

---

# F2833x Flash Programming

---

## Introduction

So far we have used the internal volatile memory (L1 - SARAM) of the F2833x to store the code for our examples. Before we could execute the code we used Code Composer Studio to load it into L1 - SARAM (“File” → “Load Program”). This is fine for projects in a development and debug phase where there are frequent changes to parts and components of the software. However, when it comes to production versions with a standalone embedded control unit based on the F2833x, we no longer have the option to download our control code using Code Composer Studio. Imagine a control unit for an automotive braking system, where you have to download the control code first when you hit the brake pedal (“Do you really want to brake? ...”).

For standalone embedded control applications, we need to store our control code in NON-Volatile memory. This way it will be available immediately after system power-up. The question is: what type of non-volatile memory is available? There are several physically different memories of this type: Read Only Memory (ROM), Electrically Programmable Read Only Memory (EPROM), Electrically Programmable and Erasable Read Only Memory (EEPROM) and Flash-Memory. In the case of the F28335, we can add any of the above types of memory to the control unit using the external interface (XINTF).

The F2833x is also equipped with an internal Flash memory of 256K x 16 bits. This is quite a large amount of memory and more than sufficient for our lab exercises!

Before we can go to modify one of our existing lab solutions to start up out of Flash memory, we have to go through a short explanation of how to use this memory. This module also covers the boot sequence of the F2833x - what happens when we power on the F2833x?

This chapter also covers the password feature of the F2833x code security module. This module is used to embed dedicated portions of the F2833x memory in a secure section with a 128 bit-password. If the user does not know the correct combination that was programmed into the password section, any access to the secured areas will be denied! This is a security measure to prevent reverse-engineering.

At the end of this lesson we will do a lab exercise to load one of our existing solutions into the internal Flash memory.

**CAUTION:** Please do not upset your teacher by programming the password area! Be careful, if you program the password by accident the device will be locked forever! If you decide to make your mark at your university by locking the device with your own password, be sure to have passed all your exams first.

# Module Topics

<b>F2833x Flash Programming.....</b>	<b>14-1</b>
<i>Introduction.....</i>	<i>14-1</i>
<i>Module Topics.....</i>	<i>14-2</i>
<i>F2833x Start-up Sequences.....</i>	<i>14-3</i>
<i>F2833x Flash Memory Sectors.....</i>	<i>14-5</i>
<i>Flash Speed Initialization.....</i>	<i>14-5</i>
<i>Flash Configuration Registers .....</i>	<i>14-8</i>
<i>Flash Programming Procedure.....</i>	<i>14-9</i>
<i>CCS Flash Plug-In .....</i>	<i>14-11</i>
<i>Code Security Mode .....</i>	<i>14-12</i>
<i>Lab Exercise 14: Standalone Project.....</i>	<i>14-16</i>
Objective.....	14-16
Procedure .....	14-17
Open Files, Create Project File .....	14-17
Project Build Options.....	14-18
Add Additional Source Code Files .....	14-18
Modify Source Code to Speed up Flash memory .....	14-18
Build project .....	14-19
Verify Linker Results: The “.map” - File.....	14-20
Use CCS integrated Flash Programming .....	14-20
Shut down CCS and Restart FLASH - Code .....	14-21

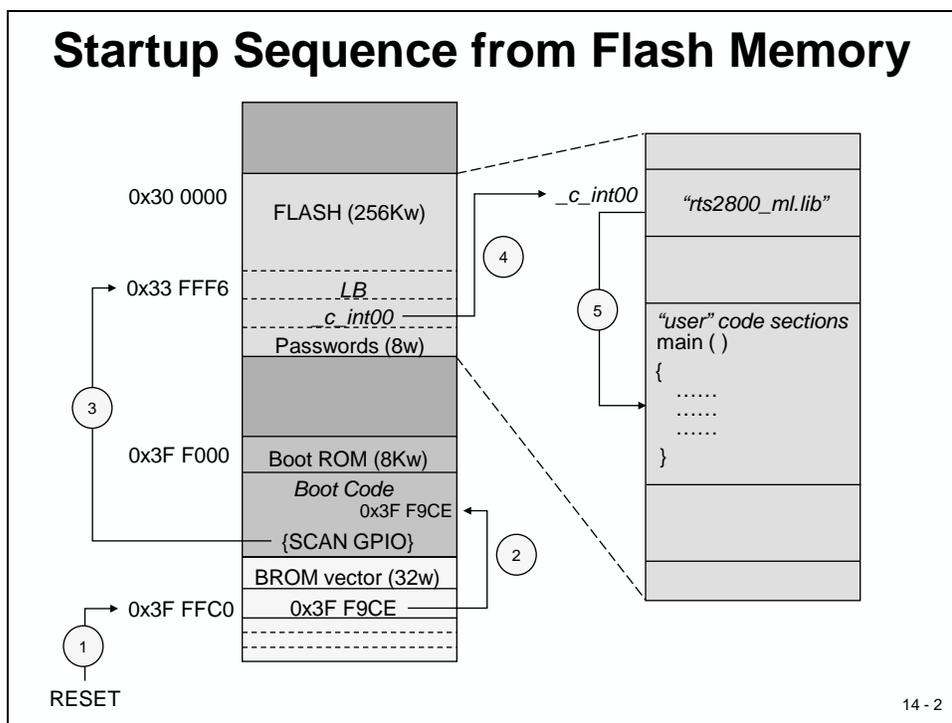
## F2833x Start-up Sequences

There are 16 different options to start the F2833x out of power-on. The options are hard-coded by 4 GPIO-Inputs (GPIO 87, 86, 85 and 84). The 4 pins are sampled during power-on. Depending on their status, one of the following options is selected:

GPIO87	GPIO86	GPIO85	GPIO84	Mode
1	1	1	1	Jump to Flash 0x33FFF6
1	1	1	0	SCI-A Boot loader
1	1	0	1	SPI-A Boot loader
1	1	0	0	I2C-A Boot loader
1	0	1	1	eCAN-A Boot loader
1	0	1	0	McBSP-A Boot loader
1	0	0	1	Jump to XINTF x16
1	0	0	0	Jump to XINTF x32
0	1	1	1	Jump to OTP
0	1	1	0	Parallel GPIO – Boot loader
0	1	0	1	Parallel XINTF – Boot loader
0	1	0	0	Jump to SARAM 0x000000
0	0	1	1	Jump “to check boot mode”
0	0	1	0	Jump to Flash, without ADC – calibration
0	0	0	1	Jump to SARAM, without ADC – calibration
0	0	0	0	SCI – A boot loader, without ADC - calibration

On the F28335ControlCard, the four GPIOs are pulled high by resistors R3, R4, R5 and R14 (47 kOhm each) to code “1111” (FLASH). The Peripheral Explorer Board offers only one other selection for the boot mode: a closed header J3 allows to pull-down GPIO84 to select

“1110” (SCI-A Boot loader). The following slide shows the sequence that takes place when we start from Flash.



1. RESET-address is always defined in address 0x3F FFC0. This is part of TI's internal BOOT-ROM. This address is loaded into the program counter (PC).
2. The BOOT-ROM code performs a basic initialization of the CPU and selects the boot-code sequence or calculates the entry point address.
3. If GPIO pins 87 to 84 are pulled high “1111” and a jump to address 0x33 FFF6 is performed. This address is called “the Flash entry point”, which is an empty 2-word memory space. One of our tasks in preparation to use the Flash is to add a jump instruction to this two-word space. If we use a project based on the C language, we have to jump to the C start-up function “c\_int00”, which is part of the runtime library “rts2800\_ml.lib”.

**CAUTION:** Do never exceed the two word memory space for this step. Addresses 0x33 FFF8 to 0x33 FFFF are reserved for the password area!!

4. Function “c\_int00” performs initialization routines for the C-environment and global variables. For this module, we will have to place this function into a specific Flash section.
5. At the very end, “c\_int00” branches to our C-function called “main()”, which also must be loaded into a flash section.

## F2833x Flash Memory Sectors

<b>TMS320F28335 Flash Memory Map</b>	
Address Range	Data & Program Space
0x30 0000 – 0x30 7FFF	Sector H; 32K x 16
0x30 8000 – 0x30 FFFF	Sector G; 32K x 16
0x31 0000 – 0x31 7FFF	Sector F; 32K x 16
0x31 8000 – 0x31 FFFF	Sector E; 32K x 16
0x32 0000 – 0x32 7FFF	Sector D; 32K x 16
0x32 8000 – 0x32 FFFF	Sector C; 32K x 16
0x33 0000 – 0x33 7FFF	Sector B; 32K x 16
0x33 8000 – 0x33 FF7F	Sector A; (32K-128) x 16
0x33 FF80 – 0x33 FFF5	Program to 0x0000 when using Code Security Mode !
0x33 FFF6 – 0x33 FFF7	Flash Entry Point; 2 x 16
0x33 FFF8 – 0x33 FFFF	Security Password; 8 x 16

14 - 3

The 256k x 16 bit Flash is divided into 8 groups called “sectors”. Each sector can be programmed independently from the others. Please note that the highest 128 addresses of sector A (0x33FF80 to 0x33 FFFF) are not available for general purpose. Lab 14 will use sections A and D.

## Flash Speed Initialization

To derive the highest possible speed for the execution of our code we have to initialize the number of wait states that are added when the Flash area is accessed. When we start the F2833x out of RESET, the number of wait states defaults to 16. Wait states are additional clock cycles, that extend the FLASH - access period. For our tiny lab exercises, this extension is of no significance, but when you work on a real-world project, where computing power is so important, it would be a shame not to make best use of these wait states. So let us assume that our lab examples are ‘real’ projects and that we want to use the maximum frequency for the Flash. But why do we not initialize the wait states down to zero? Well, the number of wait states is related to the operational speed of the FLASH memory. According to the data-sheet of the F2833x there is a limit for the minimum number of wait states. For the current silicon revision of the F2833x this limit is set to 5 for a 150MHz device.

## Basic Flash Operation

- ◆ Flash is arranged in pages of 128 addresses
- ◆ Wait states are specified for consecutive accesses within a page, and random accesses across pages
- ◆ OTP has random access only
- ◆ Must specify the number of SYSCLKOUT wait-states
  - ◆ *Reset defaults are maximum values!*
- ◆ Flash configuration code must not run from Flash memory!



\*\*\* Refer to the F2833x datasheet for detailed numbers \*\*\*

For 150 MHz, PAGEWAIT = 5, RANDWAIT = 5, OTPWAIT = 8

For 100 MHz, PAGEWAIT = 3, RANDWAIT = 3, OTPWAIT = 5

14 - 4

There are two bit-fields in the “FBANKWAIT” register that are used to specify the number of wait states – PAGEWAIT and RANDWAIT. Consecutive page accesses are performed within an area of 128 addresses whereas a sequence of random accesses is performed in any order of addresses. So how fast is the F2833x running out of Flash or, in computer language: How many millions of instructions (MIPS) is the F2833x doing?

Answer:

The F2833x executes one instruction (a 16-bit word) in 1 cycle. Adding the 5 wait states we end up with:

$$1 \text{ instruction} / 6 \text{ cycles} * 150\text{MHz} = 25 \text{ MHz.}$$

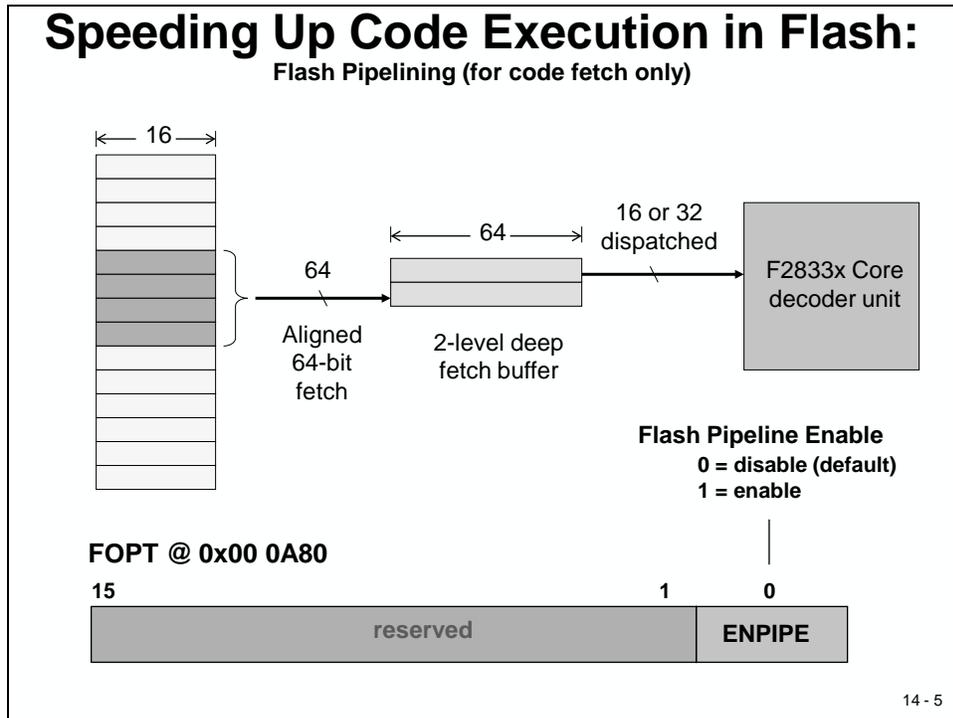
For a one-cycle instruction machine like the F2833x, the 25 MHz translate into 25MIPS. This is pretty slow compared to the original system frequency of 150 MHz! Is this all we can expect from Texas Instruments? No! The hardware solution is called a “pipeline”, which is shown in next slide!

Instead of reading only one 16-bit instruction from Flash code memory, Texas Instruments has implemented a 64-bit access – reading up to 4 instructions in 1+5 cycles. This leads to the final estimation of the speed of the internal Flash:

$$4 \text{ instructions} / 6 \text{ cycles} * 150 \text{ MHz} = 100 \text{ MHz.}$$

Using the Flash code Pipeline, the real **Flash speed is 100 MIPS!**

To use the Flash pipelining code fetch method we have to set bit “ENPIPE” to 1 to enable pipeline operations. By default after RESET, this feature is disabled.



## Flash Configuration Registers

There are some more registers to control the timing and operation modes of the F2833x internal Flash memory. For our lab exercise and most of the ‘real’ F2833x applications, it is sufficient to use the default values after RESET.

Texas Instruments provides an initialization function for the internal Flash, called “**InitFlash()**”. This function is part of file “DSP2833x\_SysCtrl.c” of the Peripheral Register Header Files that we have already used in our previous labs. We will use this function in our coming lab exercise.

### Other Flash Configuration Registers

Address	Name	Description
0x00 0A80	FOPT	Flash option register
0x00 0A82	FPWR	Flash power modes registers
0x00 0A83	FSTATUS	Flash status register
0x00 0A84	FSTDBYWAIT	Flash sleep to standby wait register
0x00 0A85	FACTIVEWAIT	Flash standby to active wait register
0x00 0A86	FBANKWAIT	Flash read access wait state register
0x00 0A87	FOTPWAIT	OTP read access wait state register

- ◆ **FPWR:** Save power by putting Flash/OTP to ‘Sleep’ or ‘Standby’ mode; Flash will automatically enter active mode if a Flash/OTP access is made
- ◆ **FSTATUS:** Various status bits (e.g. PWR mode)
- ◆ **FSTDBYWAIT:** Specify number of cycles to wait during wake-up from sleep to standby
- ◆ **FACTIVEWAIT:** Specify number of cycles to wait during wake-up from standby to active

Defaults for these registers are often sufficient – See “*TMS320F2833x System Control and Interrupts Reference Guide*,” *SPRUFB0*, for more information

14 - 6

## Flash Programming Procedure

The procedure to load a portion of code into the Flash is not as simple as loading a program into the internal RAM. Recall that Flash is non-volatile memory. Flash is based on a floating gate technology. To store a binary 1 or 0 this gate must load / unload electrons. The term “Floating Gate” means this is an isolated gate, with no electrical connections. Two effects are used to force electrons into this gate: ‘Hot electron injection’ or ‘electron tunnelling’ performed by a charge pump on board of the F2833x.

But how do we get the code into the internal Flash?

The F2833x itself will take care of the Flash programming procedure. Texas Instruments provides the code to execute the sequence of actions. The Flash Utility code can be applied using two basic options:

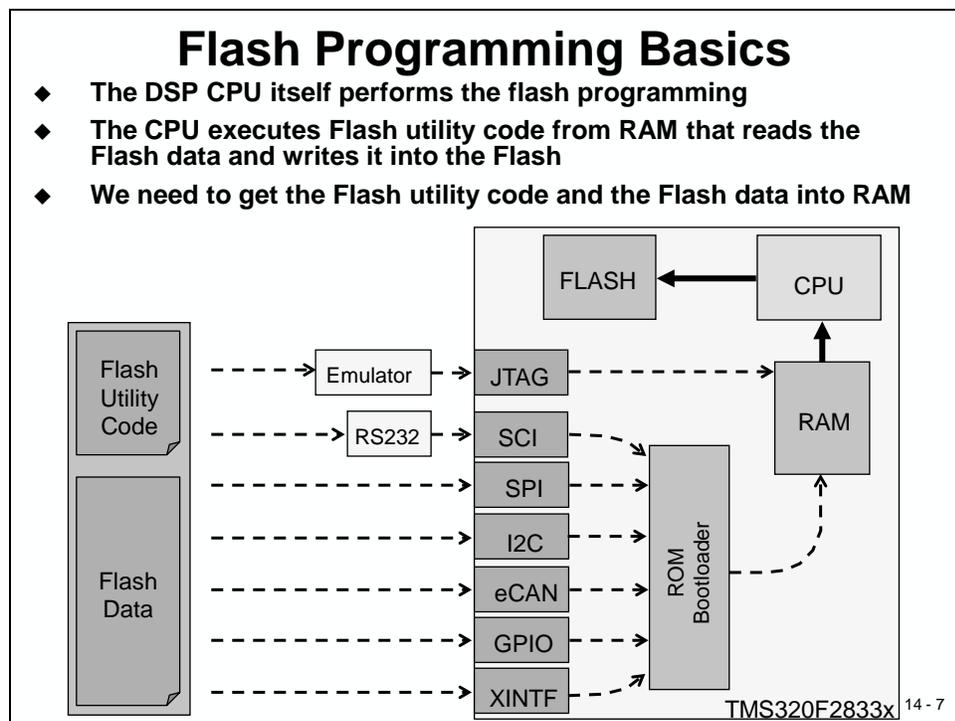
1. Code Composer Studio integrated tool

➔ **Tools ➔ On Chip Flash**

2. Download both the Flash Utility code and the Flash Data via one of the boot loader options of the F2833x.

For our lab we will use the CCS-Tool.

Please note that the Flash Utility code must be executed from a SARAM portion of the F2833x.



The steps “Erase” and “Program” to program the Flash are mandatory; “Verify” is an option but is highly recommended.

## Flash Programming Basics

### ◆ Sequence of steps for Flash programming:

Algorithm	Function
1. Erase	- Set all bits to zero, then to one
2. Program	- Program selected bits with zero
3. Verify	- Verify flash contents

- ◆ **Minimum Erase size is a sector**
- ◆ **Minimum Program size is a bit!**
- ◆ **Important not to lose power during erase step: If CSM passwords happen to be all zeros, the CSM will be permanently locked!**
- ◆ **Chance of this happening is quite small! (Erase step is performed sector by sector)**

14 - 8

## Flash Programming Utilities

- ◆ **Code Composer Studio Plug-in (uses JTAG)**
- ◆ **Third-party JTAG utilities**
  - SDFlash JTAG from Spectrum Digital (requires SD emulator)
  - Signum System Flash utilities (requires Signum emulator)
  - BlackHawk Flash utilities (requires Blackhawk emulator)
- ◆ **SDFlash Serial utility (uses SCI boot)**
- ◆ **Gang Programmers (use GPIO boot)**
  - BP Micro programmer
  - Data I/O programmer
- ◆ **Build your own custom utility**
  - Use a different ROM bootloader method than SCI
  - Embed flash programming into your application
  - Flash API algorithms provided by TI

\* TI web has links to all utilities (<http://www.ti.com/c2000>)

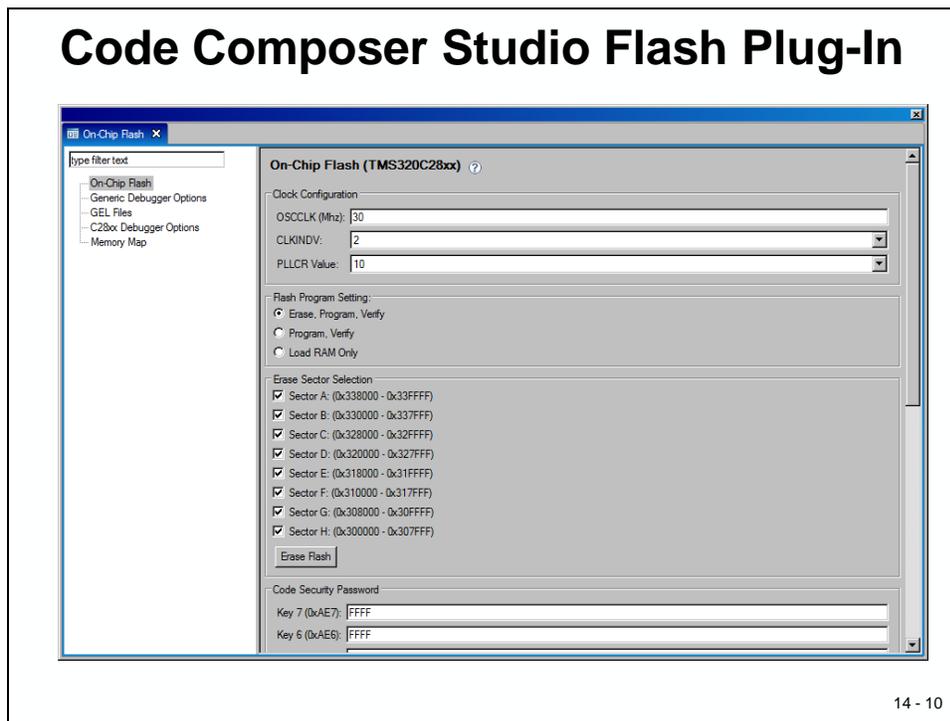
14 - 9

## CCS Flash Plug-In

The Code Composer Studio Flash Plug-in is called from:

→ Tools → On Chip Flash

It opens with the following window:



First verify that the OSCCLK is set to:

- 30MHz and PLLCR – value to 10 for a F28335ControlCard @ 30MHz or
- 20MHz and PLLCR – value to 10 for a F28335ControlCard @ 20MHz

The resulting SYSCLKOUT frequency is either 150 or 100MHz. Please make sure to use the correct numbers, which are equivalent to the physical set up of your F28335ControlCard.

**NEVER use the buttons “Program Password” or “LOCK”!**

Leave all 8 entries for Key 0 to Key 7 filled with “FFFF”.

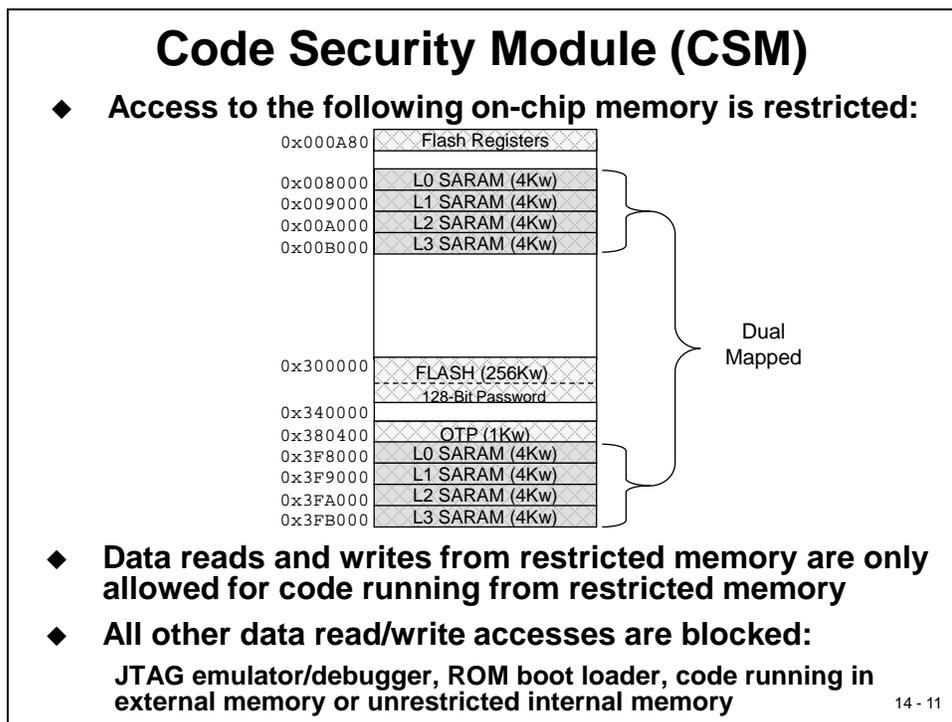
On the top right-hand side, we can exclude some of the sectors from being erased.

The lower right side is the Operation part of the window. First we have to specify the name of the projects out-file. The Plug-In extracts all the information needed to program the Flash from this COFF- File.

Before you start the programming procedure, it is highly recommended that you inspect the linker map-file (\*.map) in the “Debug”-Subfolder. This file provides a statistical view of the usage of the different Flash sections by your project. Verify that all sections are used as expected.

## Code Security Mode

Before we continue with our next lab, let us first discuss the Code Security feature of the F2833x. As mentioned earlier in this module, dedicated areas of memory are password protected. This is valid for memory L0, L1, L2, L3, OTP and Flash.



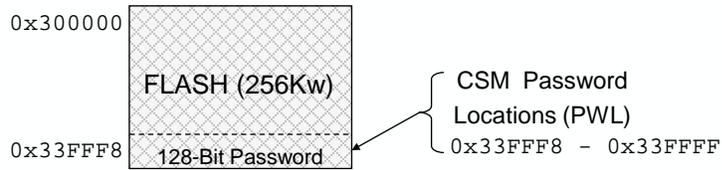
Once a password is applied, a data read or write operation from/to restricted memory locations is only allowed from code in restricted memory. All other accesses, including accesses from code running from external or unrestricted internal memories as well as JTAG access attempts are denied.

As mentioned earlier, the password is located in address space 0x33 FFF8 to 0x33 FFFF and has a field size of 128-bits. The 8 key registers (Key0 to Key7) are used to allow an access to a locked device. All you need to do is to write the correct password sequence in Key 0 -7 (address space 0x00 0AE0 – 0x00 0AE7).

The password area filled with 0xFFFF in all 8 words is equivalent to an unsecured device.

The password area filled with 0x0000 in all 8 words locks the device **FOREVER!**

## CSM Password



- ◆ 128-bit user defined password is stored in Flash
- ◆ 128-bit KEY registers are used to lock and unlock the device
  - ◆ Mapped in memory space 0x00 0AE0 – 0x00 0AE7
  - ◆ Registers “EALLOW” protected

14 - 12

## CSM Registers

Key Registers – accessible by user; EALLOW protected

Address	Name	Reset Value	Description
0x00 0AE0	KEY0	0xFFFF	Low word of 128-bit Key register
0x00 0AE1	KEY1	0xFFFF	2 <sup>nd</sup> word of 128-bit Key register
0x00 0AE2	KEY2	0xFFFF	3 <sup>rd</sup> word of 128-bit Key register
0x00 0AE3	KEY3	0xFFFF	4 <sup>th</sup> word of 128-bit Key register
0x00 0AE4	KEY4	0xFFFF	5 <sup>th</sup> word of 128-bit Key register
0x00 0AE5	KEY5	0xFFFF	6 <sup>th</sup> word of 128-bit Key register
0x00 0AE6	KEY6	0xFFFF	7 <sup>th</sup> word of 128-bit Key register
0x00 0AE7	KEY7	0xFFFF	High word of 128-bit Key register
0x00 0AEF	CSMSCR	0xFFFF	CSM status and control register

PWL in memory – reserved for passwords only

Address	Name	Reset Value	Description
0x33 7FF8	PWL0	user defined	Low word of 128-bit password
0x33 7FF9	PWL1	user defined	2 <sup>nd</sup> word of 128-bit password
0x33 7FFA	PWL2	user defined	3 <sup>rd</sup> word of 128-bit password
0x33 7FFB	PWL3	user defined	4 <sup>th</sup> word of 128-bit password
0x33 7FFC	PWL4	user defined	5 <sup>th</sup> word of 128-bit password
0x33 7FFD	PWL5	user defined	6 <sup>th</sup> word of 128-bit password
0x33 7FFE	PWL6	user defined	7 <sup>th</sup> word of 128-bit password
0x33 7FFF	PWL7	user defined	High word of 128-bit password

14 - 13

## Locking and Unlocking the CSM

- ◆ **The CSM is locked at power-up and reset**
- ◆ **To unlock the CSM:**
  - ◆ Perform a dummy read of each password in the Flash
  - ◆ Write the correct passwords to the key registers
- ◆ **New Flash Devices (PWL are all 0xFFFF):**
  - ◆ When all passwords are 0xFFFF – only a read of the PWL is required to bring the device into unlocked mode

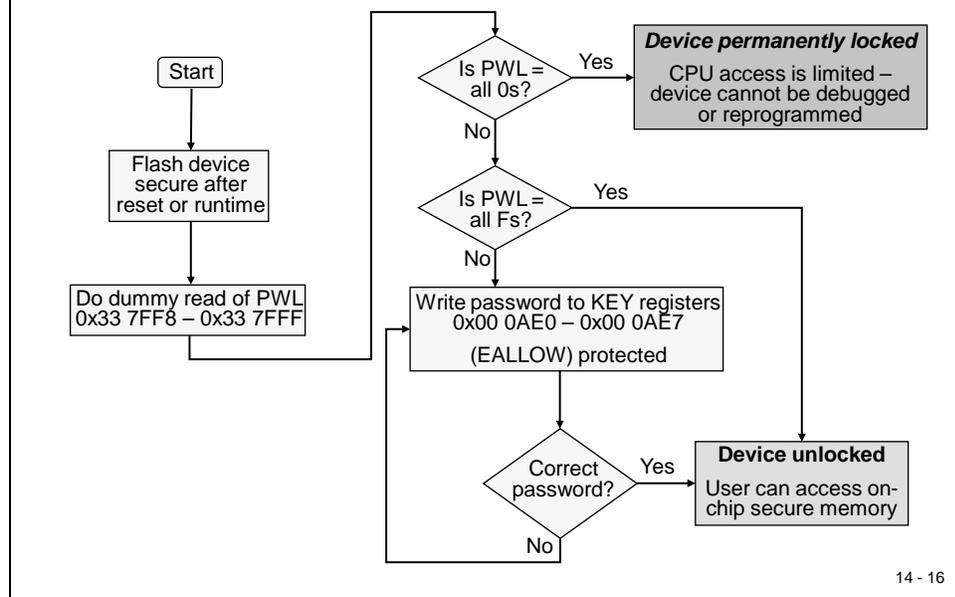
14 - 14

## CSM Caveats

- ◆ **Never program all the PWL's as 0x0000**
  - ◆ *Doing so will permanently lock the CSM*
- ◆ **Flash addresses 0x337F80 to 0x337FF5, inclusive, must be programmed to 0x0000 to securely lock the CSM**
- ◆ **Remember that code running in unsecured RAM cannot access data in secured memory**
  - ◆ Don't link the stack to secured RAM if you have any code that runs from unsecured RAM
- ◆ **Do not embed the passwords in your code!**
  - ◆ Generally, the CSM is unlocked only for debug
  - ◆ Code Composer Studio can do the unlocking

14 - 15

## CSM Password Match Flow



14 - 16

## CSM C-Code Examples

### Unlocking the CSM:

```

volatile int *PWL = &CsmPwl.PSWD0; //Pointer to PWL register file
volatile int i, tmp;

for (i = 0; i<8; i++) tmp = *PWL++; //Dummy reads of PWL locations

asm (" EALLOW"); //KEY regs are EALLOW protected
CsmRegs.KEY0 = PASSWORD0; //Write the passwords
CsmRegs.KEY1 = PASSWORD0; //to the Key registers
CsmRegs.KEY2 = PASSWORD2;
CsmRegs.KEY3 = PASSWORD3;
CsmRegs.KEY4 = PASSWORD4;
CsmRegs.KEY5 = PASSWORD5;
CsmRegs.KEY6 = PASSWORD6;
CsmRegs.KEY7 = PASSWORD7;
asm (" EDIS");
  
```

### Locking the CSM:

```

asm(" EALLOW"); //CSMSCR reg is EALLOW protected
CsmRegs.CSMSCR.bit.FORCESEC = 1; //Set FORCESEC bit
asm ("EDIS");
  
```

14 - 17

## Lab Exercise 14: Standalone Project

### Lab 14: Load an application into Flash

- Use Solution for Lab6 to begin with
- Modify the project to use internal Flash for code
- Add “DSP2833x\_CodeStartBranch.asm” to branch from Flash entry point ( 0x33 FFF6) to C - library function “\_c\_int00”
- Add TI - code to set up the speed of Flash
- Add a function to move the speed-up code from Flash to SARAM Adjust Linker Command File
- Use CCS plug-in tool to perform the Flash download
- Disconnect emulator and re-power the board!
- Code should be executed out of Flash
- For details see procedure in textbook!

14 - 18

## Objective

The objective of this laboratory exercise is to practice working with the F2833x internal Flash Memory. Let us assume your task is to prepare one of your previous laboratory solutions to run as a stand-alone solution, direct from Flash memory after powering up the F2833x. You can select any of your existing solutions, but to keep it easier for your supervisor to assist you during the debug phase let us take the “binary counter” (Lab 6) as the starting point.

What do we have to modify?

In Lab 6 the code was loaded by CCS via the JTAG-Emulator into L1-SARAM after a successful build operation. The linker command file “28335\_RAM\_lnk.cmd” took care of the correct connection of the code sections to physical memory addresses of L1-SARAM. Obviously, we will have to modify this part. Instead of editing the command file, we will use another one (“F28335.cmd”), also provided by Texas Instruments header file package.

In addition, we will have to fill in the Flash entry point address with a connection to the C environment start function (“c\_int00”). Following a RESET, the Flash memory itself operates with the maximum number of wait states – our code should reduce this number of wait states to gain the highest possible speed for Flash operations. Unfortunately we cannot call this speed-up function when it is still located in Flash – we will have to copy this function temporarily into any code SARAM before we can call it.

Finally we will use Code Composer Studio’s Flash Programming plug in tool to load our code into Flash.

**Please recall the explanations about the Code Security Module in this lesson, be aware of the password feature all the time in this lab session and do NOT program the password area!**

There are several things to take into account in this lab session, so as usual, let us use a procedure to prepare the project.

## Procedure

### Open Files, Create Project File

1. Create a new project, called **Lab14.pjt** in C:\DSP2833x\Labs.
2. Open the file “Lab6.c” from C:\DSP2833x\_V4\Labs\Lab6 and save it as “Lab14.c” in C:\DSP2833x\_V4\Labs\Lab14.
3. Define the size of the C system stack. In the project window, right click at project “Lab14” and select “Properties”. In category “C/C++ Build”, “C2000 Linker”, “Basic Options” set the C stack size to 0x400.

Link some of the source code files, provided by Texas Instruments, to the project:

4. In the C/C++ perspective, right click at project “Lab14” and select “**Link Files to Project**”. Go to folder “C:\tidcs\c28\dsp2833x\v131\DSP2833x\_headers\source” and link:

- **DSP2833x\_GlobalVariableDefs.c**

From C:\tidcs\c28\dsp2833x\v131\DSP2833x\_common\source link:

- **DSP2833x\_PieCtrl.c**
- **DSP2833x\_PieVect.c**
- **DSP2833x\_DefaultIsr.c**
- **DSP2833x\_CpuTimers.c**
- **DSP2833x\_SysCtrl.c**
- **DSP2833x\_CodeStartBranch.asm**
- **DSP2833x\_ADC\_cal.asm**
- **DSP2833x\_usDelay.asm**

From C:\tidcs\c28\dsp2833x\v131\DSP2833x\_headers\cmd link:

- **DSP2833x\_Headers\_nonBIOS.cmd**

From C:\tidcs\c28\dsp2833x\v131\DSP2833x\_common\cmd link:

- **F28335.cmd**

Exclude the file “F28335\_RAM\_Ink.cmd from the project

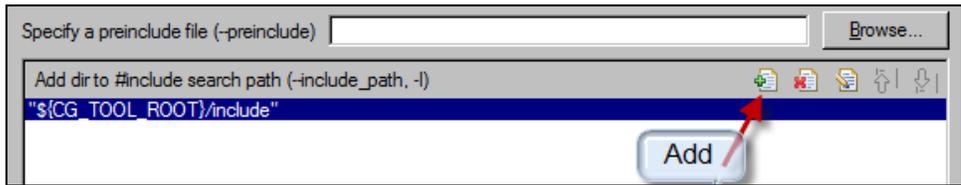
## Project Build Options

5. We have to extend the search path of the C-Compiler for include files. Right click at project “Lab13” and select “Properties”. Select “C/C++ Build”, “C2000 Compiler”, “Include Options”. In the box: “Add dir to #include search path”, add the following lines:

**C:\tidcs\C28\dsp2833x\v131\DSP2833x\_headers\include**

**C:\tidcs\c28\DSP2833x\v131\DSP2833x\_common\include**

Note: Use the “Add” Icon to add the new paths:



Close the Property Window by Clicking <OK>.

## Add Additional Source Code Files

6. To add the machine code for the Flash entry point at address 0x33 FFF6, we have to add an assembly instruction “LB \_c\_int00” and to link this instruction exactly to the given physical address. Instead of writing our own assembly code, we can make use of another of TI’s predefined functions (“code\_start”), which is part of the source code file “DSP2833x\_CodeStartBranch.asm”.

From *C:\tidcs\c28\dsp2833x\v131\DSP2833x\_common\source* we have already linked to our project:

- **DSP2833x\_CodeStartBranch.asm**

If you open the linker command file “F28335.cmd”, you will find a label “code\_start” linked to “BEGIN” which is defined at address 0x33 FFF6 in code memory page 0.

## Modify Source Code to Speed up Flash memory

7. Open file “Lab14.c” to edit.

In “main()”, after the function call “InitSysCtrl()”, we have to add the code to speed-up the Flash memory.

This will be done by the function “InitFlash()”. However, as mentioned earlier, this code must run out of SARAM. When we finally run the program from Flash and the F2833x reaches this line, all code is still located in Flash. This means that before we can call “InitFlash()”, the F2833x has to copy it from FLASH into SARAM. Standard ANSI-C provides a memory copy function “memcpy(\*dest,\*source, number)” for this purpose, the function prototype being in the file “string.h”.

What do we use for “dest” (destination address), “source” (source address) and “number” (number of elements to copy)?

Again, the solution can be found in the file “DSP2833x\_SysCtrl.c”. Open it and look at the beginning of this file. You will find a “#pragma CODE\_SECTION” – line that defines the dedicated code section “ramfuncs” and connects the function “InitFlash()” to it. The symbol “ramfuncs” is used in the file “F28335.cmd” to connect it to physical memory “FLASHD” as load-address and to memory “RAML0” as execution address. The task of the linker command file “F28335.cmd” is it to provide the physical addresses for the rest of the project. The symbols “LOAD\_START”, “LOAD\_END” and “RUN\_START” are used to define these addresses symbolically as “\_RamfuncsLoadStart”, “\_RamfuncsLoadEnd” and “\_RamfuncsRunStart”.

Add the following line to your code:

```
memcpy(&RamfuncsRunStart, &RamfuncsLoadStart,  
&RamfuncsLoadEnd - &RamfuncsLoadStart);
```

Add a call to the function “InitFlash()”, now available in RAML0:

```
InitFlash();
```

At the beginning of Lab14.c, add a function prototype for “InitFlash()”. Also declare the symbols used as parameters for “memcpy()” as externals:

```
extern unsigned int RamfuncsLoadStart;
```

```
extern unsigned int RamfuncsLoadEnd;
```

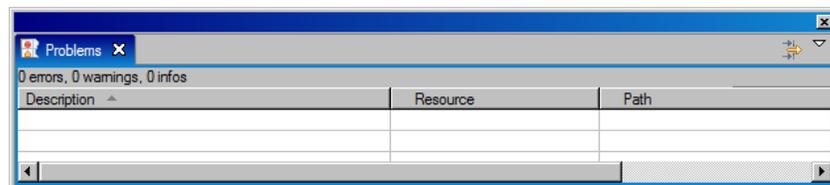
```
extern unsigned int RamfuncsRunStart;
```

## Build project

- Click the “Rebuild Active Project ” button or perform:

**Project → Rebuild Active Project (Alt +Shift + P)**

If the build was successful you should get 0 Errors, 0 Warnings and 0 Infos:



## Verify Linker Results: The “.map” - File

9. 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 “lab14.map”.
10. Open file “lab14.map” in the sub-folder “..\Debug”

In ‘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	0000001f	00000fe1	RWIX
FLASHD	00320000	00008000	0000001f	00007fe1	RWIX
FLASHA	00338000	00007f80	00000729	00007857	RWIX
BEGIN	0033fff6	00000002	00000002	00000000	RWIX
ADC_CAL	00380080	00000009	00000007	00000002	RWIX

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

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

The entry-point “codestart” connects the object “CodeStartBranch.obj” to physical address 0x3F FFF6 and occupies two words.

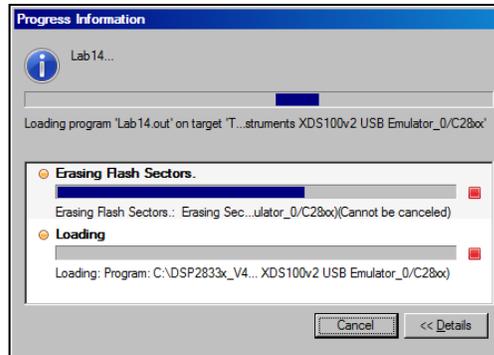
## Use CCS integrated Flash Programming

11. The next step is to program the machine code into the internal Flash. As mentioned in this lesson there are different ways to accomplish this step. The easiest way is to use the “Debug Active Project” feature of Code Composer Studio.

If there are FLASH based sections part of the project, they will be erased and programmed automatically!

Perform: ➔ **Target** ➔ **Debug Active Project**

If everything went as expected you should see these status messages:



**Congratulations!**

**Your code has been stored into FLASH – memory!**

In the “Debug” – Perspective open the disassembly window and enable “Show Source”

```

44      {
      main:
0x33840B: FE02      ADDB      SP,#2
0x33840C: 2B41      MOV       *-SP[1],#0
0x33840D: 76738316 LCR       InitSysCtrl
0x33840E:          memcopy(&RamfuncsRunStart, &RamfuncsLoadStart, &RamfuncsLoadEnd - &RamfuncsLoadSt
0x33840F: 8F720000 MOVL     XAR5,#0x320000
0x338411: 8F32001F MOVL     XAR4,#0x32001F
0x338413: A0A9      MOVL     @ACC,XAR5
0x338414: 564100A4 SUBL     @XAR4,ACC
0x338416: A8A9      MOVL     @ACC,XAR4
  
```

The blue arrow points the beginning of main. The address in the first column shows that the code has been loaded into a physical FLASH section (in the example above to address 0x33840B).

## Shut down CCS and Restart FLASH - Code

12. Close your Code Composer Studio session.
13. Disconnect the power from the Peripheral Explorer Board.
14. Verify that Peripheral Explorer Board jumper J3 (“SCI-BOOT GPIO84”) is open.
15. Reconnect Peripheral Explorer Board to power supply.

**Your code should be executed immediately out of Flash, showing the LED - binary counter at LEDs LD1...LD4.**

Blank page.