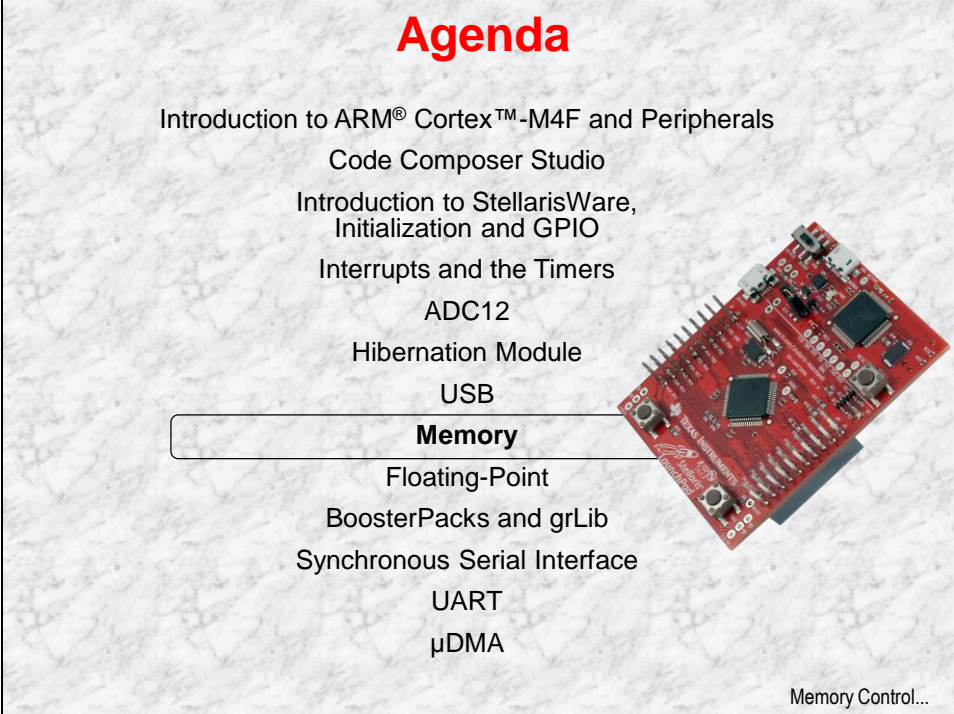


Introduction

In this chapter we will take a look at some memory issues:

- How to write to FLASH in-system.
- How to read/write from EEPROM.
- How to use bit-banding.
- How to configure the Memory Protection Unit (MPU) and deal with faults.



Agenda

Introduction to ARM® Cortex™-M4F and Peripherals
Code Composer Studio
Introduction to StellarisWare,
Initialization and GPIO
Interrupts and the Timers
ADC12
Hibernation Module
USB
Memory
Floating-Point
BoosterPacks and grLib
Synchronous Serial Interface
UART
μDMA

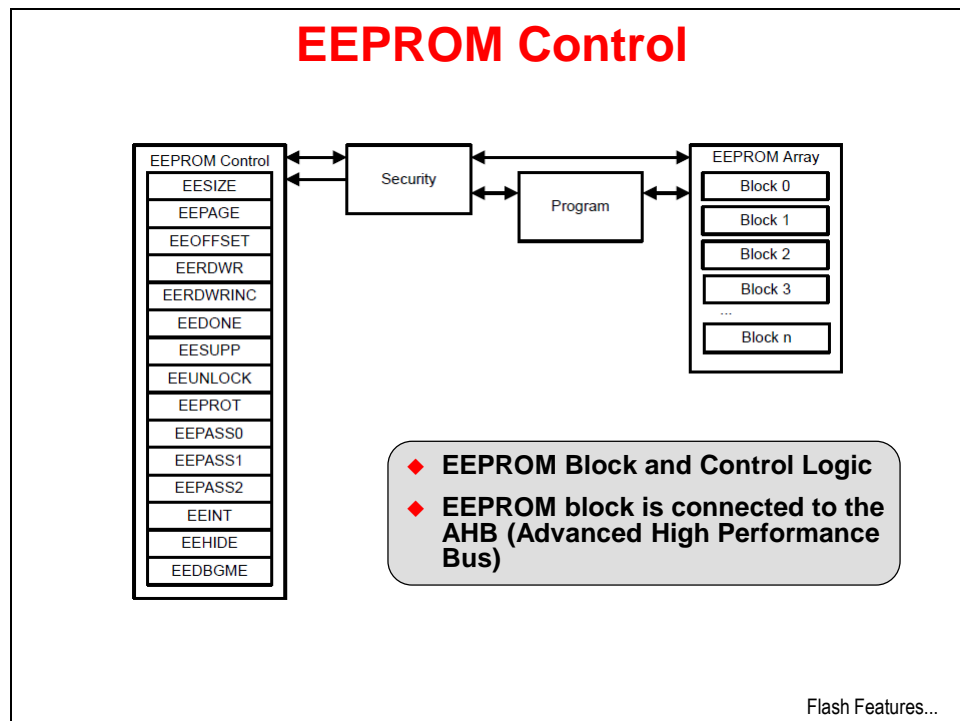
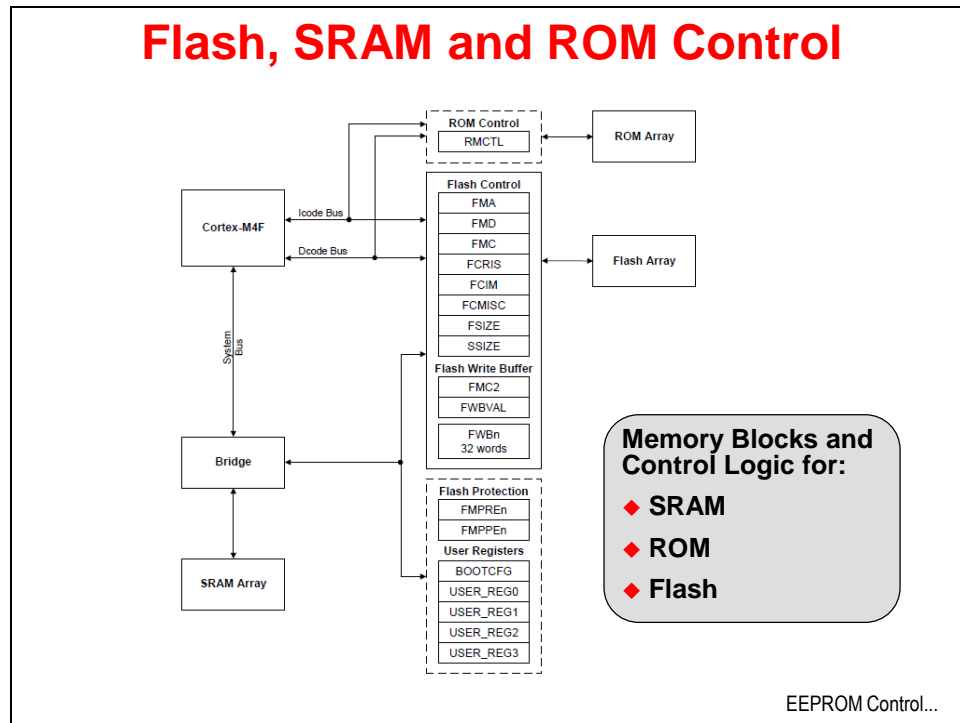
Memory Control...

The slide features a list of topics for the Memory chapter. The word "Memory" is highlighted with a white box and a black border. To the right of the text is a photograph of a red Stellaris LaunchPad board. The background of the slide is a light-colored, textured pattern.

Chapter Topics

Memory	8-1
<i>Chapter Topics.....</i>	<i>8-2</i>
<i>Internal Memory</i>	<i>8-3</i>
<i>Flash</i>	<i>8-4</i>
<i>EEPROM</i>	<i>8-5</i>
<i>SRAM</i>	<i>8-6</i>
<i>Bit-Banding.....</i>	<i>8-7</i>
<i>Memory Protection Unit</i>	<i>8-8</i>
<i>Priority Levels.....</i>	<i>8-9</i>
<i>Lab 8: Memory and the MPU</i>	<i>8-10</i>
Objective.....	8-10
Procedure.....	8-11

Internal Memory



Flash

Flash

- ◆ 256KB / 40MHz starting at 0x00000000
- ◆ Organized in 1KB independently erasable blocks
- ◆ Code fetches and data access occur over separate buses
- ◆ Below 40MHz, Flash access is single cycle
- ◆ Above 40MHz, the prefetch buffer fetches two 32-bit words/cycle. No wait states for sequential code.
- ◆ Branch speculation avoids wait state on some branches
- ◆ Programmable write and execution protection available
- ◆ Simple programming interface



0x00000000 Flash
0x01000000 ROM
0x20000000 SRAM
0x22000000 Bit-banded SRAM
0x40000000 Peripherals & EEPROM
0x42000000 Bit-banded Peripherals
0xE0000000 Instrumentation, ETM, etc.

EEPROM...

EEPROM

EEPROM

- ◆ 2KB of memory starting at 0x400AF000 in Peripheral space
- ◆ Accessible as 512 32-bit words
- ◆ 32 blocks of 16 words (64 bytes) with access protection per block
- ◆ Built-in wear leveling with endurance of 500K writes
- ◆ Lock protection option for the whole peripheral as well as per block using 32-bit to 96-bit codes
- ◆ Interrupt support for write completion to avoid polling
- ◆ Random and sequential read/write access (4 cycles max/word)




0x00000000 Flash
0x01000000 ROM
0x20000000 SRAM
0x22000000 Bit-banded SRAM
0x40000000 Peripherals & EEPROM
0x42000000 Bit-banded Peripherals
0xE0000000 Instrumentation, ETM, etc.

SRAM...

SRAM

SRAM

- ◆ 32KB / 80MHz starting at 0x20000000
- ◆ Bit banded to 0x22000000
- ◆ Can hold code or data



0x00000000 Flash
0x01000000 ROM
0x20000000 SRAM
0x22000000 Bit-banded SRAM
0x40000000 Peripherals & EEPROM
0x42000000 Bit-banded Peripherals
0xE0000000 Instrumentation, ETM, etc.

Bit-Banding...

Bit-Banding

Bit-Banding

- ◆ Reduces the number of read-modify-write operations
- ◆ SRAM and Peripheral space use address aliases to access individual bits in a single, atomic operation
- ◆ SRAM starts at base address 0x20000000
Bit-banded SRAM starts at base address 0x22000000
- ◆ Peripheral space starts at base address 0x40000000
Bit-banded peripheral space starts at base address 0x42000000

The bit-band alias is calculated by using the formula:

```
bit-band alias = bit-band base + (byte offset * 0x20) + (bit number * 4)
```

For example, bit-7 at address 0x20002000 is:

```
0x20002000 + (0x2000 * 0x20) + (7 * 4) = 0x2204001C
```

MPU...

Memory Protection Unit

Memory Protection Unit (MPU)

- ◆ Defines 8 separate memory regions plus a background region accessible only from privileged mode
- ◆ Regions of 256 bytes or more are divided into 8 equal-sized sub-regions
- ◆ MPU definitions for all regions include:
 - Location
 - Size
 - Access permissions
 - Memory attributes
- ◆ Accessing a prohibited region causes a memory management fault



Privilege Levels...

Priority Levels

Cortex M4 Privilege Levels

- ◆ **Privilege levels offer additional protection for software, particularly operating systems**
- ◆ **Unprivileged : software has ...**
 - Limited access to the Priority Mask register
 - No access to the system timer, NVIC, or system control block
 - Possibly restricted access to memory or peripherals (FPU, MPU, etc)
- ◆ **Privileged: software has ...**
 - use of all the instructions and has access to all resources
- ◆ **ISRs operate in privileged mode**
- ◆ **Thread code operates in unprivileged mode unless the level is changed via the Thread Mode Privilege Level (TMPL) bit in the CONTROL register**

Lab...

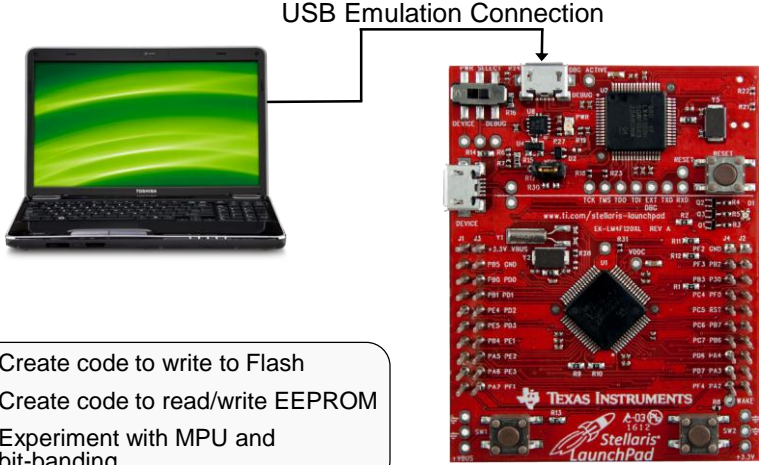
Lab 8: Memory and the MPU

Objective

In this lab you will

- write to FLASH in-system.
- read/write EEPROM.
- Experiment with using the MPU
- Experiment with bit-banding

Lab 8: Memory and the MPU



The diagram illustrates the experimental setup. On the left is a laptop with a green screen. A line connects the laptop to a red Texas Instruments Stellaris LaunchPad on the right. An arrow labeled "USB Emulation Connection" points from the laptop to the LaunchPad's USB port. The LaunchPad board is populated with various components, including a central microcontroller, several integrated circuits, and numerous pins along the edges. The text "TEXAS INSTRUMENTS Stellaris LaunchPad" is visible on the board.

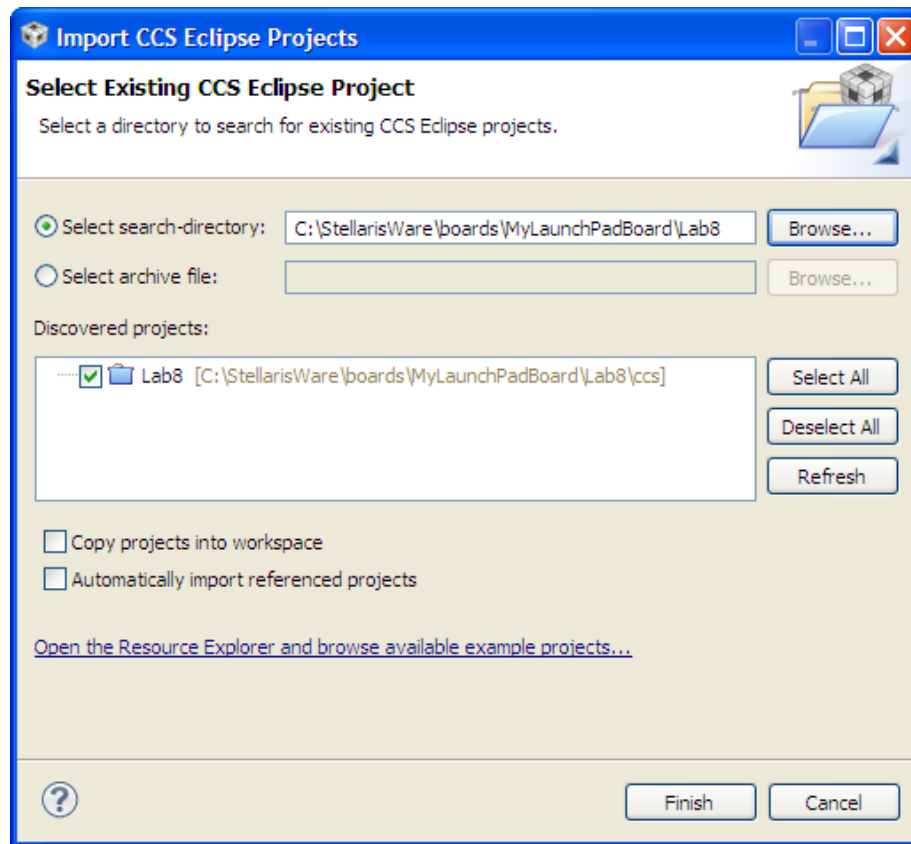
- ◆ Create code to write to Flash
- ◆ Create code to read/write EEPROM
- ◆ Experiment with MPU and bit-banding

Agenda ...

Procedure

Import Lab8

1. We have already created the Lab8 project for you with an empty `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Make sure that the “Copy projects into workspace” checkbox is **unchecked**.



2. Expand the project by clicking the + or ▸ next to Lab8 in the Project Explorer pane. Double-click on `main.c` to open it for editing.

- Let's start out with a straightforward set of starter code. Copy the code below and paste it into your empty `main.c` file.

```
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
    SysCtlDelay(2000000);

    while(1)
    {
    }
}
```

You should already know what this code does, but a quick review won't hurt. The included header files support all the usual stuff including GPIO. Inside `main()`, we set the clock for 40MHz, set the pins connected to the LEDs as outputs and then make sure all three LEDs are off. Next is a two second (approximately) delay followed by a `while(1)` trap.

Save your work.

If you're having problems, this code is in your `Lab8/ccs` folder as `main1.txt`.

Writing to Flash

- We need to find a writable block of flash memory. Right now, that would be flash memory that doesn't currently hold the program we want to execute. Under Project on the menu bar, click Build All. This will build the project without attempting to download it to the LM4F120H5QR memory.
- As we've seen before, CCS creates a map file of the program during the build process. Look in the Debug folder of Lab8 in the Project Explorer pane and double-click on `Lab8.map` to open it.

6. Find the MEMORY CONFIGURATION and SEGMENT ALLOCATION MAP sections as shown below:

```

MEMORY CONFIGURATION

      name          origin      length      used      unused      attr      fill
-----
FLASH          00000000    00040000    0000086a    0003f796    R X
SRAM           20000000    00008000    00000114    00007eec    RW X

SEGMENT ALLOCATION MAP

run origin  load origin  length  init length  attrs  members
-----
00000000   00000000   00000870  00000870    r-x
  00000000   00000000   0000026c  0000026c    r--  .intvecs
  0000026c   0000026c   0000059e  0000059e    r-x  .text
  0000080c   0000080c   00000040  00000040    r--  .const
  00000850   00000850   00000020  00000020    r--  .cinit
20000000   20000000   00000100  00000000    rw-
  20000000   20000000   00000100  00000000    rw-  .stack
20000100   20000100   00000014  00000014    rw-
  20000100   20000100   00000014  00000014    rw-  .data

```

From the map file we can see that the amount of flash memory used is 0x086A in length that starts at 0x0. That means that pretty much anywhere in flash located at an address greater than 0x1000 (for this program) is writable. Let's play it safe and pick the block starting at 0x10000. Remember that flash memory is erasable in 1K blocks. Close Lab8.map.

7. Back in main.c, add the following include to the end of the include statement to add support for flash APIs:

```
#include "driverlib/flash.h"
```

8. At the top of main(), enter the following four lines to add buffers for read and write data and to initialize the write data:

```
unsigned long pulData[2];
unsigned long pulRead[2];
pulData[0] = 0x12345678;
pulData[1] = 0x56789abc;
```

9. Just above the while(1) loop at the end of main(), add these four lines of code:

```
FlashErase(0x10000);  
FlashProgram(pulData, 0x10000, sizeof(pulData));  
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x02);  
SysCtlDelay(20000000);
```

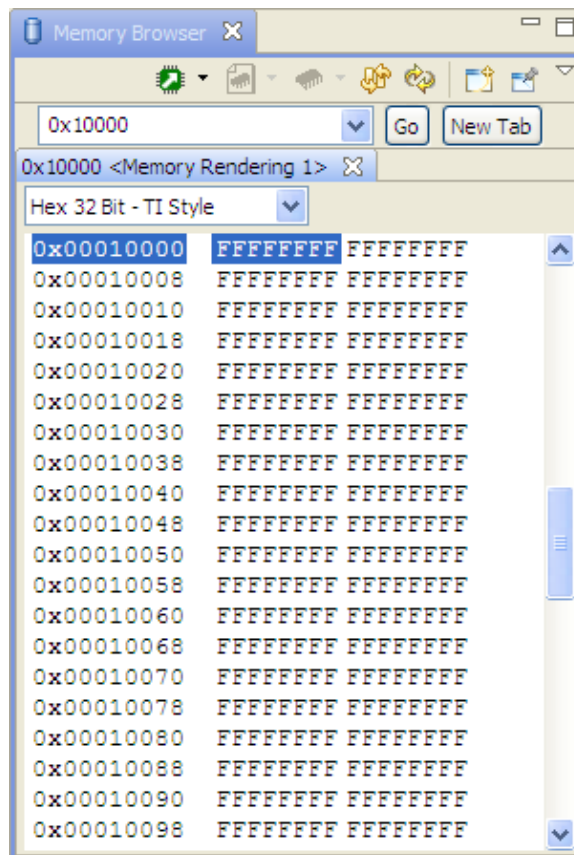
Line:

- 1: Erases the block of flash we identified earlier.
 - 2: Programs the data array we created, to the start of the block, of the length of the array.
 - 3: Lights the red LED to indicate success.
 - 4: Delays about two seconds before the program traps in the while(1) loop.
10. Your code should look like the code below. If you're having issues, this code is located in the Lab8/ccs folder as main2.txt.

```
#include "inc/hw_types.h"  
#include "inc/hw_memmap.h"  
#include "driverlib/sysctl.h"  
#include "driverlib/pin_map.h"  
#include "driverlib/debug.h"  
#include "driverlib/gpio.h"  
#include "driverlib/flash.h"  
  
int main(void)  
{  
    unsigned long pulData[2];  
    unsigned long pulRead[2];  
    pulData[0] = 0x12345678;  
    pulData[1] = 0x56789abc;  
  
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);  
  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);  
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);  
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);  
    SysCtlDelay(20000000);  
  
    FlashErase(0x10000);  
    FlashProgram(pulData, 0x10000, sizeof(pulData));  
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x02);  
    SysCtlDelay(20000000);  
  
    while(1)  
    {  
    }  
}
```

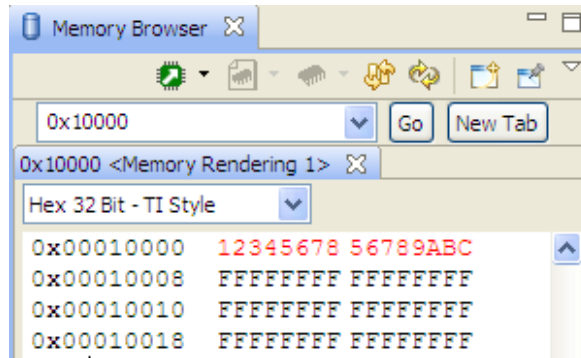
Build, Download and Run the Flash Programming Code

11. Click the Debug button to build and download your program to the LM4F120H5QR memory. Ignore the warning about variable `pulRead` not being referenced. When the process is complete, set a breakpoint on the line containing the `FlashProgram()` API function call.
12. Click the Resume button to run the code. Execution will quickly stop at the breakpoint. On the CCS menu bar, click View → Memory Browser. In the provided entry window, enter `0x10000` as shown below and click Go:



Erased flash should read as all ones. Programming flash only programs zeros. Because of this, writing to un-erased flash memory will produce unpredictable results.

13. Click the Resume button to run the code. The last line of code before the `while(1)` loop will light the red LED. Click the Suspend button. Your Memory Browser will update, displaying your successful write to flash memory.



14. Remove your breakpoint.
15. Make sure you have clicked the Terminate button to stop debugging and return to the CCS Edit perspective. Bear in mind that if you repeat this exercise, the values you just programmed in flash will remain there until that flash block is erased.

Reading and Writing EEPROM

16. Back in `main.c`, add the following line to the end of the include statements to add support for EEPROM APIs:

```
#include "driverlib/eeprom.h"
```

17. Just above the `while(1)` loop, enter the following seven lines of code:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_EEPROM0);
EEPROMInit();
EEPROMMassErase();
EEPROMRead(pulRead, 0x0, sizeof(pulRead));
EEPROMProgram(pulData, 0x0, sizeof(pulData));
EEPROMRead(pulRead, 0x0, sizeof(pulRead));
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x04);
```

Line:

- 1: Turns on the EEPROM peripheral.
- 2: Performs a recovery if power failed during a previous write operation.
- 3: Erases the entire EEPROM. This isn't strictly necessary because, unlike flash, EEPROM does not need to be erased before it is programmed. But this will allow us to see the result of our programming more easily in the lab.
- 4: Reads the erased values into `pulRead` (offset address)
- 5: Programs the data array, to the beginning of EEPROM, of the length of the array.
- 6: Reads that data into array `pulRead`.
- 7: Turns off the red LED and turns on the blue LED.

18. Your code should look like the code below. If you're having issues, this code is located in the Lab8/ccs folder as main3.txt.

```
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/flash.h"
#include "driverlib/eeprom.h"

int main(void)
{
    unsigned long pulData[2];
    unsigned long pulRead[2];
    pulData[0] = 0x12345678;
    pulData[1] = 0x56789abc;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
    SysCtlDelay(20000000);

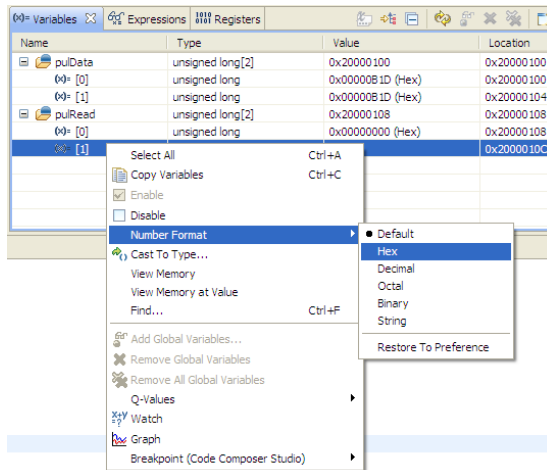
    FlashErase(0x10000);
    FlashProgram(pulData, 0x10000, sizeof(pulData));
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x02);
    SysCtlDelay(20000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_EEPROM0);
    EEPROMInit();
    EEPROMMassErase();
    EEPROMRead(pulRead, 0x0, sizeof(pulRead));
    EEPROMProgram(pulData, 0x0, sizeof(pulData));
    EEPROMRead(pulRead, 0x0, sizeof(pulRead));
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x04);

    while(1)
    {
    }
}
```

Build, Download and Run the Flash Programming Code

19. Click the Debug button to build and download your program to the LM4F120H5QR memory. Code Composer does not currently have a browser for viewing EEPROM memory located in the peripheral area. The code we've written will let us read the values and display them as array values.
20. Click on the Variables tab and expand both of the arrays by clicking the + next to them. Right-click on the first variable's row and select Number Format → Hex. Do this for all four variables.



21. Set a breakpoint on the line containing EEPROMProgram(). We want to verify the previous contents of the EEPROM. Click the Resume button to run to the breakpoint.
22. Since we included the EEPROMMassErase() in the code, the values read from memory should be all Fs as shown below:

Name	Type	Value	Location
pulData	unsigned long[2]	0x200000E8	0x200000E8
[0]	unsigned long	0x12345678 (Hex)	0x200000E8
[1]	unsigned long	0x56789ABC (Hex)	0x200000EC
pulRead	unsigned long[2]	0x200000F0	0x200000F0
[0]	unsigned long	0xFFFFFFFF (Hex)	0x200000F0
[1]	unsigned long	0xFFFFFFFF (Hex)	0x200000F4

23. Click the Resume button to run the code from the breakpoint. When the blue LED on the board lights, click the Suspend button. The values read from memory should now be the same as those in the write array:

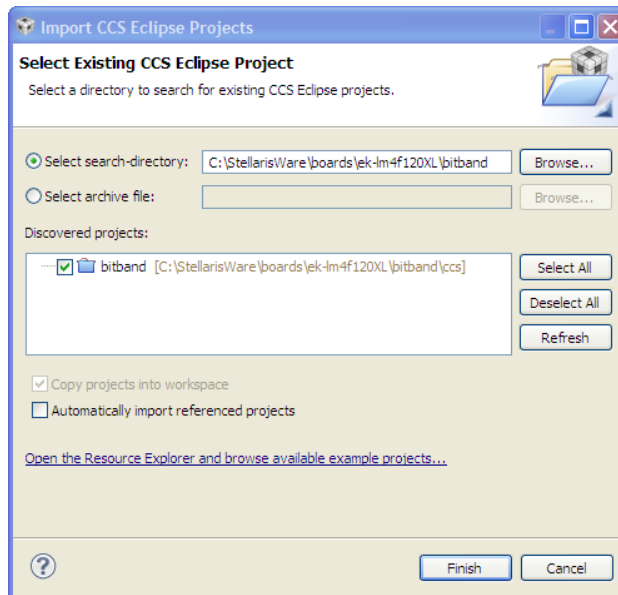
Name	Type	Value	Location
pulData	unsigned long[2]	0x200000E8	0x200000E8
[0]	unsigned long	0x12345678 (Hex)	0x200000E8
[1]	unsigned long	0x56789ABC (Hex)	0x200000EC
pulRead	unsigned long[2]	0x200000F0	0x200000F0
[0]	unsigned long	0x12345678 (Hex)	0x200000F0
[1]	unsigned long	0x56789ABC (Hex)	0x200000F4

Further EEPROM Information

24. EEPROM is unlocked at power-up. Your locking scheme, if you choose to use one, can be simple or complex. You can lock the entire EEPROM or individual blocks. You can enable reading without a password and writing with one if you desire. You can also hide blocks of EEPROM, making them invisible to further accesses.
25. EEPROM reads and writes are multi-cycle instructions. The ones used in the lab code are “blocking calls”, meaning that program execution will stall until the operation is complete. There are also “non-blocking” calls that do not stall program execution. When using those calls you should either poll the EEPROM or enable an interrupt scheme to assure the operation completes properly.
26. Remove your breakpoint, click Terminate to return to the CCS Edit perspective and close the Lab8 project.

Bit-Banding

27. The LaunchPad board Stellaris examples include a bit-banding project. Click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish.



28. Expand the project in the Project Explorer pane and double-click on `bitband.c` to open it for viewing. Page down until you reach `main()`. You should recognize most of the setup code, but note that the UART is also set up. We'll be able to watch the code run via `UARTprintf()` statements that will send data to a terminal program running on your laptop. Also note that this example uses ROM API function calls.

29. Continue paging down until you find the `for (ulIdx=0;ulIdx<32;ulIdx++)` loop. This 32-step loop will write `0xdecafbad` into memory bit by bit using bit-banding. This will be done using the `HWREGBITW()` macro.

Right-click on `HWREGBITW()` and select **Open Declaration**.

The `HWREGBITW(x,b)` macro is an alias from:

```
HWREG(((unsigned long)(x) & 0xF0000000) | 0x02000000 |
      (((unsigned long)(x) & 0x000FFFFF) << 5) | ((b) << 2))
```

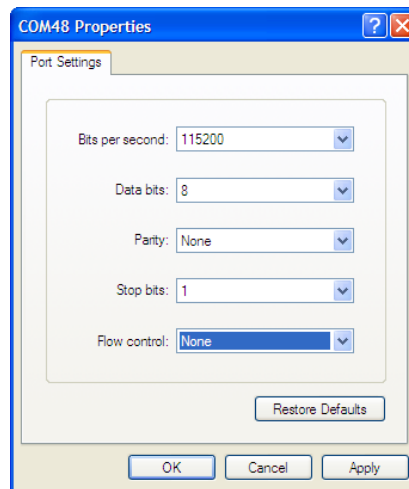
which is C code for:

```
bit-band alias = bit-band base + (byte offset * 0x20) + (bit number * 4)
```

This is the calculation for the bit-banded address of bit `b` of location `x`. `HWREG` is a macro that programs a hardware register (or memory location) with a value.

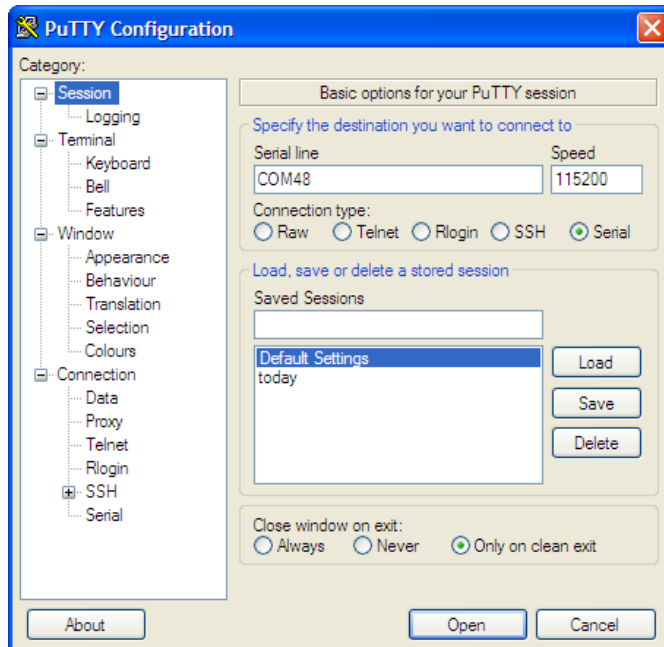
The loop in `bitband.c` reads the bits from `0xdecafbad` and programs them into the calculated bit-band addresses of `g_ulValue`. Throughout the loop the program transfers the value in `g_ulValue` to the UART for viewing on the host. Once all bits have been written to `g_ulValue`, the variable is read directly (all 32 bits) to make sure the value is `0xdecafbad`. There is another loop that reads the bits individually to make sure that they can be read back using bit-banding

30. Click the **Debug** button to build and download the program to the LM4F120H5QR.
31. If you are using **Windows 7**, skip to step 32. **In WinXP**, open HyperTerminal by clicking **Start** → **Run...**, then type `hypertrm` in the **Open:** box and click **OK**. Pick any name you like for your connection and click **OK**. In the next dialog box, change the **Connect using:** selection to **COM##**, where **##** is the COM port number you noted in Lab1. Click **OK**. Make the selections shown below and click **OK**.



Skip to step 33.

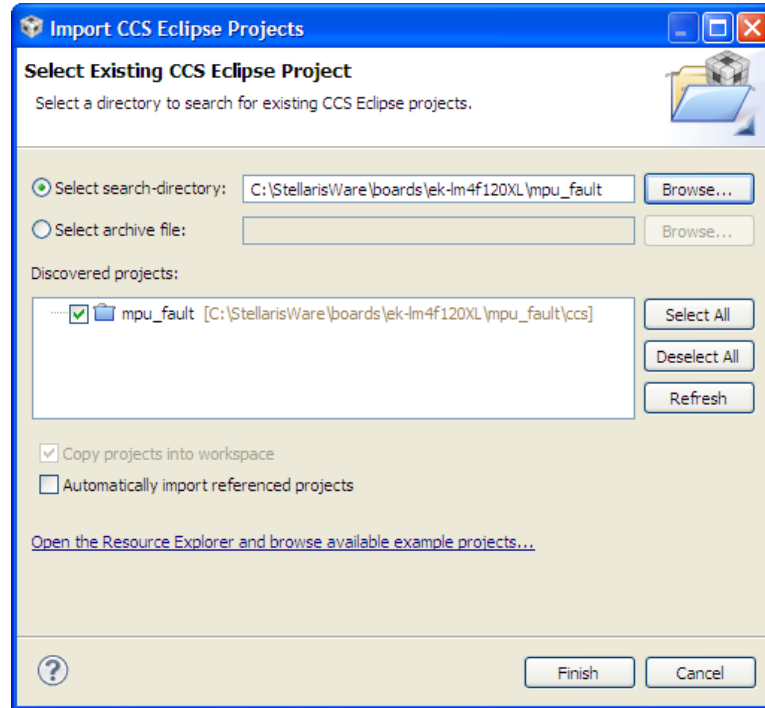
32. In **Win7**, double-click on `putty.exe`. Make the settings shown below and then click **Open**. Your COM port number will be the one you noted in Lab1.



33. Click the **Resume** button in CCS and watch the bits drop into place in your terminal window. The `Delay()` in the loop causes this to take about 30 seconds.
34. Close your terminal window. Click **Terminate** in CCS to return to the CCS Edit perspective and close the `bitband` project.

Memory Protection Unit (MPU)

35. The LaunchPad board Stellaris examples include an `mpu_fault` project. Click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Note that this project is automatically copied into your workspace.



36. Expand the project and double-click on `mpu_fault.c` for viewing.

Again, things should look pretty normal in the setup, so let's look at where things are different.

Find the function called `MPUFaultHandler`. This exception handler looks just like an ISR. The main purpose of this code is to preserve the address of the problem that caused the fault, as well as the status register.

Open `startup_ccs.c` and find where `MPUFaultHandler` has been placed in the vector table. Close `startup_ccs.c`.

37. In `mpu_fault.c`, find `main()`. Using the memory map shown, the `MPURegionSet()` calls will configure 6 different regions and parameters for the MPU. The code following the final `MPURegionSet()` call triggers (or doesn't trigger) the fault conditions. Status messages are sent to the UART for display on the host.

`MPURegionSet()` uses the following parameters:

- Region number to set up
- Address of the region (as aligned by the flags)
- Flags

Flags are a set of parameters (OR'd together) that determine the attributes of the region (size | execute permission | read/write permission | sub-region disable | enable/disable)

The size flag determines the size of a region and must be one of the following:

MPU_RGN_SIZE_32B	MPU_RGN_SIZE_512K
MPU_RGN_SIZE_64B	MPU_RGN_SIZE_1M
MPU_RGN_SIZE_128B	MPU_RGN_SIZE_2M
MPU_RGN_SIZE_256B	MPU_RGN_SIZE_4M
MPU_RGN_SIZE_512B	MPU_RGN_SIZE_8M
MPU_RGN_SIZE_1K	MPU_RGN_SIZE_16M
MPU_RGN_SIZE_2K	MPU_RGN_SIZE_32M
MPU_RGN_SIZE_4K	MPU_RGN_SIZE_64M
MPU_RGN_SIZE_8K	MPU_RGN_SIZE_128M
MPU_RGN_SIZE_16K	MPU_RGN_SIZE_256M
MPU_RGN_SIZE_32K	MPU_RGN_SIZE_512M
MPU_RGN_SIZE_64K	MPU_RGN_SIZE_1G
MPU_RGN_SIZE_128K	MPU_RGN_SIZE_2G
MPU_RGN_SIZE_256K	MPU_RGN_SIZE_4G

The execute permission flag must be one of the following:

MPU_RGN_PERM_EXEC enables the region for execution of code

MPU_RGN_PERM_NOEXEC disables the region for execution of code

The read/write access permissions are applied separately for the privileged and user modes. The read/write access flags must be one of the following:

MPU_RGN_PERM_PRV_NO_USR_NO - no access in privileged or user mode
MPU_RGN_PERM_PRV_RW_USR_NO - privileged read/write, no user access
MPU_RGN_PERM_PRV_RW_USR_RO - privileged read/write, user read-only
MPU_RGN_PERM_PRV_RW_USR_RW - privileged read/write, user read/write
MPU_RGN_PERM_PRV_RO_USR_NO - privileged read-only, no user access
MPU_RGN_PERM_PRV_RO_USR_RO - privileged read-only, user read-only

Each region is automatically divided into 8 equally-sized sub-regions by the MPU. Sub-regions can only be used in regions of size 256 bytes or larger. Any of these 8 sub-regions can be disabled, allowing for creation of “holes” in a region which can be left open, or overlaid by another region with different attributes. Any of the 8 sub-regions can be disabled with a logical OR of any of the following flags:

MPU_SUB_RGN_DISABLE_0
MPU_SUB_RGN_DISABLE_1
MPU_SUB_RGN_DISABLE_2
MPU_SUB_RGN_DISABLE_3
MPU_SUB_RGN_DISABLE_4
MPU_SUB_RGN_DISABLE_5
MPU_SUB_RGN_DISABLE_6
MPU_SUB_RGN_DISABLE_7

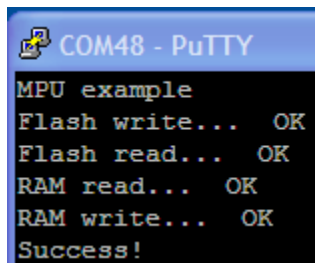
Finally, the region can be initially enabled or disabled with one of the following flags:

MPU_RGN_ENABLE
MPU_RGN_DISABLE

38. Start your terminal program as shown earlier. Click the Debug button to build and download the program to the LM4F120H5QR. Click the Resume button to run the program.

39. The tests are as follows:

- Attempt to write to the flash. This should cause a protection fault due to the fact that this region is read-only. If this fault occurs, the terminal program will show OK.
- Attempt to read from the disabled section of flash. If this fault occurs, the terminal program will show OK.
- Attempt to read from the read-only area of RAM. If a fault does not occur, the terminal program will show OK.
- Attempt to write to the read-only area of RAM. If this fault occurs, the terminal program will show OK.



```
COM48 - PuTTY
MPU example
Flash write... OK
Flash read... OK
RAM read... OK
RAM write... OK
Success!
```

40. When you are done, close your terminal program. Click the Terminate button in CCS to return to the CCS Edit perspective. Close the `mpu_fault` project and minimize Code Composer Studio.



You're done.