F2833x FLASH - API

Introduction

In Chapter 14 we discussed the option to start our embedded control program directly from the F2833x internal Flash memory. Another important task of a real-world project is to update parts of the internal FLASH whilst the control code is still running in FLASH memory.

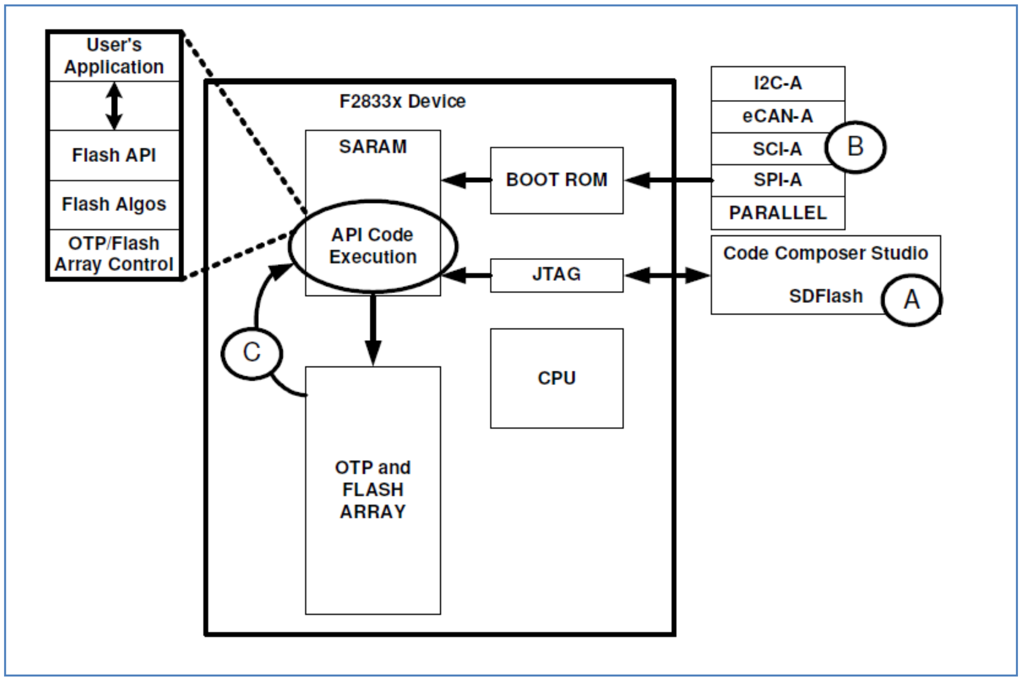
Texas Instruments provides an "Application Programmers Interface" (API) - library for such purposes. The API is free for download from Texas Instruments website (www.ti.com). There are different versions of this library, depending on the type of device. For the F2833x, the literature number is "SPRC539" and for the F2823x it can be found under "SPRC665". For the laboratory exercise at the end of this chapter it is necessary that you have installed the correct library on your PC. The default installation path is either:

C:\tidcs\c28\Flash28_API\Flash28335_API_V210\ or

C:\tidcs\c28\Flash28_API\Flash28332_API_V210

If the library is not already present on your PC, download the corresponding latest archive file from the website, unzip and install it on your PC.

Here is a block diagram that shows the execution flow, when FLASH - API - algorithms are involved. Method "C" shows the embedded code solution, which we will discuss and perform a lab exercise later in this chapter.



Module Topics

F2833x FLASH - API **16-1**

Introduction *16-1*

Module Topics *16-2*

F2833x FLASH - API Installation *16-3*

F2833x FLASH API Fundamentals *16-4*
Erase 16-4
Program 16-5
Verify 16-5

General Guidelines *16-6*

FLASH - API Checklist *16-7*
Step1: Modify Flash2833x_API_Config.h 16-8
Step 2: Include Flash2833x_API_Library.h 16-8
Step 3: Include the appropriate Flash API library 16-9
Step 4: Initialize PLL Control Register (PLLCR) 16-9
Step 5: Check PLL Status for Limp Mode Operation 16-9
Step 6: Copy the Flash API functions to Internal SARAM 16-10
Step 7: Initialize Flash_CPUScaleFactor 16-10
Step 8: Initialize the Callback Function Pointer 16-10

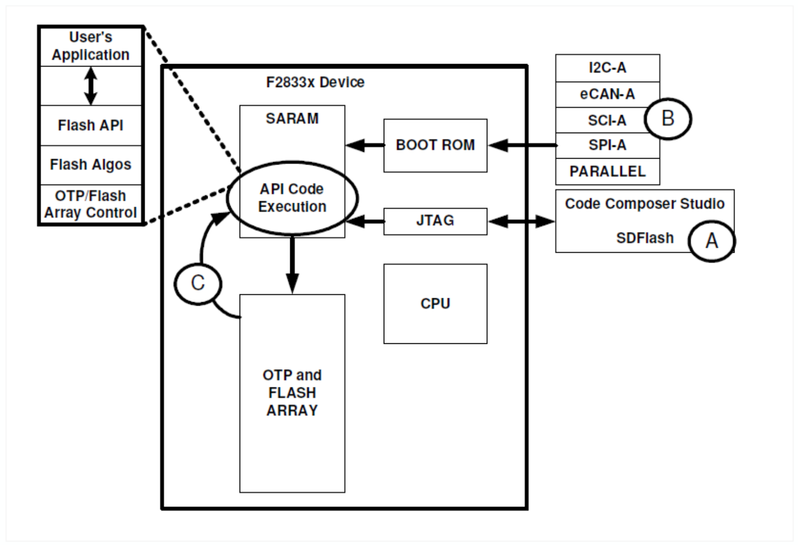
F2833x FLASH - API Reference *16-11*
Data Type Conventions 16-11
API Function Naming Conventions 16-11
FLASH - API - Functions 16-12
Files included with the API 16-12

Lab 16: Use of FLASH - API *16-13*
Objective 16-13
Procedure 16-14
Open Project 16-14
Build project 16-18
Verify Linker Results - The map - File 16-18
Use CCS integrated Flash Program Tool 16-19
Close CCS & Restart the Peripheral Explorer Board 16-20

F2833x FLASH - API Installation

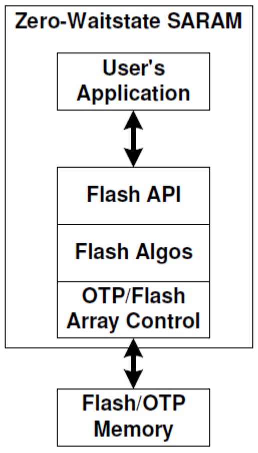
The F2833x FLASH - API function library can be downloaded free of charge from the Texas Instruments website. It supports FLASH programming via embedded function calls, as shown as path ‘C’ in Slide 16-2. All functions can be integrated into the code of an existing project.

TMS320F2833x FLASH Load Options



16 - 2

FLASH – API - Installation

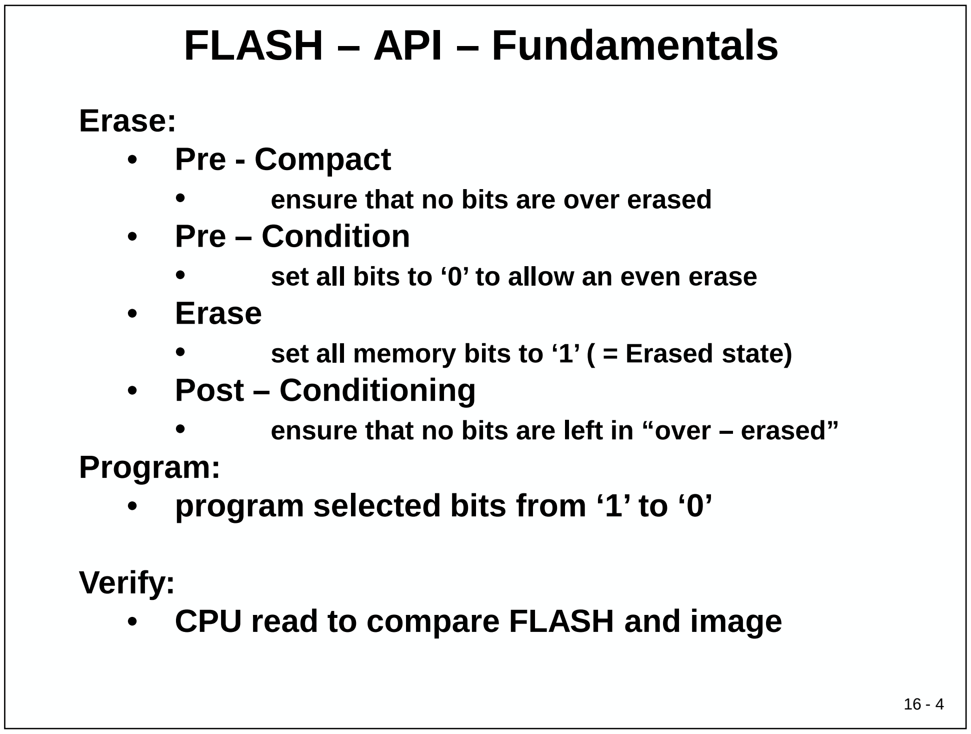


1. Download from: www.ti.com:
   - F2833x: “SPRC539”
   - F2823x: “SPRC665”
2. Default Installation path:
   - C:\tidcs\c28\FLASH28_API\
3. Read included documentation
   - “Flash2833x_API_Readme.pdf”

16 - 3

F2833x FLASH API Fundamentals

The Flash Application Program Interface (Flash API) consists of 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 provides API functions to erase, program and verify the flash array as briefly described here:



FLASH – API – Fundamentals

Erase:

- **Pre - Compact**
  - **ensure that no bits are over erased**
- **Pre – Condition**
  - **set all bits to ‘0’ to allow an even erase**
- **Erase**
  - **set all memory bits to ‘1’ (= Erased state)**
- **Post – Conditioning**
  - **ensure that no bits are left in “over – erased”**

Program:

- **program selected bits from ‘1’ to ‘0’**

Verify:

- **CPU read to compare FLASH and image**

16 - 4

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 0 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 particular 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 for the F2833x 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 in this chapter.

For a detailed description of all API - functions please refer to document "FLASH 2833x_API_Readme.pdf" (part of SPRC539.zip).

General Guidelines

Here is a list of general rules that should be followed, when using the FLASH - API:

1. Install the latest and correct version of the FLASH - API. For the F28335, the literature number is “SPRC539”. The default location of the package is: “C:\tidcs\c28\Flash28_API”.

2. Execute the Flash API code from zero-wait state internal SARAM memory.

3. Configure the API for the correct CPU frequency of operation.

4. Follow the Flash API checklist in section 5 of “FLASH 2833x_API_Readme.pdf” to integrate the API into an application.

5. Initialize the PLL control register (PLLCR) and wait for the PLL to lock before calling an API function.

6. 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. 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 of “FLASH 2833x_API_Readme.pdf”.

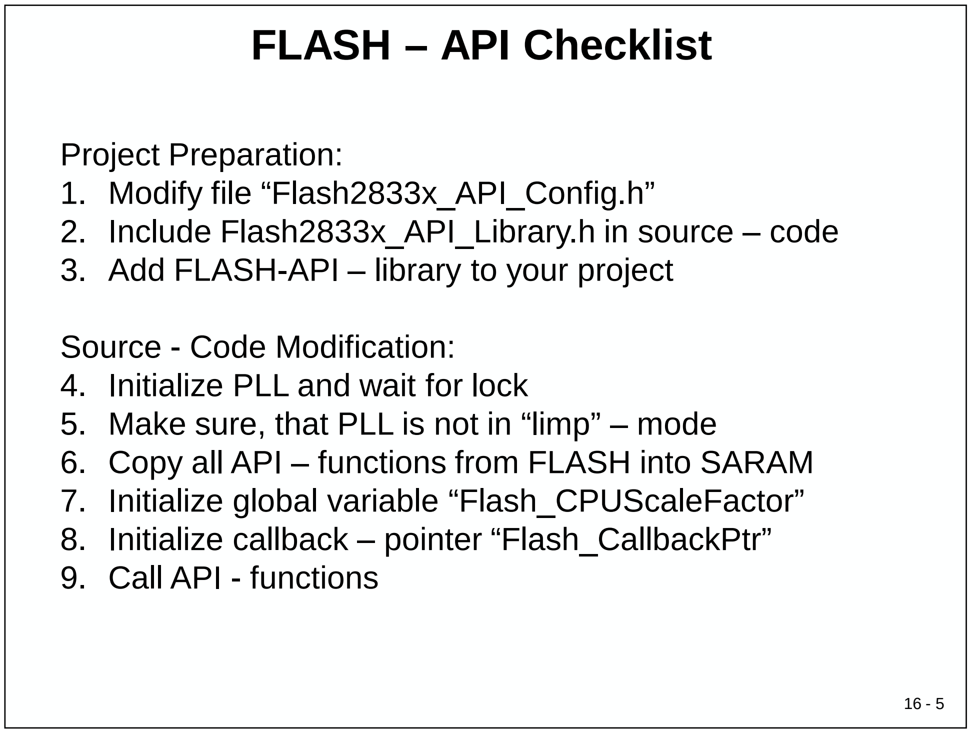
There is also a list what should be not done:

7. Do not execute the Flash APIs from the flash or OTP. If the APIs are stored in flash or OTP memory, they must first be copied to internal SARAM before they are executed.
8. Do not 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.
9. Do not 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 do the flash and OTP become available.
10. Do not stop the erase, program or depletion recovery functions while they are executing (for example, do not stop the debugger within API code, do not reset the part, etc).
11. 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.

Sounds pretty complicated, doesn’t it? Well, since we are students we can keep it simple (first). Later, when we have a functional framework, we can implement a more detailed solution.

FLASH - API Checklist

Here is the sequence of steps required to use parts of the FLASH - API Library code:



FLASH – API Checklist

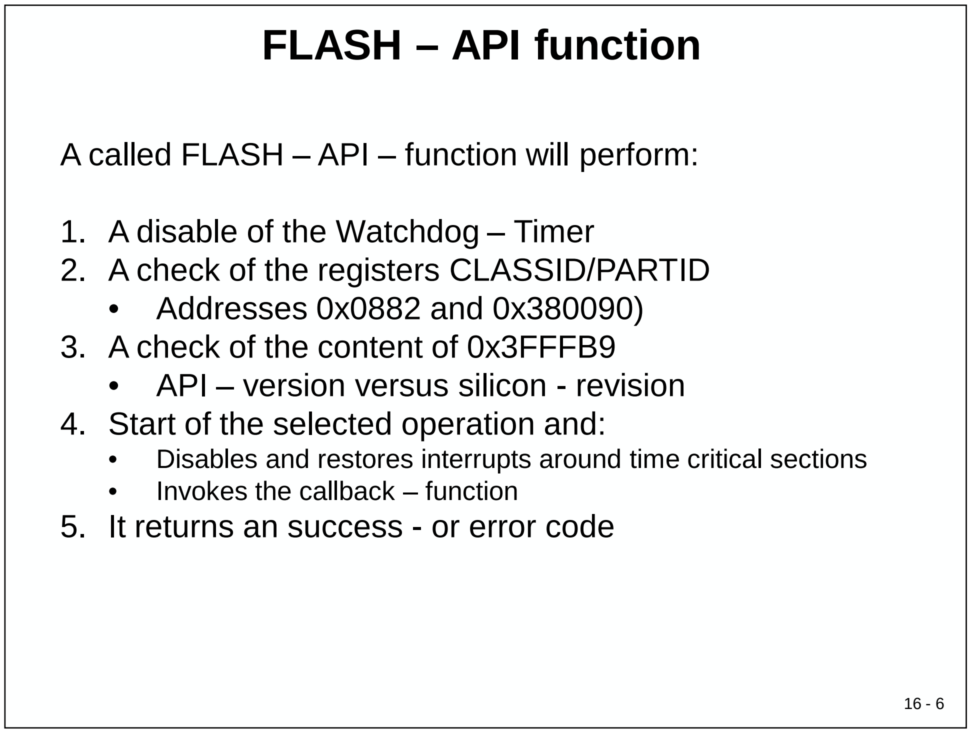
Project Preparation:
1. Modify file “Flash2833x_API_Config.h”
2. Include Flash2833x_API_Library.h in source – code
3. Add FLASH-API – library to your project

Source - Code Modification:

4. Initialize PLL and wait for lock
5. Make sure, that PLL is not in “limp” – mode
6. Copy all API – functions from FLASH into SARAM
7. Initialize global variable “Flash_CPUScaleFactor”
8. Initialize callback – pointer “Flash_CallbackPtr”
9. Call API - functions

16 - 5

A called API - Function will perform the following actions:



FLASH – API function

A called FLASH – API – function will perform:

1. A disable of the Watchdog – Timer
2. A check of the registers CLASSID/PARTID
   - Addresses 0x0882 and 0x380090)
3. A check of the content of 0x3FFFB9
   - API – version versus silicon - revision
4. Start of the selected operation and:
   - Disables and restores interrupts around time critical sections
   - Invokes the callback – function
5. It returns an success - or error code

16 - 6

Step1: Modify Flash2833x_API_Config.h

Modify file “Flash2833x_API_Config.h” to be found in the include directory of each API, to match your specific target. Set the corresponding line to ‘1’:

```
#define FLASH_F28335   1
#define FLASH_F28334   0
#define FLASH_F28332   0
```

Uncomment the line corresponding to the CPU Clock rate (SYSCLKOUT) in nanoseconds at which the API functions will run. This is done by removing the leading // in front of the required 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 150 MHz. This corresponds to a 6.667 ns cycle time. If there is no line present for this clock speed, so you should insert your own entry and comment out all other entries:

```
#define CPU_RATE     6.667L      // for a 150MHz
      SYSCLKOUT
//#define CPU_RATE    10.000L    // for a 100MHz
      SYSCLKOUT
//#define CPU_RATE    13.330L    // for a 75MHz
      SYSCLKOUT
//#define CPU_RATE    20.000L    // for a 50MHz
      SYSCLKOUT
//#define CPU_RATE    33.333L    // for a 30MHz
      SYSCLKOUT
//#define CPU_RATE    41.667L    // for a 24MHz
      SYSCLKOUT
//#define CPU_RATE    50.000L    // for a 20MHz
      SYSCLKOUT
//#define CPU_RATE    66.667L    // for a 15MHz
      SYSCLKOUT
//#define CPU_RATE   100.000L    // for a 10MHz
      SYSCLKOUT
```

The CPU clock rate is used during the compile phase 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 Flash2833x_API_Config.h file provides this calculation:

```
#define SCALE_FACTOR  1048576.0L*( (200L/CPU_RATE) )  // IQ20
```

Step 2: Include Flash2833x_API_Library.h

The file “Flash2833x_API_Library.h” is the main include file for the Flash API and should be included in any application source - code file that interfaces to the Flash API.

```
#include "FLASH2833x_API_Library.h"
```

Also, include the search path to this header - file into the project C/C++ build options. In the "C/C++" perspective, right click on the active project, select "properties", C2000 compiler, Include Options and add:

```
C:\tidcs\c28\Flash28_API\Flash28335_API_V210\include
```

Step 3: Include the appropriate Flash API library

The appropriate Flash API library must also be linked to your project.

By default, the symbol " <>" stands for "C:\tidcs\c28"

F28335: <>\Flash28_API\Flash28335_API_V210\lib\Flash28335_API_V210.lib
F28334: <>\Flash28_API\Flash28334_API_V210\lib\Flash28334_API_V210.lib
F28332: <>\Flash28_API\Flash28332_API_V210\lib\Flash28332_API_V210.lib

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).

The F2833x Flash APIs have been compiled using the "--float_support=fpu32" floating point option. Only object files compiled as such can be linked to the APIs.

Step 4: Initialize PLL Control Register (PLLCR)

It is vital that the API functions be run at the correct clock 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 device appropriate 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 to 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.

The best way to follow these requirements for setting up the PLL is to call function "InitSysCtrl()", provided by Texas Instruments in file "DSP2833x_SysCtrl.c".

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 the device appropriate TMS320x2833x System Control and Interrupts Reference Guide for more information on the missing clock detection logic of the F2833x devices.

Step 6: Copy the Flash API functions to Internal SARAM

If 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 sequence describes how to accomplish this copy:

- Link the linker command file “F28335.cmd” to your project
- This linker command file defines 3 symbols:
    - Load address start: Flash28_API_LoadStart
    - Load address end: Flash28_API_LoadEnd
    - Run address start: Flash28_API_RunStart
- Use the symbols in a copy loop, such as:

```
Uint16 * pSourceAddr;
Uint16 * pDestAddr;
Uint16 i;
pSourceAddr = &Flash28_API_LoadStart;
pDestAddr = &Flash28_API_RunStart;
for(i=0;i<(&Flash28_API_LoadEnd- &Flash28_API_LoadStart);
i++)
{
   *pDestAddr++ = *pSourceAddr++;
}
```

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).

```
Flash_CPUScaleFactor = SCALE_FACTOR;
```

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.

The variable "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. 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.

```
Flash_CallbackPtr = NULL;
```

F2833x FLASH - API Reference

Data Type Conventions

The following data type definitions are defined in Flash2833x_API_Library.h:

```
#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
```

API Function Naming Conventions

The F2833x API function names are of the following form:

`Flash`*<device>*_*<operation>*`(args)`

Where

<device> is 28335, 28334, 28332

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

For example:

```
Flash28335_Program(args)
```

is the F28335 Program function.

The API function definitions for the F2833x API libraries are compatible. For this reason the file
Flash2833x_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)`

The use of these macros is optional. They have been provided to allow easy porting of code between the devices.

FLASH - API - Functions

The following API - Functions are available:

Generic Function	F28335 API Function
Flash_ToggleTest	Flash28335_ToggleTest
Flash_Erase	Flash28335_Erase
Flash_Program	Flash28335_Program
Flash_Verify	Flash28335_Verify
Flash_DepRecover	Flash28335_DepRecover
Flash_APIVersion	Flash28335_APIVersion
Flash_APIVersionHex	Flash28335_APIVersionHex

All functions use a 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 Flash2833x_API_Library.h:

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

}FLASH_ST;
```

For the parameter list of all API - functions please refer to the documentation file “Flash2833x_API_Readme.pdf”.

Files included with the API

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

API Library:
<base>\Flash28335_API_V210\lib\Flash28335_API_V210.lib

API Include Files:
<base>\Flash28335_API_V210\include\Flash2833x_API_Library.h
<base>\Flash28335_API_V210\include\Flash2833x_API_Config.h

Documentation:
< base >\Flash28335_API_V210\doc

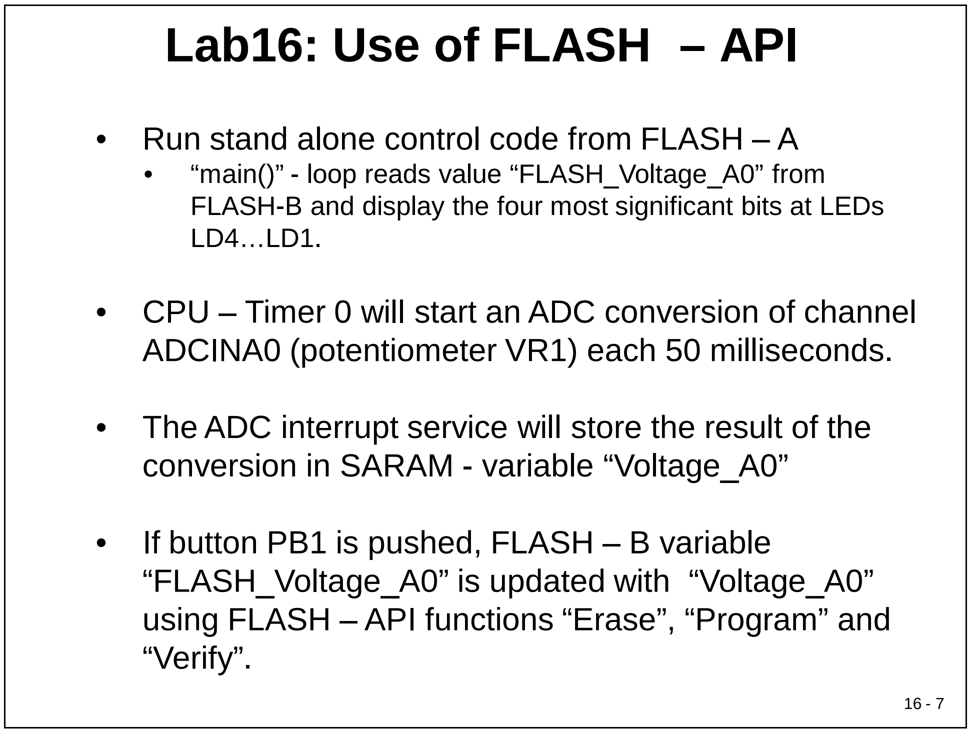
Example:
< base >\Flash28335_API_V210\example

Lab 16: Use of FLASH - API

Objective

The objective of this laboratory exercise is to practice using the F2833x FLASH - API library. Here is what we will do:

- We will run a small amount of control code direct from FLASH - A. The main - loop of this control code will permanently read a data memory variable "FLASH_Voltage_A0", located in FLASH - section B, and display the four most significant bits (bit 11…bit 8) of "FLASH_Voltage_A0" on four LEDs (LD4…LD1).



Lab16: Use of FLASH – API

- Run stand alone control code from FLASH – A
  - "main()" - loop reads value "FLASH_Voltage_A0" from FLASH-B and display the four most significant bits at LEDs LD4…LD1.
- CPU – Timer 0 will start an ADC conversion of channel ADCINA0 (potentiometer VR1) each 50 milliseconds.
- The ADC interrupt service will store the result of the conversion in SARAM - variable "Voltage_A0"
- If button PB1 is pushed, FLASH – B variable "FLASH_Voltage_A0" is updated with "Voltage_A0" using FLASH – API functions "Erase", "Program" and "Verify".

16 - 7

- ADC channel ADCINA0, started by CPU - Timer 0 every 50 milliseconds, will convert the value from potentiometer "VR1" of the Peripheral Explorer Board into a local 12 - bit - variable "Voltage_A0". The CPU-Timer 0 - Interrupt service will perform the software - start of the ADC; the ADC - Interrupt service will update "Voltage_A0"

- If we push button "PB1" of the Peripheral Explorer Board, we start another part of the control code. We will call our function "Update_FLASHB()" to update the FLASH-B - variable "FLASH_Voltage_A0" with the current value from "Voltage_A0". This function includes some API - Function calls from the Texas Instruments FLASH-API.

- After a successful update of FLASH-B, our code will perform a warm reset to re-start the code. To do so, you have to set the boot-sequence to "Boot to FLASH". On the "Peripheral Explorer Board", make sure that jumper J3 ("SCI_BOOT 84) is open!
- Note: The provided test function "Update_FLASHB()" is intended to be used in experimental student laboratories only, it does not cover error - handling or timeout monitoring. For a real product version you can easily extend the functionality of this example; all code sequences are based on Texas Instruments API - Functions. To keep the first example simple, we will use basic API - features only.

Procedure

Open Project

1. For convenience, open the project "Lab16.pjt" from C:\DSP2833x_V4\Labs\Lab16. If you create your own project, you have to add the provided files from C:\DSP2833x_V4\Labs\Lab16 manually.
2. From "C:\DSP2833x_V4\Labs\Lab14" open the file "Lab14.c", save it in "C:\DSP2833x_V4\Labs\Lab16" as "Lab16.c" and add "Lab16.c" to your project.
3. Open the file "Lab16.c" to edit.
   - At the beginning of "Lab16.c", add two macros to define the push-buttons PB1 and PB2:

```
#define START    GpioDataRegs.GPADAT.bit.GPIO17    // Button PB1
#define STOP     GpioDataRegs.GPBDAT.bit.GPIO48    // Button PB2
```

   - Also at the beginning of "main()", add an external function prototype for function "Update_FLASHB()". This function is defined in file "Lab16_FLASH_API.c", which will be inspected shortly. Add:

```
extern void Update_FLASHB(int);
```

   - Since we will use the ADC, we will also call function "InitAdc()", which is defined in the file "DSP2833x_ADC.c". Add a 2nd additional external function prototype:

```
extern void InitAdc(void);
```

   - Third, add another external function prototype:

```
extern void display_ADC(unsigned int);
```

     This function, defined in the provided source code file "Display_ADC.c", will be used to convert the four most significant bits of an input value into a "light-beam" at the four LEDs LD1…LD4.

   - Next, add a prototype for the local ADC interrupt service routine. Add:

```
interrupt void adc_isr(void);
```

   - Finally add two global variables:

unsigned int Voltage_A0;
extern unsigned int FLASH_Voltage_A0;

The variable "FLASH_Voltage_A0" is already defined in the file "Lab16_FLASH_DATA.c"; therefore we need the "extern" keyword.

4. Edit function "main()" of file "Lab16.c":

- Delete the variable "counter" and the code in the endless while(1)-loop of "main()", which is related to "counter".
- Next, add a line to re-load the entry for the ADC in the PIE -vector table with the name of our own interrupt service function. Search for line of code, which we used to reload TINT0 and add:

  PieVectTable.ADCINT = &adc_isr;

- Change the function call to "ConfigCpuTimer()" from a 100 milliseconds to a 50 milliseconds period.
- After the function call to "ConfigCpuTimer()", add a new call to the function "InitAdc()". This function, provided by Texas Instruments, will switch the ADC - module to a default standby mode. Add:

  InitAdc();

- Next, add the initialization for the ADC sequencer unit.

  For register "ADCTRL1":

  - set acquisition window to 8 clocks (ACQ_PS)
  - set clock prescaler CPS to "divide by 1"
  - select single run mode (CONT_RUN)
  - select cascaded mode (SEQ_CASC)

  For register "ADCTRL2":

  - enable interrupt (INT_ENA_SEQ1)

  For register "ADCCTRL3":

  - Set bit fields "ADCCLKPS" to select HSPCLK / 8

  Set "MAXCONV" to convert 1 channel

  Set channel selector "CONV00" to convert ADCINA0

- Next, add a line to enable also PIE-interrupt 1.6 (ADC). Note: PIE-interrupt 1.7 (Timer 0) is also active; keep its enable command line in your code. Add:

  PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

5. Edit the endless while(1) - loop of function “main()”:

- At the beginning of this loop, add a call to the function “display_ADC(0);”. This will switch OFF all four LEDs LD1…LD4.
- Next, wait for 100 milliseconds
- Next update the LEDs by a 2nd function call “display_ADC(FLASH_Voltage_A0);”. This will update the LEDs with the value from the global variable “FLASH_Voltage_A0”
- Next, wait for another 100 milliseconds, before you clear the Timer 0 interrupt counter and before you service the watchdog. The whole new code snippet looks like:

```
display_ADC(0);
while(CpuTimer0.InterruptCount < 2);
display_ADC(FLASH_Voltage_A0);
while(CpuTimer0.InterruptCount < 4);
CpuTimer0.InterruptCount = 0;
EALLOW;
SysCtrlRegs.WDKEY = 0x55;
EDIS;
```

- Finally, we have to add code that samples button “PB1”. In the event of an active button (pushed down = 0), we have to call our FLASH re-programming function “Update_FLASHB(Voltage_A0)”. After returning from this call, the new data values are programmed into FLASH and we have to start the F2833x with a reset. The question is: how can we cause a reset by an instruction? One answer is: the watchdog control register does the trick. If we intentionally violate the watchdog control register security bits (WDCHK2…0) by writing 000 into this bit-field, we cause a reset. The whole code snippet look like this:

```
if (START == 0)        // START Button is pressed down (zero)
{
        Update_FLASHB(Voltage_A0);
        EALLOW;
        SysCtrlRegs.WDCR = 0;     // force a “warm” - RESET
        while(1);                 // line is never reached
}
```

6. Change CPU - Timer 0 Interrupt Service Routine

When we enter the FLASH - API functions, the hardware interrupts are still active and must be serviced. The problem is that we cannot execute code from FLASH, when we re-program it. This includes the Interrupt Service Functions. The solution is to copy the ISRs from FLASH to RAM at the beginning of the code execution. We have used a similar principle for function “InitFlash()”. All we need is a connection of the RAM runtime functions to section “ramfuncs”. In front of function “cpu_timer0_isr()”, add:

```
#pragma CODE_SECTION(cpu_timer0_isr, "ramfuncs");
```

To start the ADC, add a line in the function “cpu_timer0_isr()” to force a software start:

```
AdcRegs.ADCTRL2.bit.SOC_SEQ1 = 1;    // start ADC by SW
```

At the end of “Lab16.c” add a new interrupt function “adc_isr()”.

Again, use a pragma - statement to assign this function to a RAM run - time location

```
#pragma CODE_SECTION(adc_isr, "ramfuncs");
```

In this function, read the ADC result, store it in the global variable “Voltage_A0” and clear the ADC for the next conversion. The whole function is:

```
interrupt void  adc_isr(void)
{
        Voltage_A0 = AdcRegs.ADCRESULT0>>4;
  AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;            // Reset SEQ1
  AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;          // Clear INT SEQ1 bit
  PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;      // Acknowledge PIE
}
```

7. Inspect and adjust the header file “Flash2833x_API_Config.h”. This file defines a few macros. Make sure to have those macros active that correspond to your F28335ControlCard. There are two versions out, a 20MHz (100MHz SYSCLKOUT) and a 30MHz (150MHz SYSCLKOUT) version. The following snippet is for a F28335 running at external 30MHz clock speed:

```
#define FLASH_F28335   1
#define FLASH_F28334   0
#define FLASH_F28332   0
#define CPU_RATE    6.667L   // for a 150MHz CPU (SYSCLKOUT)
//#define CPU_RATE   10.000L    // for a 100MHz CPU (SYSCLKOUT)
```

8. Inspect the provided file “Lab16_FLASH_DATA.c”. This file defines a new global variable “FLASH_Voltage_A0”. The “DATA_SECTION” directive connects the variable to a linker symbol “myFlashConstants”.

```
#ifdef  __cplusplus
#pragma DATA_SECTION("myFlashConstants")
#else
#pragma DATA_SECTION(FLASH_Voltage_A0,"myFlashConstants");
#endif
volatile unsigned int FLASH_Voltage_A0;
```

9. Inspect the provided linker command file “Lab16.cmd”.

```
SECTIONS
{
        myFlashConstants    : > FLASHB      PAGE =1
        Flash28_API:
        {
                -lFlash28335_API_V210.lib(.econst)
                -lFlash28335_API_V210.lib(.text)
         }       LOAD = FLASHD,
                RUN = RAML0,
                LOAD_START(_Flash28_API_LoadStart),
                LOAD_END(_Flash28_API_LoadEnd),
                RUN_START(_Flash28_API_RunStart),
                PAGE = 0
}
```

First, this file connects the section “myFlashConstants” to physical FLASH-B memory block (0x330000). Second, it connects section “Flash28_API” to a load address in FLASH- D block (0x320000) and to run-address RAML0 (0x8000). It also defines symbols “Flash28_API_LoadStart”, “Flash28_API_RunStart” and “Flash28_API_LoadEnd”, which are used in the next source file (see procedure step 11). All memory blocks (FLASHB, FLASHD, and RAML0) are defined in the default linker command file “F28335.cmd”.

10. Inspect the source code file “Lab16_FLASH_API.c”. This file is an example on how to use the FLASH-API - functions. Note: Again, this is an example just for student exercises and not for real production code. It does not cover any error situations, as you can see in the rather sparse function “Error()” at the end of this file.

The function “Update_FLASHB()” basically performs the following steps:

(1) It checks, whether the F2833x is in “Limp”-Mode (clock has been lost). If so, the function just returns (which is one point to be improved for production code)

(2) If not in limp - mode, it copies all FLASH-API - functions from FLASHD into RAML0.

(3) Next, it checks the correct FLASH-API - version (“Flash_APIVersionHex()”).

(4) If the version is correct, it erases FLASHB by function call to “**Flash_Erase**(SECTORB, &Flash_Status)”.

(5) It programs new data into section FLASHB by function call “Flash_Program(&FLASH_Voltage_A0,&new_value,1,&Flash_Status)”.

Now let us finish the lab exercise!

Build project

11. Click the “Rebuild Active Project ” button or perform:

Project → Rebuild All (Alt +B)

Verify Linker Results - The map - File

12. Before we actually start the Flash programming, it is always good practice to verify the used sections of the project. This is done by inspecting the linker output file “lab16.map”

13. Open the file “lab16.map” in the sub-folder ..\Debug

In the “MEMORY CONFIGURATION” column "used" you will find the amount of physical memory that is used by your project.

Verify that only the following five lines from PAGE 0 are used:

```
Name          origin     length     used       unused    attr
RAML0       00008000   00001000   0000055f   00000aa1   RWIX
FLASHD      00320000   00008000   0000055f   00007aa1   RWIX
FLASHA      00338000   00007f80   000007f3   0000778d   RWIX
BEGIN       0033fff6   00000002   00000002   00000000   RWIX
ADC_CAL     00380080   00000009   00000007   00000002   RWIX
```

The number of addresses used in FLASHA and FLASHD might be different in your lab session. Depending on how efficiently you programmed your code, you will end up with more or less words in this section.

Verify that in PAGE1 section FLASHB has been allocated:

```
Name          origin     length     used       unused    attr
FLASHB        00330000   00008000   00000001   00007fff   RWIX
```

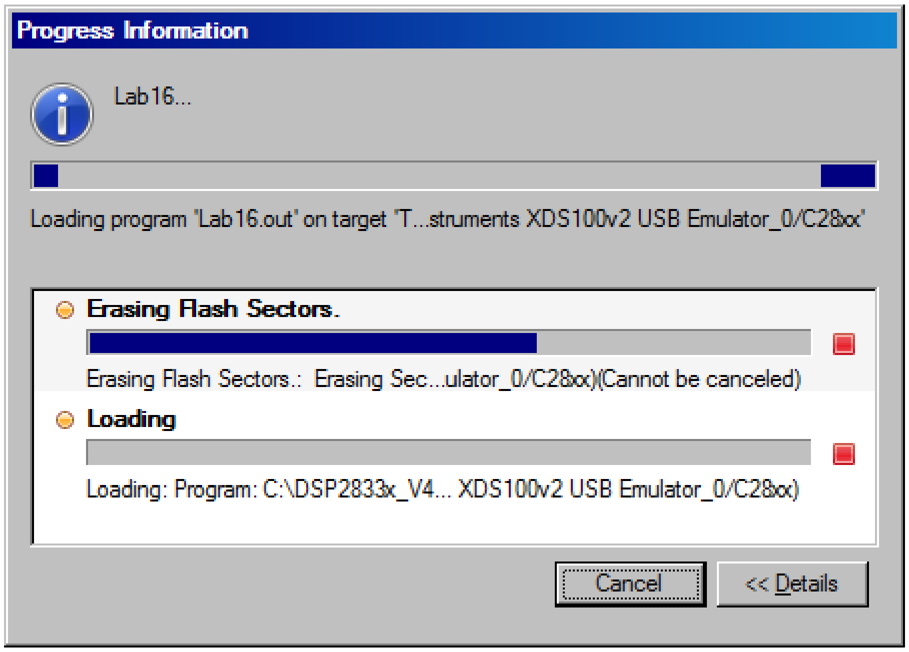
In the SECTION ALLOCATION MAP you can see how the different portions of our project's code files are distributed into the physical memory sections. For example, the .text - entry shows all the objects that were concatenated into FLASHA.

Entry symbol "codestart" connects the object "CodeStartBranch.obj" to physical address 0x33 FFF6 and occupies two words.

Use CCS integrated Flash Program Tool

14. Perform ➔ Target ➔ Debug Active Project

The FLASH based sections of the project will be erased and programmed automatically!


Progress Information

Lab16...

Loading program 'Lab16.out' on target 'T...struments XDS100v2 USB Emulator_0/C28xx'

Erasing Flash Sectors.

Erasing Flash Sectors.: Erasing Sec...ulator_0/C28xx)(Cannot be canceled)

Loading

Loading: Program: C:\DSP2833x_V4... XDS100v2 USB Emulator_0/C28xx)

Close CCS & Restart the Peripheral Explorer Board

15. Close your Code Composer Studio session.

16. Disconnect power from the Peripheral Explorer Board.

17. Verify that Peripheral Explorer Board Jumper J3 (“SCI-BOOT GPIO84”) is open

Test Application

18. Re-connect the Peripheral Explorer Board to the power supply. The code should run immediately after power on. If this is you first test of “Lab16” and FLASH-B has not been used so far, e.g. variable “FLASH_Voltage_A0” is still programmed with 0xFFFF, the four LEDs LD1…LD4 should blink simultaneously at 100 milliseconds intervals.

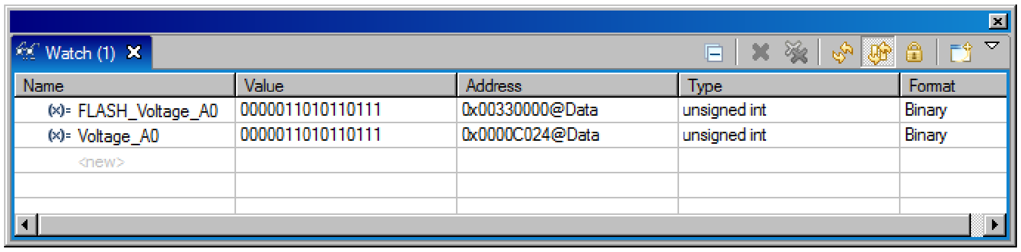
19. Turn potentiometer VR1 into its middle position. Next push PB1 shortly. This push should start the FLASH - programming sequence and program the new voltage into FLASHB. The LED - blinking should stop for approximately 1 second. After that programming time, the code should start again, now showing the new value in FLASHB.

20. If you power OFF and ON again, the code should immediately show the value, which was stored in FLASHB, before powering OFF the tool.

21. Re-Start Code Composer Studio and connect to the target.

22. To test code in FLASH, we can also apply a symbolic test strategy.

- Reload project “Lab16.pjt”.
- Perform “Go main” and Run (F8)
- Inspect variables “FLASH_Voltage_A0” and “Voltage_A0”.



Name	Value	Address	Type	Format
(x)= FLASH_Voltage_A0	0000011010110111	0x00330000@Data	unsigned int	Binary
(x)= Voltage_A0	0000011010110111	0x0000C024@Data	unsigned int	Binary
<new>				

- LEDs LD1...LD4 are toggled between “0000” and the corresponding values in bit 11 to bit 8 of “FLASH_Voltage_A0”

END of Lab16.