C28x Flash Programming

Introduction

So far we have used the C28x internal volatile memory (H0 – SARAM) to store the code of our examples. Before we could execute the code we used Code Composer Studio to load it into H0-SARAM (“File” ➔ “Load Program”). This is fine for projects in a development and debug phase with 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 C28x, 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 power-up of the system. 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 case of the F2812, we can add any of the memory to the control unit using the external interface (XINTF).

The F2812 is also equipped with an internal Flash memory area of 128Kx16. 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 out of Flash we have to go through a short explanation of how to use this memory. This module also covers the boot sequence of the C28x - what happens when we power on the C28x?

Chapter 10 also covers the password feature of the C28x code security module. This module is used to embed dedicated portions of the C28x memory in a secure section with a 128bit-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 for ever! If you decide to make your mark in your university by locking the device with your own password, be sure to have already passed all exams.

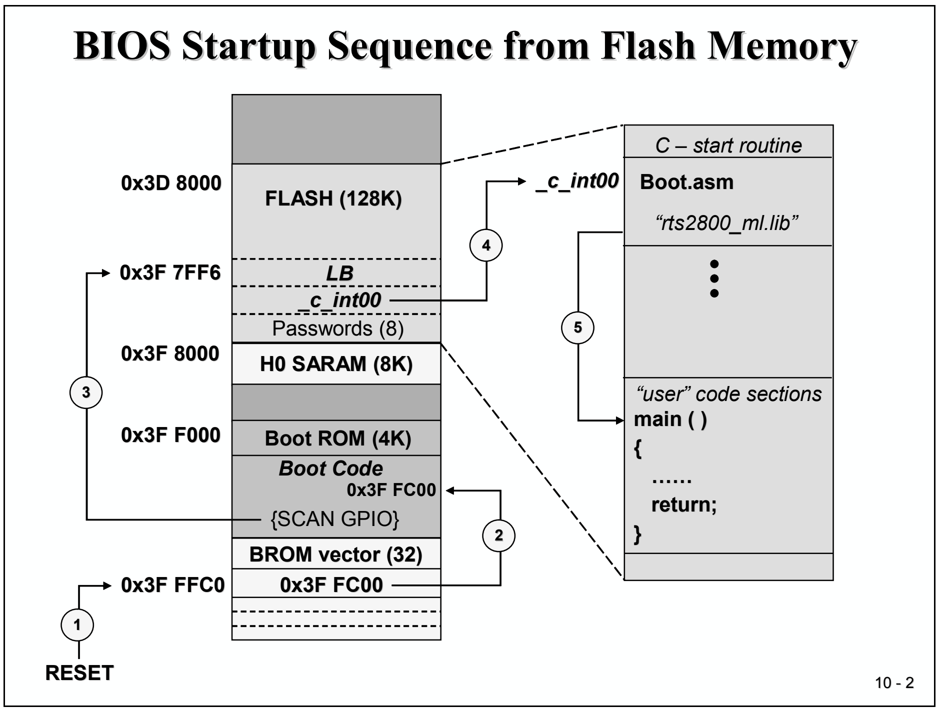
Module Topics

C28x Start-up Sequences

There are 6 different options to start the C28x out of power-on. The options are hard-coded by 4 GPIO-Inputs of Port F (F4, F12, F3 and F2). The 4 pins are sampled during power-on. Depending on the status one of the following options is selected:

F4	F12	F3	F2		
1	x	x	x	:	FLASH address 0x3F 7FF6 (see slide 10-2)
0	0	1	0	:	H0 – SARAM address 0x3F 8000
0	0	0	1	:	OTP address 0x3D 7800
0	1	x	x	:	boot load from SPI
0	0	1	1	:	boot load from SCI-A
0	0	0	0	:	boot load from parallel GPIO – Port B

To switch from H0-SARAM mode to Flash mode we have to change F4 from 0 to 1. At the eZdsp this is done using jumper JP7 (1-2 = Flash; 2-3 = H0-SARAM). Please note that the C28x must also run in Microcomputer-Mode (JP1 = 2-3). The following slide shows the sequence that takes place if we start from Flash.



BIOS Startup Sequence from Flash Memory

1. RESET-address is always 0x3F FFC0. This is part of TI’s internal BOOT-ROM.

2. BOOT-ROM executes a jump to address 0x3F FC00 (Boot Code). Here basic initialization tasks are performed and the type of the boot sequence is selected.

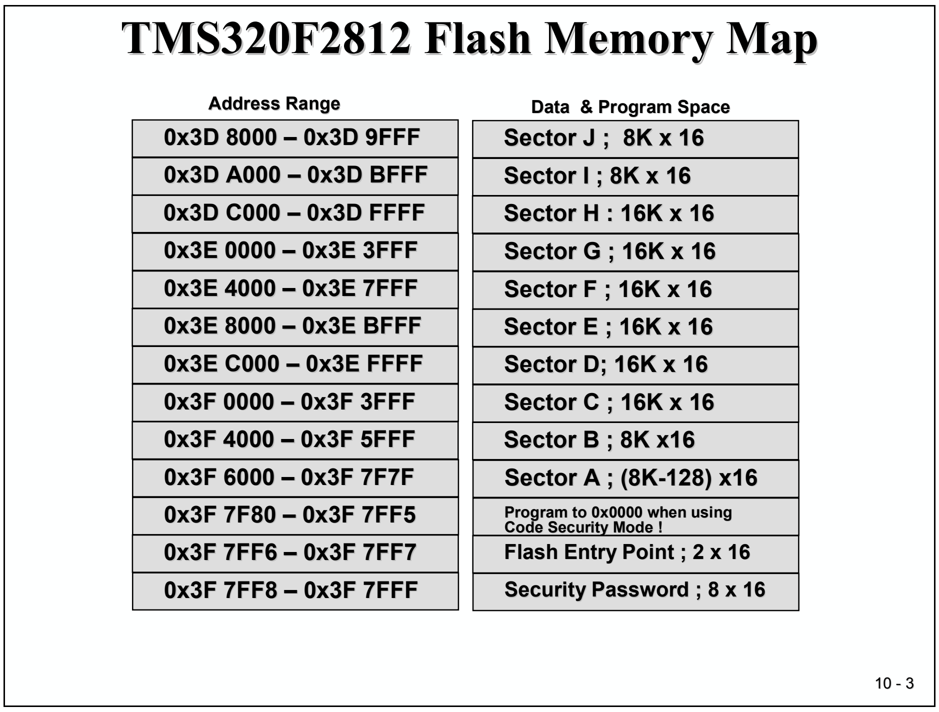
3. If GPIO-F4 = =1, a jump to address 0x3F 7FF6 is performed. This is the Flash-Entry-Point. It is only a 2 word memory space and this space is not filled yet. One of our tasks to use the Flash is to add a jump instruction into this two-word-space. If we use a project based on C language we have to jump 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 0x3F 7FF8 to 0x3F 7FFF 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 a function called “main”, which also must be loaded into a flash section.

C28x Flash Memory Sectors



TMS320F2812 Flash Memory Map

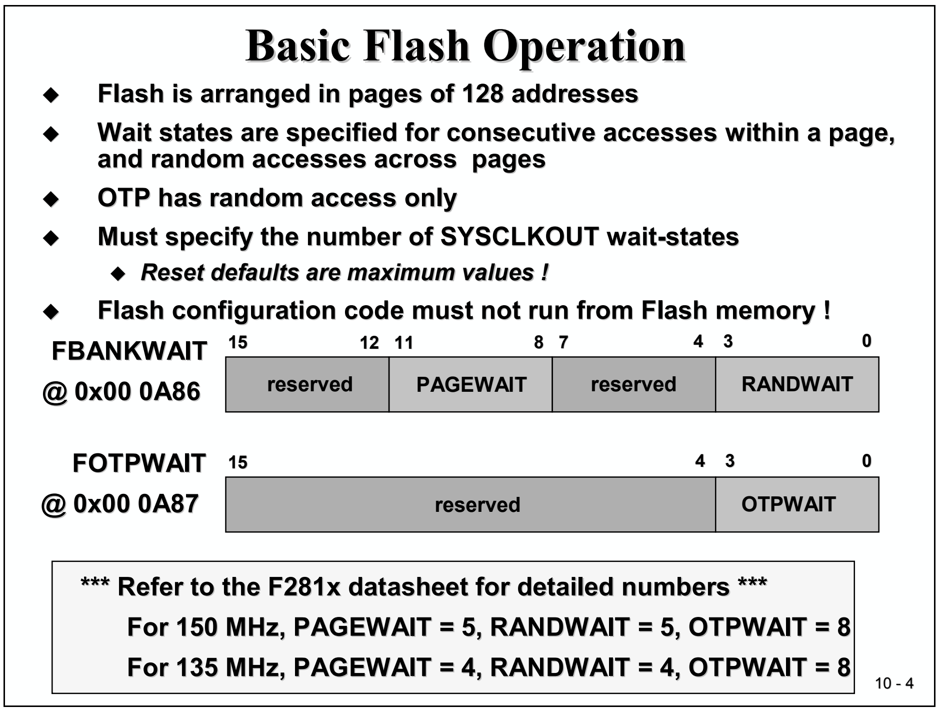
Address Range	Data & Program Space
0x3D 8000 – 0x3D 9FFF	Sector J ; 8K x 16
0x3D A000 – 0x3D BFFF	Sector I ; 8K x 16
0x3D C000 – 0x3D FFFF	Sector H : 16K x 16
0x3E 0000 – 0x3E 3FFF	Sector G ; 16K x 16
0x3E 4000 – 0x3E 7FFF	Sector F ; 16K x 16
0x3E 8000 – 0x3E BFFF	Sector E ; 16K x 16
0x3E C000 – 0x3E FFFF	Sector D; 16K x 16
0x3F 0000 – 0x3F 3FFF	Sector C ; 16K x 16
0x3F 4000 – 0x3F 5FFF	Sector B ; 8K x16
0x3F 6000 – 0x3F 7F7F	Sector A ; (8K-128) x16
0x3F 7F80 – 0x3F 7FF5	Program to 0x0000 when using Code Security Mode !
0x3F 7FF6 – 0x3F 7FF7	Flash Entry Point ; 2 x 16
0x3F 7FF8 – 0x3F 7FFF	Security Password ; 8 x 16

10 - 3

The 128k x 16 Flash is divided into 10 portions called “sectors”. Each sector can be programmed independently from the others. Please note that the highest 128 addresses of sector A (0x3F7F80 to 0x3F 7FFF) are not available for general purpose. Lab 11 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 is added when the Flash area is accessed. When we start the C28x out of RESET the number of wait states defaults to 16. For our tiny lab exercises this is of no significance, but when you think about real projects, where computing power is so important, it would be a shame not to make best use of these wait states. So let's assume that our lab examples are 'real' projects and that we want to use the maximum frequency for the Flash. So why not initialize the wait states to zero? According to the data-sheet of the C28x there is a limit for the minimum number of wait states. For silicon revision C this limit is set to 5 for a 150MHz C28x.



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 !

FBANKWAIT @ 0x00 0A86

15–12	11–8	7–4	3–0
reserved	PAGEWAIT	reserved	RANDWAIT

FOTPWAIT @ 0x00 0A87

15–4	3–0
reserved	OTPWAIT

*** Refer to the F281x datasheet for detailed numbers ***
For 150 MHz, PAGEWAIT = 5, RANDWAIT = 5, OTPWAIT = 8
For 135 MHz, PAGEWAIT = 4, RANDWAIT = 4, OTPWAIT = 8

10 - 4

There are two bit fields of register "FBANKWAIT" that are used to specify the number of wait states – PAGEWAIT and RANDWAIT. Consecutive page accesses are done within an area of 128 addresses whereas a sequence of random accesses is performed in any order of addresses. So how fast is the C28x running out of Flash or, in computer language: How many millions of instructions (MIPS) is the C28x doing?

Answer:

The C28x executes one instruction (a 16 bit word) in 1 cycle. Adding the 5 wait states we end up with

1 instruction / 6 cycles * 150MHz = 25 MHz.

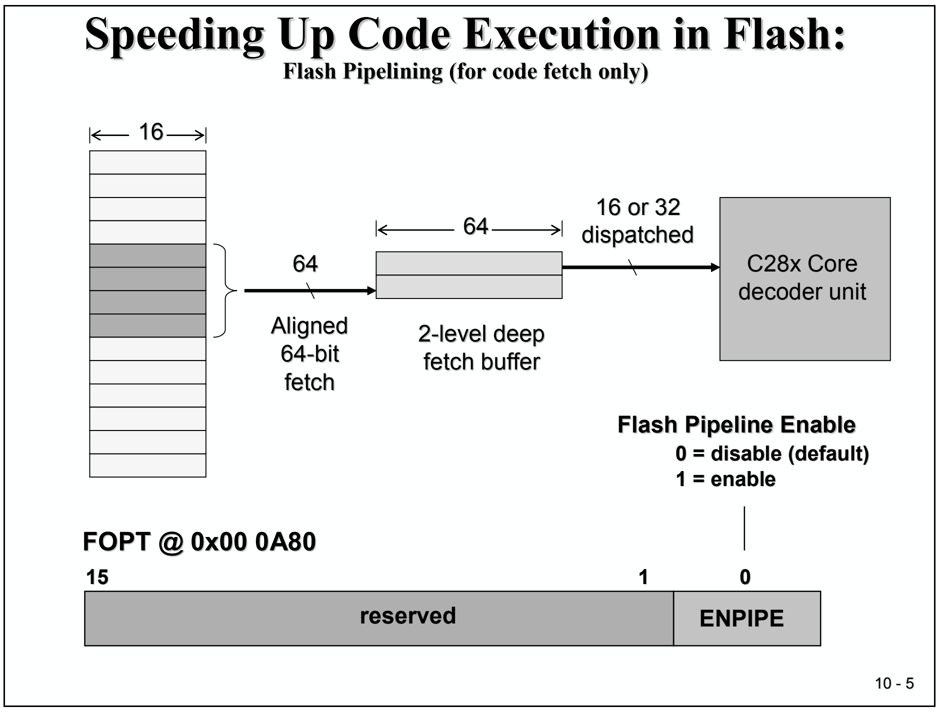
For a one-cycle instruction machine like the C28x, the 25 MHz translate into 25MIPS. This is pretty slow compared to the original system frequency of 150 MHz! Is this all we can expect from Texas Instruments? No! The hardware solution is called "pipeline", see next slide!

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

4 instructions / 6 cycles * 150 MHz = 100 MHz.

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

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



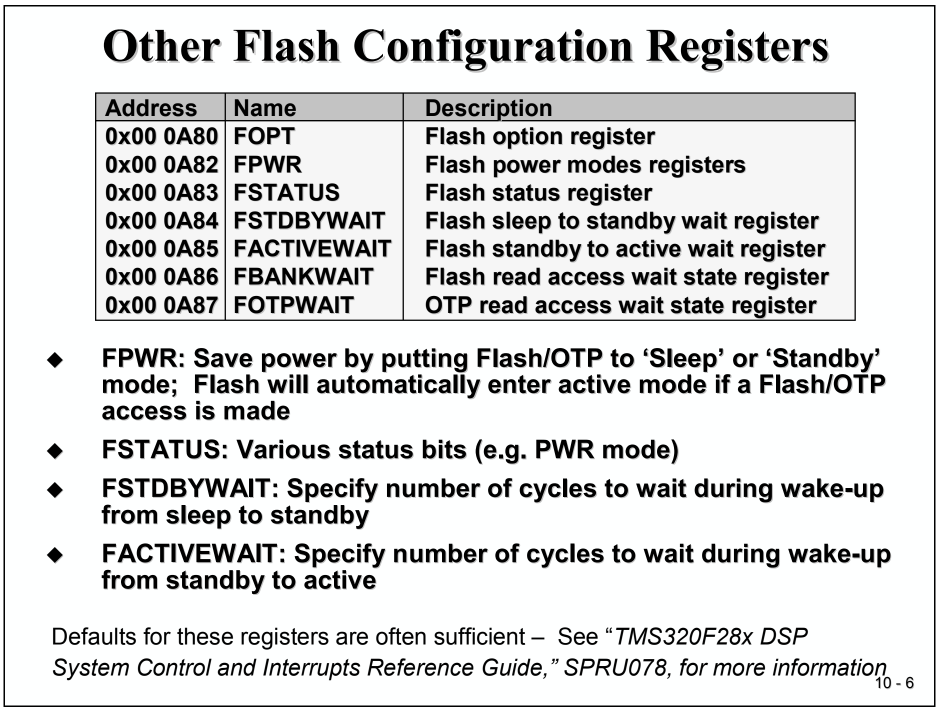
Speeding Up Code Execution in Flash:

Flash Pipelining (for code fetch only)

Flash Configuration Registers

There are some more registers to control the timing and operation modes of the C28x internal Flash memory. For our lab exercise and most of the 'real' C28x 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 the Peripheral Register Header Files, Version 1.00 that we already used in our previous labs. The source code of this function is part of file "**DSP281x_SysCtrl.c**". All we have to do to use this function in our coming lab is to add this source code file to our project.



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 "*TMS320F28x DSP System Control and Interrupts Reference Guide," SPRU078, for more information*

10 - 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. 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' done by a charge pump on board of the C28x.

How do we get the code into the internal Flash?

The C28x 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 in two basic options:

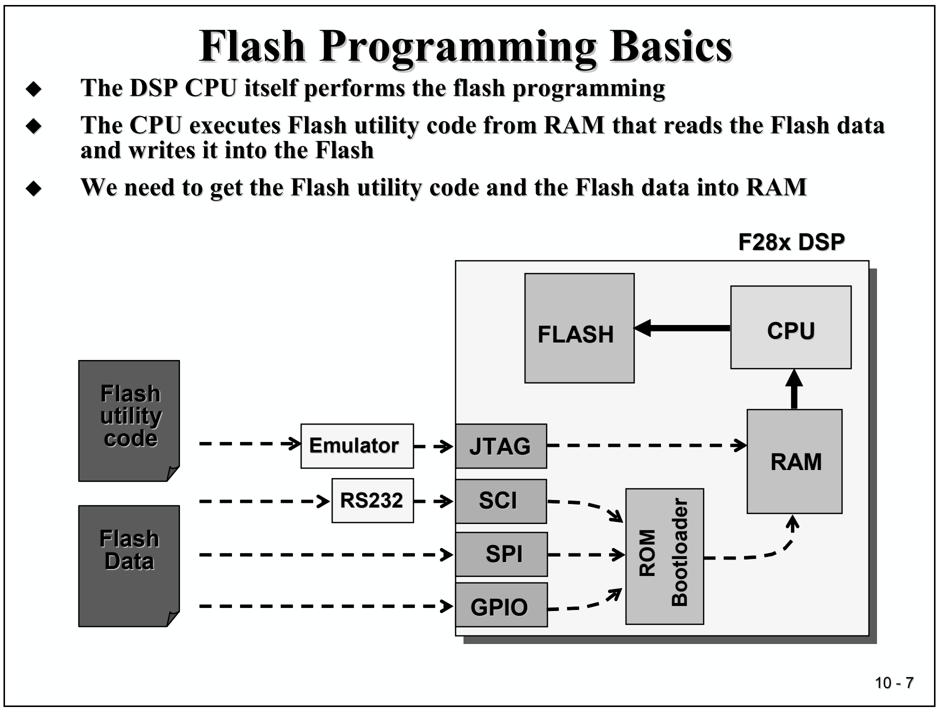
1. Code Composer Studio Plug-in Tool

   ➔ Tools ➔ F28xx On Chip Flash Programmer

2. Download both the Flash Utility code and the Flash Data via one of the 3 boot load options SCI-A, SPI or GPIO-B.

For our lab we will use the CCS-Tool.

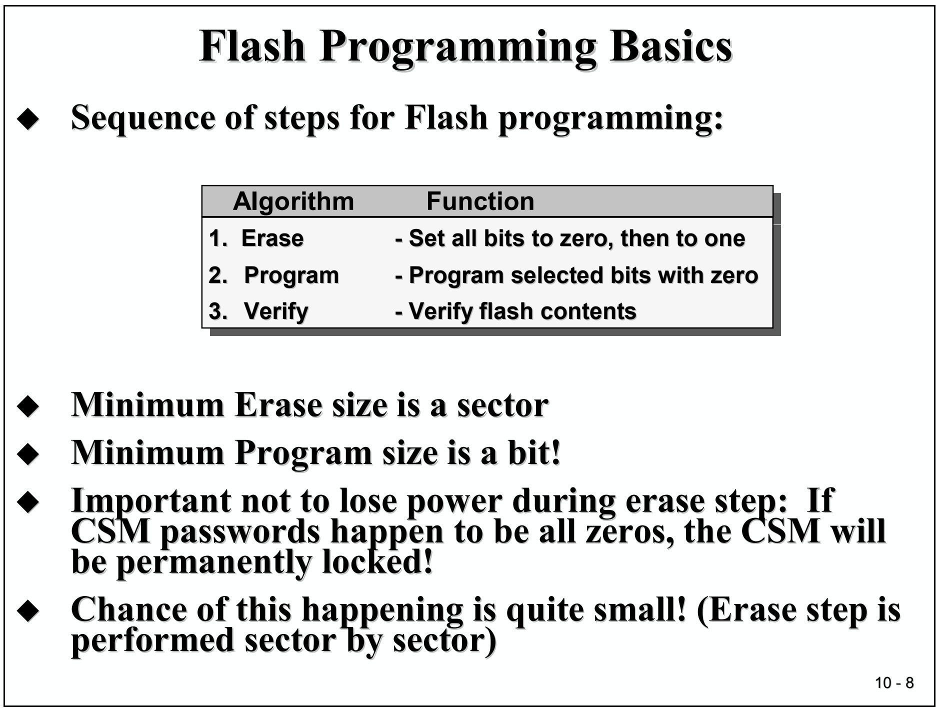
Please note that the Flash Utility code must be executed from a SARAM portion of the C28x.



Flash Programming Basics

- **The DSP CPU itself performs the flash programming**
- **The CPU executes Flash utility code from RAM that reads the Flash data and writes it into the Flash**
- **We need to get the Flash utility code and the Flash data into RAM**

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



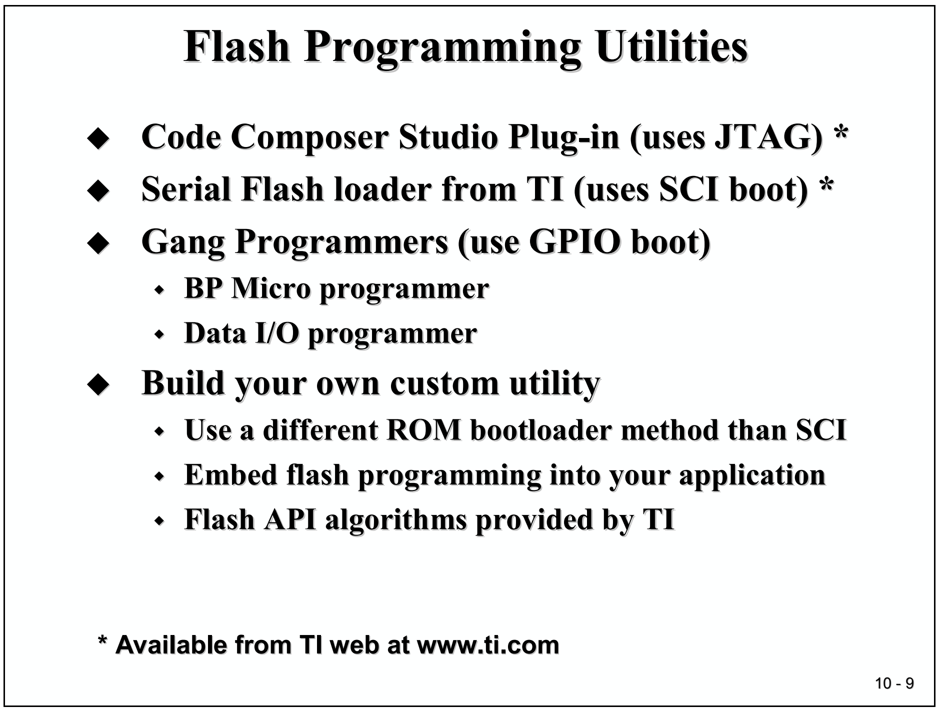
Flash Programming Basics

- 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)

10 - 8



Flash Programming Utilities

- Code Composer Studio Plug-in (uses JTAG) *
- Serial Flash loader from TI (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

* Available from TI web at www.ti.com

10 - 9

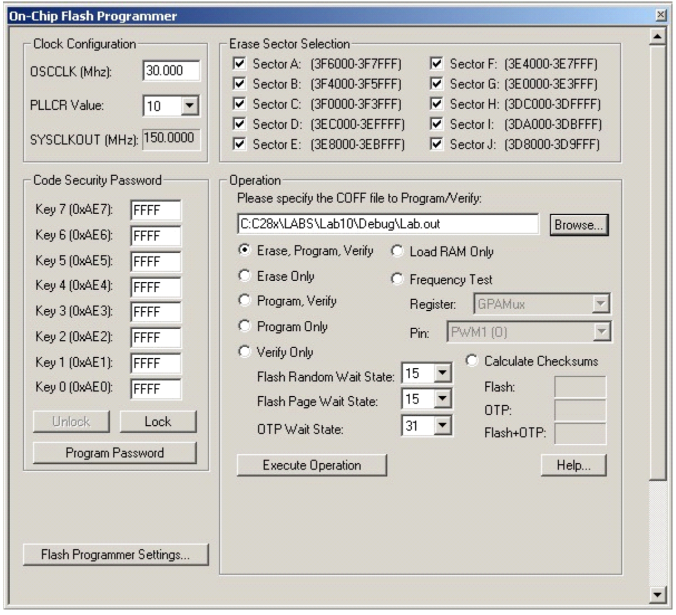
CCS Flash Plug-In

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

➔ Tools ➔ F28xx On Chip Flash Programmer

and opens with the following window:

Code Composer Studio Flash Plug-In



On-Chip Flash Programmer

Clock Configuration
- OSCCLK (Mhz): 30.000
- PLLCR Value: 10
- SYSCLKOUT (MHz): 150.0000

Erase Sector Selection
- [✓] Sector A: (3F6000-3F7FFF)
- [✓] Sector B: (3F4000-3F5FFF)
- [✓] Sector C: (3F0000-3F3FFF)
- [✓] Sector D: (3EC000-3EFFFF)
- [✓] Sector E: (3E8000-3EBFFF)
- [✓] Sector F: (3E4000-3E7FFF)
- [✓] Sector G: (3E0000-3E3FFF)
- [✓] Sector H: (3DC000-3DFFFF)
- [✓] Sector I: (3DA000-3DBFFF)
- [✓] Sector J: (3D8000-3D9FFF)

Code Security Password
- Key 7 (0xAE7): FFFF
- Key 6 (0xAE6): FFFF
- Key 5 (0xAE5): FFFF
- Key 4 (0xAE4): FFFF
- Key 3 (0xAE3): FFFF
- Key 2 (0xAE2): FFFF
- Key 1 (0xAE1): FFFF
- Key 0 (0xAE0): FFFF

Operation
Please specify the COFF file to Program/Verify:
C:C28x\LABS\Lab10\Debug\Lab.out

[✓ Erase, Program, Verify] Load RAM Only Erase Only Frequency Test Program, Verify Program Only Verify Only Calculate Checksums

- Register: GPAMux
- Pin: PWM1 (0)
- Flash Random Wait State: 15
- Flash Page Wait State: 15
- OTP Wait State: 31

10 - 10

First verify that the OSCCLK is set to 30MHz and the PLLCR to 10 which gives a SYSCLKOUT frequency of 150MHz. This is equivalent to the physical set up of the eZdsp2812.

NEVER use the buttons "Program Password" or "LOCK"!

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

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

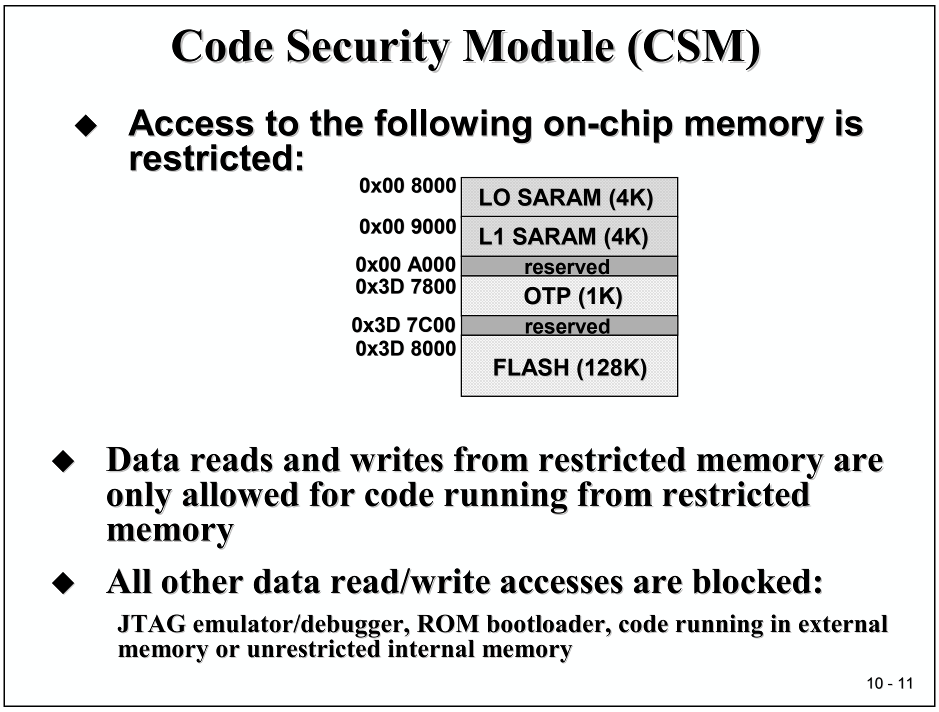
The lower right side is the command window. First we have to specify the name of the projects out-file. The Plug-In extracts all the information needed to program the Flash out of this COFF-File.

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

Start the programming sequence by clicking on "Execute Operation".

Code Security Mode

Before we go into our next lab let's discuss the Code Security feature of the C28x. As mentioned earlier in this module, dedicated areas of memory are password protected. This is valid for memory L0, L1, OTP and Flash.



Code Security Module (CSM)

- Access to the following on-chip memory is restricted:

Address	Memory
0x00 8000	LO SARAM (4K)
0x00 9000	L1 SARAM (4K)
0x00 A000	reserved
0x3D 7800	OTP (1K)
0x3D 7C00	reserved
0x3D 8000	FLASH (128K)

- **Data reads and writes from restricted memory are only allowed for code running from restricted memory**
- **All other data read/write accesses are blocked:**

  JTAG emulator/debugger, ROM bootloader, code running in external memory or unrestricted internal memory

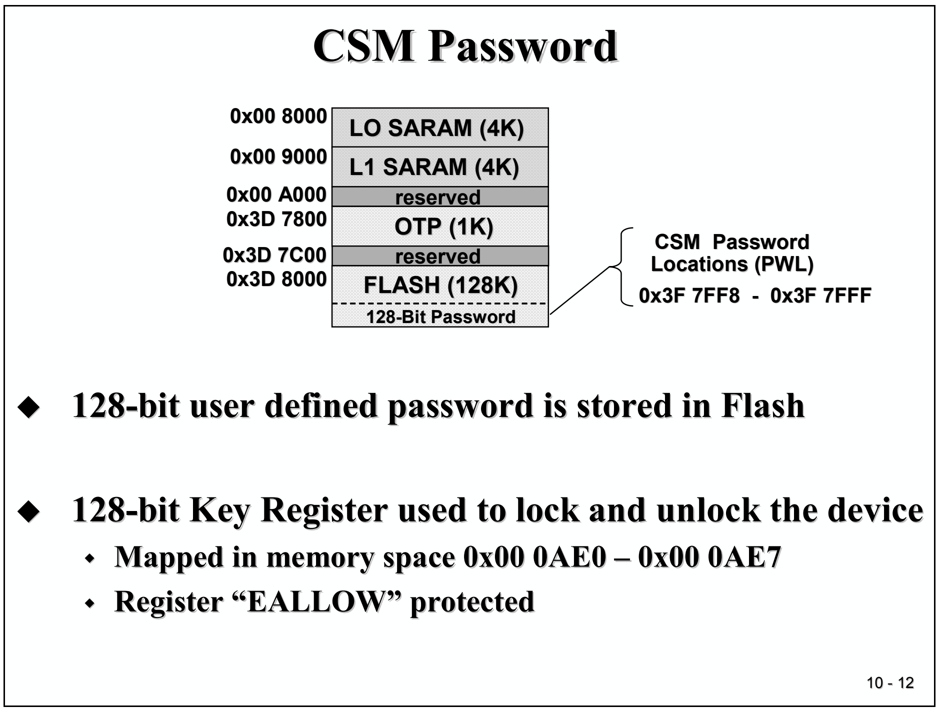
10 - 11

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.

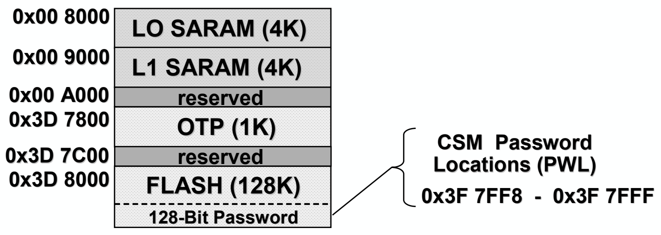
As mentioned earlier the password is located at address space 0x3F 7FF8 to 0x3F 7FFF and covers a 128-bit field. The 8 key registers (Key0 to Key7) are used to allow an access to a locked device. All you need to do is to write the correct password sequence in Key 0 -7 (address space 0x00 0AE0 – 0x00 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**!

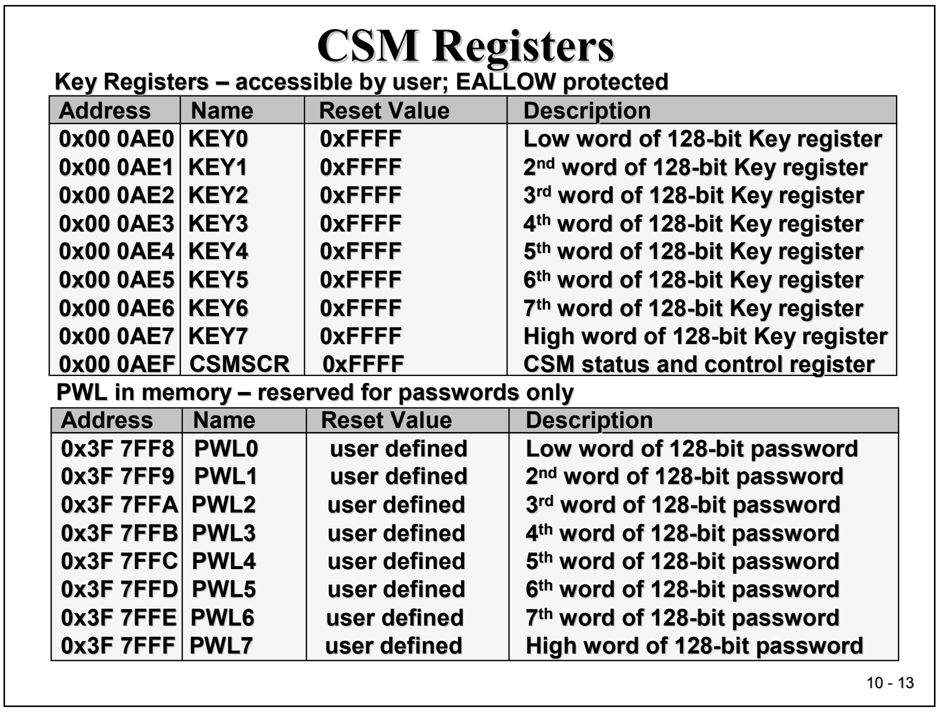


CSM Password



- **128-bit user defined password is stored in Flash**
- **128-bit Key Register used to lock and unlock the device**
  - **Mapped in memory space 0x00 0AE0 – 0x00 0AE7**
  - **Register "EALLOW" protected**

10 - 12



CSM Registers

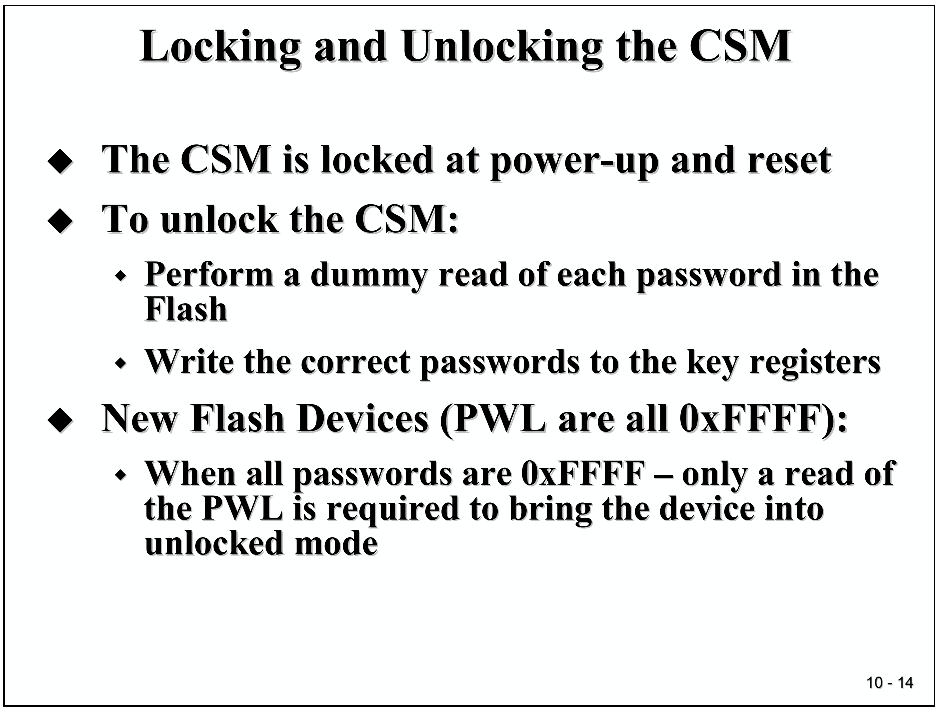
Key Registers – accessible by user; EALLOW protected

Address	Name	Reset Value	Description
0x00 0AE0	KEY0	0xFFFF	Low word of 128-bit Key register
0x00 0AE1	KEY1	0xFFFF	2nd word of 128-bit Key register
0x00 0AE2	KEY2	0xFFFF	3rd word of 128-bit Key register
0x00 0AE3	KEY3	0xFFFF	4th word of 128-bit Key register
0x00 0AE4	KEY4	0xFFFF	5th word of 128-bit Key register
0x00 0AE5	KEY5	0xFFFF	6th word of 128-bit Key register
0x00 0AE6	KEY6	0xFFFF	7th 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
0x3F 7FF8	PWL0	user defined	Low word of 128-bit password
0x3F 7FF9	PWL1	user defined	2nd word of 128-bit password
0x3F 7FFA	PWL2	user defined	3rd word of 128-bit password
0x3F 7FFB	PWL3	user defined	4th word of 128-bit password
0x3F 7FFC	PWL4	user defined	5th word of 128-bit password
0x3F 7FFD	PWL5	user defined	6th word of 128-bit password
0x3F 7FFE	PWL6	user defined	7th word of 128-bit password
0x3F 7FFF	PWL7	user defined	High word of 128-bit password

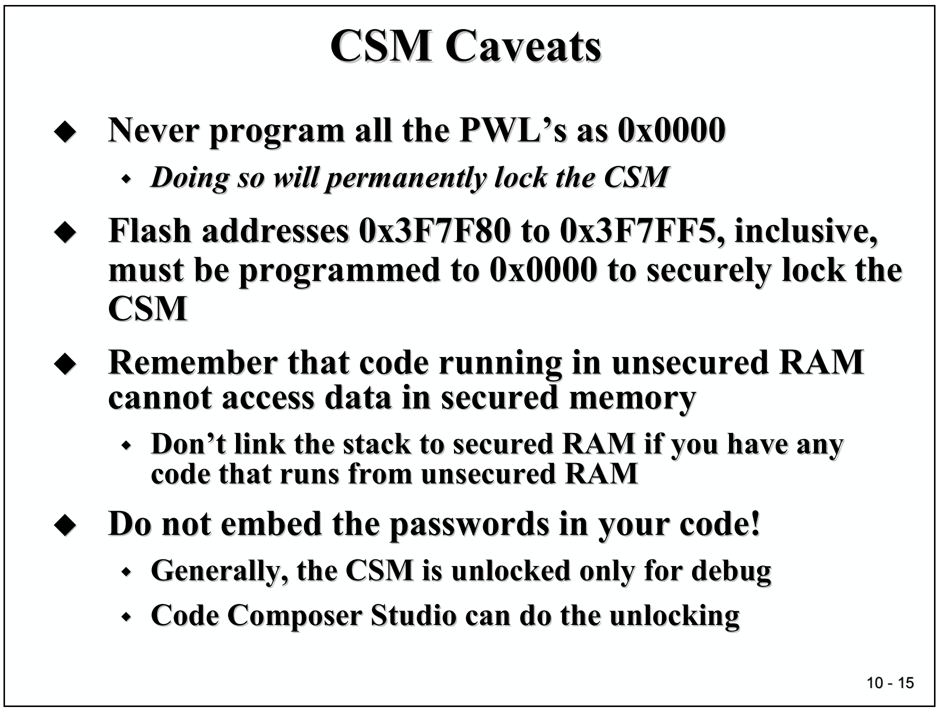
10 - 13



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**

10 - 14

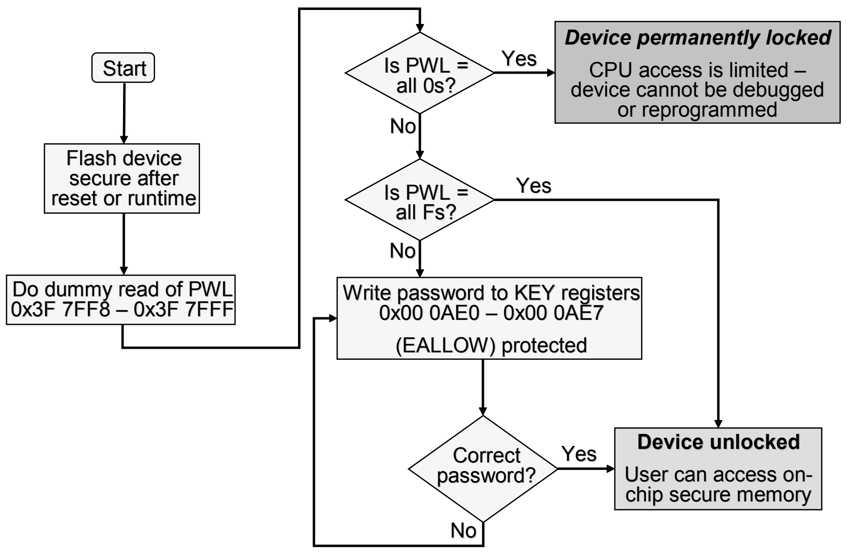


CSM Caveats

- **Never program all the PWL’s as 0x0000**
  - ***Doing so will permanently lock the CSM***
- **Flash addresses 0x3F7F80 to 0x3F7FF5, 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**

10 - 15

CSM Password Match Flow


Start
Flash device secure after reset or runtime
Do dummy read of PWL 0x3F 7FF8 – 0x3F 7FFF
Is PWL = all 0s?
Yes
Device permanently locked
CPU access is limited – device cannot be debugged or reprogrammed
No
Is PWL = all Fs?
Yes
No
Write password to KEY registers 0x00 0AE0 – 0x00 0AE7
(EALLOW) protected
Correct password?
Yes
Device unlocked
User can access on-chip secure memory
No

10 - 16

CSM C-Code Examples

Unlocking the CSM:

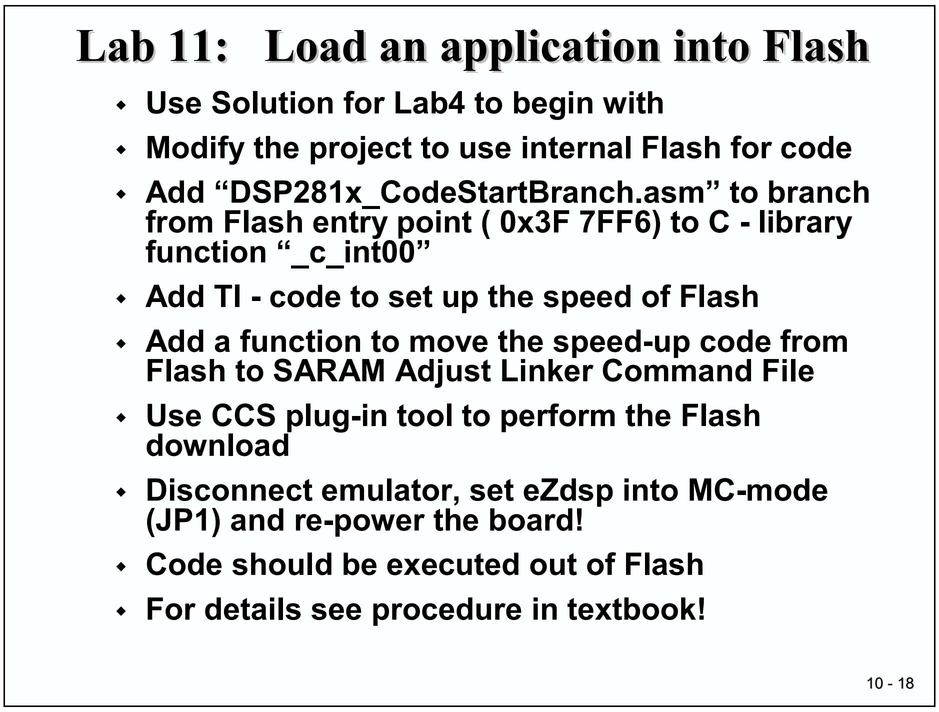
```
volatile int *PWL = &CsmPwl.PSWD0;   //Pointer to PWL register file
volatile int i, tmp;

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”);
```

10 - 17

Lab Exercise 11



Lab 11: Load an application into Flash

- Use Solution for Lab4 to begin with
- Modify the project to use internal Flash for code
- Add "DSP281x_CodeStartBranch.asm" to branch from Flash entry point (0x3F 7FF6) 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, set eZdsp into MC-mode (JP1) and re-power the board!
- Code should be executed out of Flash
- For details see procedure in textbook!

10 - 18

Objective

The objective of this laboratory exercise is to practice with the C28x 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 power on of the C28x. 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 'knight rider' (Lab 4) as the starting point.

What do we have to modify?

In Lab 4 the code was loaded by CCS via the JTAG-Emulator into H0-SARAM after a successful build operation. The linker command file "F2812_EzDSP_RAM_lnk.cmd" took care of the correct connection of the code sections to physical memory addresses of H0SARAM. Obviously, we will have to modify this part. Instead of editing the command file we will use another one ("F2812.cmd"), also provided by TI's header file package.

Furthermore we will have to fill in the Flash entry point address with a connection to the C environment start function ("c_int00"). Out of RESET the Flash memory itself operates with the maximum number of wait states – our code should reduce this wait states to gain the highest possible speed for Flash operations. Unfortunately we can't 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!

A couple of things to take into account in this lab session, as usual let us use a procedure to prepare the project.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab11.pjt** in E:\C281x\Labs.
2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab11.c in E:\C281x\Labs\Lab11.
3. Add the source code file to your project:
   - **Lab11.c**
4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add:
   - **DSP281x_PieCtrl.c**
   - **DSP281x_PieVect.c**
   - **DSP281x_DefaultIsr.c**
   - **DSP281x_CpuTimers.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
   - **F2812_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
   - **F2812.cmd**

   From *C:\ti\c2000\cgtoolslib* add:
   - **rts2800_ml.lib**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

   **C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
   ..\include**

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Add Additional Source Code Files

7. To add the machine code for the Flash entry point at address 0x3F 7FF6 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 "DSP218x_CodeStartBranch.asm".

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add:

   - **DSP281x_CodeStartBranch.asm**

   If you open the file "F2812.cmd" you will see that label "code_start" is linked to "BEGIN" which is defined at address 0x3F 7FF6 in code memory page 0.

8. The function to speed up the internal Flash ("InitFlash()") is also available from TI as part of the source code file "DSP281x_SysCtrl.c".

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add:

   - **DSP281x_SysCtrl.c**

Modify Source Code to Speed up Flash memory

9. Open Lab11.c to edit.

   In main, after the function call "InitSystem()" we have to add code to speed up the Flash memory.

   This will be done by function "InitFlash". But, as already mentioned, this code must run out of SARAM. When we finally run the program out of Flash and the C28x reaches this line all code is still located in Flash. That means, before we can call "InitFlash" the C28x has to copy it into SARAM. Standard ANSI-C provides a memory copy function "memcpy(*dest,*source, number)" for this purpose.

   What do we use for "dest", "source" and "number"?

   Again, the solution can be found in file "DSP281x_SysCtrl.c". Open it and look at the beginning of this file. You will find a "#pragma CODE_SECTION" – line that defines a dedicated code section "ramfuncs" and connects the function "InitFlash()" to it. Symbol "ramfuncs" is used in file "F2812.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 "F2812.cmd" is it to provide the physical addresses to 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 of function "InitFlash()", now available in RAML0:

   InitFlash();

   At begin of Lab11.c declare the symbols used as parameters for memcpy as externals:

   extern unsigned int RamfuncsLoadStart;

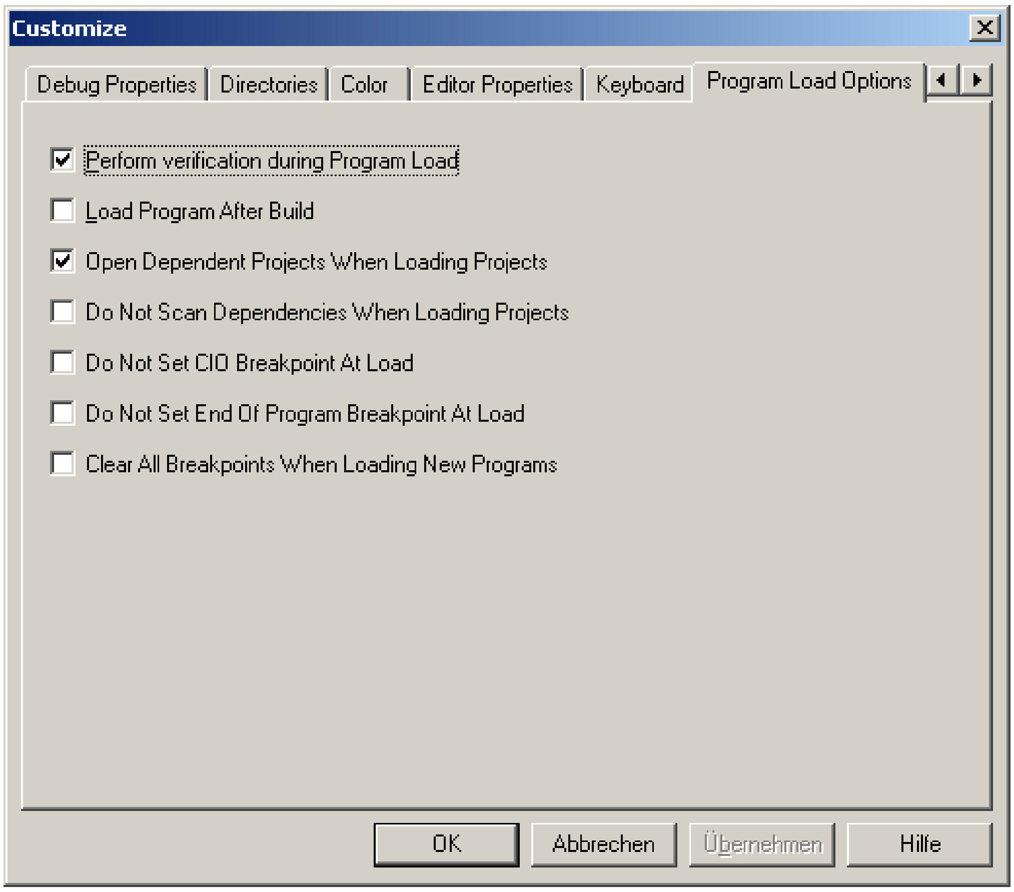
   extern unsigned int RamfuncsLoadEnd;

   extern unsigned int RamfuncsRunStart;

Build project

10. Our code will be compiled to be located in Flash. In our previous lab exercises you probably used the option to download the program after successful build" in CCS ➔ Option ➔ Customize ➔ Load Program After Build. We can't use this feature for this exercise because the destination is Flash.

    Please make sure, that this option is disabled now!



- [✓] Perform verification during Program Load
- [ ] Load Program After Build
- [✓] Open Dependent Projects When Loading Projects
- [ ] Do Not Scan Dependencies When Loading Projects
- [ ] Do Not Set CIO Breakpoint At Load
- [ ] Do Not Set End Of Program Breakpoint At Load
- [ ] Clear All Breakpoints When Loading New Programs

11. Click the "Rebuild All" button or perform:

    Project → Build

    If build was successful you'll get:

    Build Complete,

    0 Errors, 0 Warnings, 0 Remarks.

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 'lab11.map'

13. Open file 'lab11.map' out of subdirectory ..\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 four parts of PAGE 0 are used:

```
RAML0      00008000   00001000  00000016  RWIX
FLASHD     003ec000   00004000  00000016  RWIX
FLASHA     003f6000   00001f80  0000056b  RWIX
BEGIN      003f7ff6   00000002  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 you, you will end up with more or less words in this section.

    Verify also that in PAGE1 the memory RAMH0 is not used.

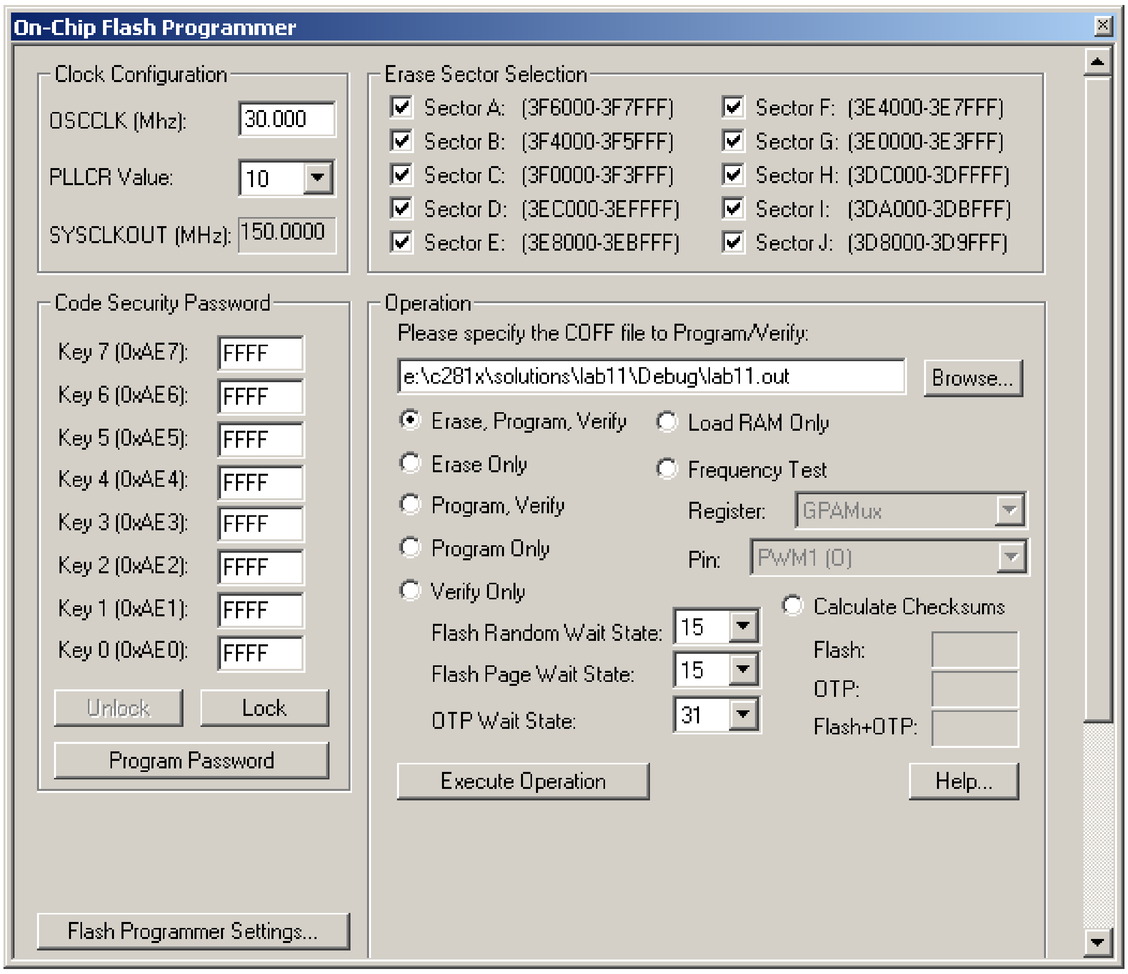
    In the SECTION ALLOCATION MAP you can see how the different portions of our projects code files are distributed into the physical memory sections. For example, the .text-entry shows all the objects that were concatenated into FLASHA.

    Entry 'codestart' connects the object 'CodeStartBranch.obj' to physical address 0x3F 7FF6 and occupies two words.

Use CCS Flash Program Tool

13. 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 Code Composer Studio plug-in tool:

    Tools → F28xx On-Chip Flash Programmer



On-Chip Flash Programmer

Clock Configuration
OSCCLK (Mhz): 30.000
PLLCR Value: 10
SYSCLKOUT (MHz): 150.0000

Erase Sector Selection
✓ Sector A: (3F6000-3F7FFF)
✓ Sector B: (3F4000-3F5FFF)
✓ Sector C: (3F0000-3F3FFF)
✓ Sector D: (3EC000-3EFFFF)
✓ Sector E: (3E8000-3EBFFF)
✓ Sector F: (3E4000-3E7FFF)
✓ Sector G: (3E0000-3E3FFF)
✓ Sector H: (3DC000-3DFFFF)
✓ Sector I: (3DA000-3DBFFF)
✓ Sector J: (3D8000-3D9FFF)

Code Security Password
Key 7 (0xAE7): FFFF
Key 6 (0xAE6): FFFF
Key 5 (0xAE5): FFFF
Key 4 (0xAE4): FFFF
Key 3 (0xAE3): FFFF
Key 2 (0xAE2): FFFF
Key 1 (0xAE1): FFFF
Key 0 (0xAE0): FFFF

Operation
Please specify the COFF file to Program/Verify:
e:\c281x\solutions\lab11\Debug\lab11.out
[✓ Erase, Program, Verify] Erase Only / Program, Verify / Program Only / Verify Only / Load RAM Only / Frequency Test / Calculate Checksums
Register: GPAMux
Pin: PWM1 (0)
Flash Random Wait State: 15
Flash Page Wait State: 15
OTP Wait State: 31
Flash:
OTP:
Flash+OTP:

- Please make sure that OSCCLK is set to 30MHz and PLLCR to 10.
- Do ***NOT*** change the Key 7 to Key 0 entries! They should all show "FFFF"!
- Select the current COFF – out file: ..\Debug\lab11.out
- Select the operation "Erase, Program, Verify
- Hit button "Execute Operation"

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

```
**** Begin Erase/Program/Verify Operation. ***
Erase/Program/Verify Operation in progress...
Erase operation in progress...
Erase operation was successful.
Program operation in progress...
Program operation was successful.
Verify operation in progress...
Verify operation successful.
Erase/Program/Verify Operation succeeded
**** End Erase/Program/Verify Operation. ***
```

Now the code is stored in Flash!

Shut down CCS & Restart eZdsp

14. Close your CCS session.
15. Disconnect the eZdsp from power.
16. Verify that eZdsp Jumper JP1 is in position 2-3 (Microcomputer Mode).
17. Verify that eZdsp Jumper JP7 is in position 1-2 (Boot from Flash)
18. Reconnect eZdsp to power.

Your code should be executed immediately out of Flash, showing the LED-sequence at GPIO-port B.