
14.7. Verify Function ...................................................................................................................... 23
14.8. Step 10: Return Status Values ............................................................................................. 25
14.9. Files included in this release ............................................................................................... 26
1. Release Notes

1. The F281x Flash APIs have all 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.

2. For information on the large memory model refer to the TMS320C28x Optimizing C/C++ Compiler User’s Guide (literature #SPRU514).

3. 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”.

4. Note: The CSM will be permanently 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.

5. 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 and SPI modes, to get the flash programming code into the DSP.

6. For programming the F281x through the JTAG port, use the SDFlash programmer from TI 3rd Party vendor Spectrum Digital Inc. (www.spectrumdigital.com). A Code Composer Studio™ Plug-in is scheduled for release by the end of 2003. Check the TI website for updates.

Changes since last revision (Beta1)

1. Added support for the F2811. This device is identical to the F2812 device from a Flash memory standpoint. A separate Flash API library has been included simply for clarity.

2. Silicon revision checking: The beta1 API checks the device revision register and only works if the revision matches Rev C of the F281x devices. This was done in case changes were required based on characterization and testing of the APIs. Now that characterization is complete, this revision of the Flash API only checks that the revision is Rev C or later. Users should migrate to the new APIs before future releases of the silicon occur.

3. Verify Function: In the previous version (Beta1), verify would return STATUS_BUSY upon failure. Corrected the problem and now verify returns STATUS_VERIFY_FAIL upon failure.

4. ToggleTest Function: In the previous version (Beta1), the ToggleTest function would take over the entire port specified. That is the entire port that the specified pin belonged to was made GP outputs. This has been corrected and now the ToggleTest will only change the mode for the pin specified by the user. The remaining pins will remain as they were configured prior to the test.

5. Example program: Example program was changed to illustrate different error messages as well as correct usage. In addition the example no longer erases the entire flash by default. Thus the example can be executed more than one time without having to re-flash the part.

6. Filename changes: The following API file name changes were made. This was done because the files are the same across the F281x API’s and not device specific.

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash2812_API_Config.h</td>
<td>Flash281x_API_Config.h</td>
</tr>
<tr>
<td>Flash2812_API_Library.h</td>
<td>Flash281x_API_Library.h</td>
</tr>
</tbody>
</table>
2. Revision Identification

These flash API algorithms will function only on specific revisions of the F281x devices.

The silicon revision can be determined by the lot trace code marked on the top of the package. The locations of the lot trace codes for the F2812/F2811 GHH, PGF packages and the F2810 PBK package are shown below. Also refer to the "TMS320F2810, TMS320F2811, and TMS320F2812 Digital Signal Processors Silicon Errata," literature #SPRZ193 for more information on determining your device’s silicon revision.

<table>
<thead>
<tr>
<th>Second Letter in Prefix of Lot Trace Code</th>
<th>Silicon Revision</th>
<th>REVID (Addr 0x883)</th>
<th>Tested Flash API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank (no second letter in Prefix)</td>
<td>Revision 0</td>
<td>0x0000</td>
<td>Not Supported</td>
</tr>
<tr>
<td>A</td>
<td>Revision A</td>
<td>0x0001</td>
<td>Not Supported</td>
</tr>
<tr>
<td>C Only</td>
<td>Revision C Only</td>
<td>0x0003 Only</td>
<td>F2812: F2812_API_Beta1.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F2810: F2810_API_Beta1.lib</td>
</tr>
<tr>
<td>C</td>
<td>Revision C</td>
<td>0x0003</td>
<td>F2812: F2812_API_V100.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F2811: F2811_API_V100.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F2810: F2810_API_V100.lib</td>
</tr>
<tr>
<td>Future</td>
<td>&gt; 0x0003</td>
<td>V1.00 or API update if required.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. Revision 0, A and Revision B silicon were pre-production devices and no longer shipping. Revision B was a TI internal test device.

2. The F2810 and F2812 Beta1 APIs are locked to Rev C only. Users should migrate to V100 before the release of any future silicon revision.

3. For future silicon revisions, TI anticipates that no functional changes will be required to these API's (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 API's 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 F281x devices 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 smallest amount of memory that can be erased at a time is a single sector. The Flash array and OTP block are in an erased state when the device is shipped from the factory. Some traditional algorithms, such as those for the 240x family, require that the flash be pre-conditioned or “cleared” before it is erased. The F281x Flash API erase function includes the flash pre-conditioning and a separate “clear” step is not required.

**Program:** The program function operates on both the Flash array and the OTP block. This function is used to put user’s 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 before calling the programming function. The programming function operates on a single 16-bit word at a time.

**Verify:** The erase and program functions perform a verification as they execute. The Verify function provides a second check 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 F281x 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-waitstate internal SARAM memory.
- Configure the API’s for the correct CPU frequency of operation.

**API Don’ts:**
- Do not expect interrupts to be serviced while the Flash API code is executing.
- Interrupt the Flash API Erase, Program or Verify functions while they are executing (ex stop the debugger within API code).
- Do not execute code or fetch data from the Flash array or OTP while the Flash and/or OTP is being erased, programmed or verified.

4. Example Program

An example program that uses the Flash API has been included in this release. This example demonstrates how to interface to the API. The example is setup to be stored in the flash and the appropriate code and constants are copied to SARAM for execution.
5. Flash API Checklist

Integration of the F281x Flash APIs 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 Flash APIs. These steps are further discussed in detail in the reference section indicated.

This checklist applies to the F2810 Flash API, F2811 Flash API, and the F2812 Flash API.

Before using the API, do the following:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Modify Flash281x_API_Config.h for your target operating conditions.</td>
<td>Section 6</td>
</tr>
<tr>
<td>2</td>
<td>Include Flash281x_API_Library.h in your source code.</td>
<td>Section 7</td>
</tr>
<tr>
<td>3</td>
<td>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)</td>
<td>Section 8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Initialize the PLL control register (PLLCR).</td>
<td>Section 9</td>
</tr>
<tr>
<td>5</td>
<td>Modify the linker command file (.cmd) to handle sections of the Flash API. Use these sections to copy the Flash API functions and any other needed code/data to internal SARAM (as opposed to running them from Flash or external memory).</td>
<td>Section 10</td>
</tr>
<tr>
<td>6</td>
<td>Initialize the 32-bit global variable Flash_CPUScaleFactor</td>
<td>Section 11</td>
</tr>
<tr>
<td>7</td>
<td>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.</td>
<td>Section 12</td>
</tr>
<tr>
<td>8</td>
<td>Optional: Unlock the code security module (CSM).</td>
<td>Section 13</td>
</tr>
<tr>
<td>9</td>
<td>Call the Flash API Functions.</td>
<td>Section 14</td>
</tr>
</tbody>
</table>

The called flash API function will do the following:

- Disables global interrupts.
- Disables the watchdog timer. Because of this requirement, the user must not clear the WD_OVERIDE bit before calling the API. If the watchdog fires during a flash operation, the flash can be left in an undetermined state.
- Checks the REVID register. It is important that the Flash API is used only for the device revision for which it was developed. If the REVID register does not match Rev C silicon then the function will exit.
- Performs the called operation and return status.

The user’s code should then do the following:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Check the return status against the error codes.</td>
<td>Section 14.8</td>
</tr>
<tr>
<td>11</td>
<td>Optional: Re-enable global interrupts.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Optional: Re-enable the watchdog timer.</td>
<td></td>
</tr>
</tbody>
</table>
6. Step 1: Modify Flash281x_API_Config.h

The Flash281x_API_Config.h file, found in the include directory, contains configuration setup definitions. Modify this file to match your specific target operating conditions. This file is used for the F2810, F2811 and F2812 APIs.

6.1. Specify the device.

This definition is used by the API main include file, Flash281x_API_Library.h, to conditionally compile in options specific to the F2810, F2811 and F2812 devices. These options include sector bit masks and macros that interface to the API functions.

6.2. 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 135 MHz. This corresponds to a 7.407 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:

```c
#define CPU_RATE 6.667L   // for a 150MHz CPU clock speed (SYSCLKOUT)
#define CPU_RATE 7.143L   // for a 140MHz CPU clock speed (SYSCLKOUT)
#define CPU_RATE 7.407L   // for a 135MHz CPU clock speed (SYSCLKOUT)
#define CPU_RATE 8.333L   // for a 120MHz 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 Flash281x_API_Config.h, file for this calculation is:

```c
#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 30 MHz program the device at 150 MHz rather than 15 MHz or 30 MHz.

The flash API is not designed to function properly below 10 MHz.
7. Step 2: Include Flash281x_API_Library.h

Flash281x_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.

This file, Flash281x_API_Library.h, is used for interfacing to the F2810, F2811 and F2812 API libraries.

This file contains the following:

- Return status definitions. Refer to section 14.8.
- Sector bit mask definitions that can be used when calling the erase function. Refer to Section 14.5
- Flash status structure (FLASH_ST) definition used by the API functions to return information back to the calling routine.
- F2810, F2811 and F2812 API function prototypes. Refer to section 14.
- Frequency scale factor definition: Flash_CPScaleFactor. Refer to section 11.
- Macros to allow easy porting between the F2810, F2811 and F2812 libraries. Refer to section 14.2.

8. Step 3: Include the proper Flash API library

The proper Flash API library must also be included in your project.

- F2810: \Flash2810_API_V100\lib\Flash2810_API_V100.lib
- F2811: \Flash2811_API_V100\lib\Flash2811_API_V100.lib
- F2812: \Flash2812_API_V100\lib\Flash2812_API_V100.lib

The F281x 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.

As part of this initialization, the calling application must guarantee through a software delay or other means that the PLL has had enough time to lock at the new frequency before making API calls. The PLL lock time required, as of Rev C silicon is 131072 cycles.
10. Step 5: 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 the F281x 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 memory. The figure below illustrates three different methods that can be used to load the API code into the device.

Method A: The 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 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.

Method C: The 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 API functions are loaded directly into on-chip, zero-waitstate SARAM as shown in method A or B then this step can be skipped. If, however, the Flash API functions are 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:

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

F2812 API Library Group Section Example:

```c
SECTIONS
{
  Flash28_API:
  {
    Flash2812_API_V100.lib(.econst)
    Flash2812_API_V100.lib(.text)
    LOAD = FLASHC,
    RUN = RAML0,
    LOAD_START(_Flash28_API_LoadStart),
    LOAD_END(_Flash28_API_LoadEnd),
    RUN_START(_Flash28_API_RunStart),
    PAGE = 0
  ...
  }
}
```

F2811 API Library Group Section Example:

```c
SECTIONS
{
  Flash28_API:
  {
    Flash2811_API_V100.lib(.econst)
    Flash2811_API_V100.lib(.text)
    LOAD = FLASHC,
    RUN = RAML0,
    LOAD_START(_Flash28_API_LoadStart),
    LOAD_END(_Flash28_API_LoadEnd),
    RUN_START(_Flash28_API_RunStart),
    PAGE = 0
  ...
  }
}
```

F2810 API Library Group Section Example:

```c
SECTIONS
{
  Flash28_API:
  {
    Flash2810_API_V100.lib(.econst)
    Flash2810_API_V100.lib(.text)
    LOAD = FLASHC,
    RUN = RAML0,
    LOAD_START(_Flash28_API_LoadStart),
    LOAD_END(_Flash28_API_LoadEnd),
    RUN_START(_Flash28_API_RunStart),
    PAGE = 0
  ...
  }
}
```
10.2. This group section must contain the following blocks:

- API library source code: Example: Flash2810_API_V100.lib(.text)
- API constant parameters: Example: Flash2810_API_V100.lib(.econst).

It is important to make sure to include both the constants (.econst) as well as the code (.text) sections of the library.

10.3. 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: Flash28_API_LoadStart
- Load address end: Flash28_API_LoadEnd
- Run address start: Flash28_API_RunStart

These symbols are already declared in the main library include file, Flash281x_API_Library.h for use within your application.

```c
extern Uint16 Flash28_API_LoadStart;
extern Uint16 Flash28_API_LoadEnd;
extern Uint16 Flash28_API_RunStart;
```

10.4. 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 F2812 program.

Example Call to the Memory Copy Routine:

```c
Example_MemCopy(&Flash28_API_LoadStart, &Flash28_API_LoadEnd, 
&Flash28_API_RunStart);
```

```c
/------------------------------------------------------------------
Example memory copy routine to move code out of flash into SARAM
------------------------------------------------------------------*/

void Example_MemCopy(Uint16 *SourceAddr, Uint16* SourceEndAddr, Uint16* DestAddr)
{
    while(SourceAddr < SourceEndAddr)
    {
        *DestAddr++ = *SourceAddr++;
    }
    return;
}
```

10.5. This same method and copy routine can be used to copy any additional code and data that is needed during the programming operation.
11. Step 6: 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 Flash281x_API_Config.h. This step is described in section 6.2.

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

```
#define SCALE_FACTOR  1048576.0L* ( (200L/CPU_RATE) )
```

This formula is already defined in the file Flash281x_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:

```c
extern Uint32 Flash_CPUScaleFactor;   // In Flash281x_API_Library.h
Flash_CPUScaleFactor = SCALE_FACTOR;
```

CAUTION

It is strongly recommended that you test the CPU frequency and PLL configuration using the configuration toggle test described in section 12 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.
12. **Step 7: 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 11) and PLL multiplier (Refer to section 9). During this test, a specified GPIO 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 14.4. This function allows you to specify which GPIO 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.

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 not 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. 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.

13. **Step 8: Optional: Unlock the Code Security Module (CSM)**

The Code Security Module (CSM) protects the contents of the F281x Flash and OTP memory blocks as well as L0/L1 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 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 *TMS320F28x System Control and Interrupts Peripheral Reference Guide*, literature #SPRU078, for details on the proper operation of the CSM.
14. **Step 9: API Reference**

14.1. **Data Type Conventions**

The following data type definitions are defined in Flash281x_API_Library.h and are used within this document:

```c
#ifndef DSP28_DATA_TYPES
#define DSP28_DATA_TYPES

typedef int int16;
typedef long int32;
typedef unsigned int Uint16;
typedef unsigned long Uint32;
#endif
```

These data types are also used in the example code for *C28x Peripheral Examples in C* (literature number SPRC097) V.58. If both of these files are used in your project you will receive a warning regarding typedef duplication. This warning can be ignored and the `#ifndef`/#define statements were added in V1.00 of SPRC097 to avoid this warning.

14.2. **API Function Naming Conventions and Function List**

The F281x API function names are of the following form:

```c
Flash_<device>_<_operation>(args)
```

Where  
- `<device>` is either 2810, 2811, or 2812.
- `<operation>` is the operation being performed such as Erase, Program, Verify

For example: `Flash2812_Program(args)` is the F2812 Program function.

The API function definitions for the F281x API libraries are compatible. For this reason the file Flash281x_API_Library.h includes macro definitions that allow a generic function call to be used in place of the device specific function call.

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

Use of these macros is optional. They have been provided to allow easy porting of code between the F2810 and F2812 devices. All of the examples shown in this document use the generic function call.

<table>
<thead>
<tr>
<th>Generic Function</th>
<th>F2810 API Function</th>
<th>F2811 API Function</th>
<th>F2812 API Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash_ToggleTest</td>
<td>Flash2810_ToggleTest</td>
<td>Flash2811_ToggleTest</td>
<td>Flash2812_ToggleTest</td>
</tr>
<tr>
<td>Flash_Erase</td>
<td>Flash2810_Erase</td>
<td>Flash2811_Erase</td>
<td>Flash2812_Erase</td>
</tr>
<tr>
<td>Flash_Program</td>
<td>Flash2810_Program</td>
<td>Flash2811_Program</td>
<td>Flash2812_Program</td>
</tr>
<tr>
<td>Flash_Verify</td>
<td>Flash2810_Verify</td>
<td>Flash2811_Verify</td>
<td>Flash2812_Verify</td>
</tr>
</tbody>
</table>

14.3. **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 Flash281x_API_Library.h:

```c
typedef struct {
    Uint32 FirstFailAddr;
    Uint16 ExpectedData;
    Uint16 ActualData;
}FLASH_ST;
```
14.4. ToggleTest Function

Description: The ToggleTest function toggles a specified GPIO 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.

Function Prototype (Defined in Flash281x_API_Library.h)

F2810:
```
extern void Flash2810_ToggleTest(
    volatile Uint16 *MuxReg,    // Pointer to I/O port MUX register
    volatile Uint16 *ToggleReg, // Pointer to I/O port TOGGLE register
    Uint16 Mask                 // Pin Mask
);
```

F2811:
```
extern void Flash2811_ToggleTest(
    volatile Uint16 *MuxReg,    // Pointer to I/O port MUX register
    volatile Uint16 *ToggleReg, // Pointer to I/O port TOGGLE register
    Uint16 Mask                 // Pin Mask
);
```

F2812:
```
extern void Flash2812_ToggleTest(
    volatile Uint16 *MuxReg,    // Pointer to I/O port MUX register
    volatile Uint16 *ToggleReg, // Pointer to I/O port TOGGLE register
    Uint16 Mask                 // Pin Mask
);
```

Parameter: Description:
- volatile Uint16 *MuxReg
  Pointer to the desired GPIO Mux Register
- volatile Uint16 *ToggleReg
  Pointer to the desired GPIO Toggle Register
- Uint16 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 A0: Use Mask: 0x0001
- Toggle Pin A1: Use Mask: 0x0002 etc..

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

Notes:
- By using the F281x API compatibility macros provided in Flash281x_API_Library, this function could be called as Flash_ToggleTest. All of the examples shown use this generic function call. Refer to section 14.2.
- 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: F2812 Toggle XF

```c
#define GPFMUX   (volatile Uint16*)0x000070D4 // GPIO F mux
#define GPFTOGGLE (volatile Uint16*)0x000070F7 // GPIO F toggle
#define GPIOF14_XF_MASK (Uint16)0x4000 // Pin 14 mask
```
```
... 
Flash2812_ToggleTest(GPFMUX,GPFTOGGLE,GPIOF14_XF_MASK);
```
14.5. Erase Function

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

**Function Prototypes (Defined in Flash281x_API_Library.h)**

F2810:
```c
extern Uint16 Flash2810_Erase(
    Uint16 SectorMask,         // Sector mask
    FLASH_ST *FEraseStat       // Pointer to the status structure
);}
```

F2811:
```c
extern Uint16 Flash2811_Erase(
    Uint16 SectorMask,         // Sector mask
    FLASH_ST *FEraseStat       // Pointer to the status structure
);}
```

F2812:
```c
extern Uint16 Flash2812_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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>F2810</th>
<th>F2811</th>
<th>F2812</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>F</td>
<td>ignored</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>G</td>
<td>ignored</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
<td>H</td>
<td>ignored</td>
</tr>
<tr>
<td>8</td>
<td>I</td>
<td>I</td>
<td>ignored</td>
</tr>
<tr>
<td>9</td>
<td>J</td>
<td>J</td>
<td>ignored</td>
</tr>
<tr>
<td>10</td>
<td>ignored</td>
<td>ignored</td>
<td>ignored</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

FLASH_ST *FEraseStat

**Description:**

Pointer to a flash status structure. This structure is defined in Flash281x_API_Library.h:

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

**Note:** for erase, only the FirstFailAddr of this structure is currently used.
Erase continued...

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 F281x API compatibility macros provided in Flash281x_API.Library, this function can be called as Flash_Erase. All of the examples shown use this generic function call. Refer to section 14.2.
- After a sector is erased, all of its memory locations will read back 0xFFFF.
- The Erase function includes preconditioning (“clear”) and post-conditioning of the specified sectors.
- A separate step to precondition the flash is not required.
- On the F2810 the specified sectors are erased in order Sector E – Sector A
- On the F2811 and F2812 the specified sectors are erased in order Sector J – Sector A
- The minimum amount of flash memory that can be erased on the F281x devices 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 Flash281x_API.Library.h

```c
#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
#define SECTORI   (Uint16)0x0100
#define SECTORJ   (Uint16)0x0200
// All sectors on an F2811 - Sectors A - J
#define SECTOR_F2812 (SECTORA|SECTORB|SECTORC| SECTORD|SECTORE|SECTORF| SECTORG|SECTORH|SECTORI| SECTORJ)
// All sectors on an F2811 - Sectors A - J
#define SECTOR_F2811 (SECTORA|SECTORB|SECTORC| SECTORD|SECTORE|SECTORF| SECTORG|SECTORH|SECTORI| SECTORJ)
// All sectors on an F2810 - Sectors A - E
#define SECTOR_F2810 (SECTORA|SECTORB|SECTORC| SECTORD|SECTORE)
```
Erase continued...

Example:

    Uint16 Status;
    FLASH_ST EraseStatus;

    // Erase Sector I
    // Following is defined in Flash281x_API Library.h
    #define SECTORI (Uint16)0x0100

    // User’s Code:
    Status = Flash_Erase(SECTORI,&EraseStatus);
    if(Status != STATUS_SUCCESS)
    {
        Error(Status);
    }

    // Erase Sector C and Sector D

    // Following is defined in Flash281x_API Library.h
    #define SECTORC   (Uint16)0x0004
    #define SECTORD   (Uint16)0x0008

    // User’s Code:
    Status = Flash_Erase((SECTORC|SECTORD),&EraseStatus);
    if(Status != STATUS_SUCCESS)
    {
        Error(Status);
    }
14.6. Program Function

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

Function Prototypes (Defined in Flash281x_API_Library.h)

F2810:

```c
extern Uint16 Flash2810_Program(
    volatile Uint16 *FlashAddr,  // Pointer to the first flash/OTP loc
    volatile Uint16 *BufAddr,    // Pointer to the buffer
    Uint32 Length,               // Number of 16-bit values to program
    FLASH_ST *FProgStatus        // Pointer to the status structure
);
```

F2811:

```c
extern Uint16 Flash2811_Program(
    volatile Uint16 *FlashAddr,  // Pointer to the first flash/OTP loc
    volatile Uint16 *BufAddr,    // Pointer to the buffer
    Uint32 Length,               // Number of 16-bit values to program
    FLASH_ST *FProgStatus        // Pointer to the status structure
);
```

F2812:

```c
extern Uint16 Flash2812_Program(
    volatile Uint16 *FlashAddr,  // Pointer to the first flash/OTP loc
    volatile 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:

- `volatile Uint16 *FlashAddr`: Pointer to the first 16-bit location in flash or OTP to be programmed.
- `volatile 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 Flash281x_API_Library.h:

```c
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 F281x API compatibility macros provided in Flash281x_API_Library, this function can be called as Flash_Program. All of the examples shown use this generic function call. Refer to section 14.2.
- 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 0xFFFFE 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: Program a pattern of 0x400 values into the flash starting at 0x3F0000

```c
// Setup a buffer to hold the pattern to be programmed
#define WORDS_IN_FLASH_BUFFER 0x400
volatile Uint16 Buf[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

......

// Fill the buffer with some data to program into the flash
for(i=0;i<0x400;i++) Buf[i] = 0x8000+i;
Flash_ptr = (Uint16 *)0x003F0000;
Length = 0x400;

......

// Call the program API function
Status = Flash_Program(Flash_ptr,Buf,Length,&ProgStatus);
if(Status != STATUS_SUCCESS)
{
    Error(Status);
}
14.7. 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 Flash281x_API_Library.h)**

```c
F2810:
extern Uint16 Flash2810_Verify(
    volatile Uint16 *StartAddr,   // Pointer to the first flash/OTP loc
    volatile Uint16 *BufAddr,     // Pointer to the buffer
    Uint32 Length,                // Number of 16-bit values to verify
    FLASH_ST *FVerifyStat         // Pointers to the status structure
);
```

```c
F2811:
extern Uint16 Flash2811_Verify(
    volatile Uint16 *StartAddr,   // Pointer to the first flash/OTP loc
    volatile Uint16 *BufAddr,     // Pointer to the buffer
    Uint32 Length,                // Number of 16-bit values to verify
    FLASH_ST *FVerifyStat         // Pointers to the status structure
);
```

```c
F2812:
extern Uint16 Flash2812_Verify(
    volatile Uint16 *StartAddr,   // Pointer to the first flash/OTP loc
    volatile Uint16 *BufAddr,     // Pointer to the buffer
    Uint32 Length,                // Number of 16-bit values to verify
    FLASH_ST *FVerifyStat         // Pointers to the status structure
);
```

**Parameter:**
- volatile Uint16 *FlashAddr Pointer to the first location in flash or OTP to be programmed.
- volatile 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 Flash281x_API_Library.h:

```c
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.
Verify continued...

Notes:

- By using the F281x API compatibility macros provided in Flash281x_API_Library, this function can be called as Flash_Verify. All of the examples shown use this generic function call. Refer to section 14.2.

Example: Verify a pattern of 0x400 values in the flash starting at 0x3F0000

```c
// Setup a buffer to hold the pattern to be verified against
#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 verified
FLASH_ST VerifyStatus; // Status structure

......

// Fill the buffer with some data to verify
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);
}
```
14.8. Step 10: Return Status Values

To communicate back to the calling application, the API returns the following status messages. These status values are defined in the Flash281x_API_Library.h file for use within your application.

<table>
<thead>
<tr>
<th>Status</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STATUS_SUCCESS</td>
<td>Operation was successful.</td>
</tr>
<tr>
<td>10</td>
<td>STATUS_FAIL_CSM_LOCKED</td>
<td>The API function is unable to access the flash array due to a locked Code Security Module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to section 13.</td>
</tr>
<tr>
<td>11</td>
<td>STATUS_FAIL_REVID_INVALID</td>
<td>The REVID is incorrect for this version of the API.</td>
</tr>
<tr>
<td>12</td>
<td>STATUS_FAIL_ADDR_INVALID</td>
<td>An invalid address (outside of the Flash or OTP bank) was passed to the Flash API.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This could be caused by the first address being outside of Flash/OTP or the length being such</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that the last address will be outside of Flash/OTP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>None of the values will be programmed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In this case the flash status structure (FLASH_ST) is not updated with any information.</td>
</tr>
<tr>
<td>20</td>
<td>STATUS_FAIL_NO_SECTOR_SPECIFIED</td>
<td>Erase had nothing to do because no valid sectors were specified.</td>
</tr>
<tr>
<td>21</td>
<td>STATUS_FAIL_PRECONDITION</td>
<td>Erase operation failed because the precondition operation failed.</td>
</tr>
<tr>
<td>22</td>
<td>STATUS_FAIL_ERASE</td>
<td>Erase operation failed because the sector could not be erased with the maximum allowed number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of pulses.</td>
</tr>
<tr>
<td>23</td>
<td>STATUS_FAIL_COMPACT</td>
<td>Erase operation failed because the post-conditioning failed.</td>
</tr>
<tr>
<td>30</td>
<td>STATUS_FAIL_PROGRAM</td>
<td>Program operation failed because one or more bits could not be programmed.</td>
</tr>
<tr>
<td>31</td>
<td>STATUS_FAIL_ZERO_BIT_ERROR</td>
<td>Program operation failed because one or more bits were already programmed (0) that should have</td>
</tr>
<tr>
<td></td>
<td></td>
<td>been erased (1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this happens it could be because the sector was not erased before attempting to program.</td>
</tr>
<tr>
<td>40</td>
<td>STATUS_FAIL_VERIFY</td>
<td>The verify operation failed because one or more bits did not match the reference data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Try increasing the Flash or OTP wait states.</td>
</tr>
</tbody>
</table>
14.9. Files included in this release

The F2810, F2811 and F2812 each have an API library. Include the proper library for your device in your project.

Since the API function call definitions are compatible between the F281x devices, the include files and documentation is the same for both the F2810, F2811 and F2812. They have been duplicated under both sub-directories for convenience and clarity.

F2810

API Library:
\Flash28_API\Flash2810_API_V100\lib\Flash2810_API_V100.lib

API Include Files (also used for F2811 & F2812):
\Flash28_API\Flash2810_API_V100\include\Flash281x_API_Library.h
\Flash28_API\Flash2810_API_V100\include\Flash281x_API_Config.h

Example program
\Flash28_API\Flash2810_API_V100\example\n
Documentation (also used for F2811 & F2812):
\Flash28_API\Flash2810_API_V100\doc

F2811

API Library:
\Flash28_API\Flash2811_API_V100\lib\Flash2810_API_V100.lib

API Include Files (also used for F2812):
\Flash28_API\Flash2811_API_V100\include\Flash281x_API_Library.h
\Flash28_API\Flash2811_API_V100\include\Flash281x_API_Config.h

Example program
\Flash28_API\Flash2810_API_V100\example\n
Documentation (also used for F2810 & F2812):
\Flash28_API\Flash2810_API_V100\doc

F2812

API Library:
\Flash28_API\Flash2812_API_V100\lib\Flash2812_API_V100.lib

API Include Files (also used for F2810 & F2811):
\Flash28_API\Flash2812_API_V100\include\Flash281x_API_Library.h
\Flash28_API\Flash2812_API_V100\include\Flash281x_API_Config.h

Example program
\Flash28_API\Flash2812_API_V100\example\n
Documentation (also used for F2810 & F2811):
\Flash28_API\Flash2812_API_V100\doc