Lab 8

EEPROM Programming

```c
#include <stdint.h>
#include <stdbool.h>
#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)
{
	uint32_t pui32Data[2], pui32Read[2];
	pui32Data[0] = 0x12345678;
	pui32Data[1] = 0x56789abc;

	SysCtlClockSet
(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);	//sets the
clock to 40 MHz

	SysCtlPeripheralEnable (SYSCTL_PERIPH_GPIOF); //enables the gpio port f
	GPIOPinTypeGPIOOutput (GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
	//sets GPIO.PF.1-3 as outputs
	GPIOPinWrite (GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
	//writes 0 to PF 1-3
	SysCtlDelay (20000000);	//delay of 20 seconds

	FlashErase (0x10000);	//erases the block of data at 0x10000
	FlashProgram (pui32Data, 0x10000, sizeof (pui32Data));	//programs the data
to the start of the block
	GPIOPinWrite (GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x02);
	//writes 0x02 to port F
	SysCtlDelay (20000000);

	SysCtlPeripheralEnable (SYSCTL_PERIPH_EEPROM0);	//enables the EEPROM
	EEPROMInit ();	//performs recovery if power failed
	EEPROMMassErase (); //erases the EEPROM
	EEPROMRead (pui32Read, 0x0, sizeof (pui32Read));	//read the EEPROM memory
at location 0x0 and store in puid32Read
	EEPROMProgram (pui32Data, 0x0, sizeof (pui32Data)); //write pui32Data contents
to 0x0
	EEPROMRead (pui32Read, 0x0, sizeof (pui32Read));	//read the EEPROM memory
at location 0x0 and store in puid32Read
	GPIOPinWrite (GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x04);
	//writes 0x04 to port f
	while (1)
	{}
}
```

Bit Banding

```
//*****************************************************************************
//
// bitband.c - Bit-band manipulation example.
//
// Copyright (c) 2012-2016 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 2.1.3.156 of the EK-TM4C123GXL Firmware Package.
//
//*****************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/fpu.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/rom.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

//*****************************************************************************
//
//! \addtogroup example_list
//! <h1>Bit-Banding (bitband)</h1>
//!
//! This example application demonstrates the use of the bit-banding
//! capabilities of the Cortex-M4F microprocessor.  All of SRAM and all of the
//! peripherals reside within bit-band regions, meaning that bit-banding
//! operations can be applied to any of them.  In this example, a variable in
//! SRAM is set to a particular value one bit at a time using bit-banding
//! operations (it would be more efficient to do a single non-bit-banded write;
//! this simply demonstrates the operation of bit-banding).
//
//*****************************************************************************
```

```c
//*****************************************************************************
//
// The value that is to be modified via bit-banding.
//
//*****************************************************************************
static volatile uint32_t g_ui32Value;

//*****************************************************************************
//
// The error routine that is called if the driver library encounters an error.
//
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
    while(1)
    {
        //
        // Hang on runtime error.
        //
    }
}
#endif

//*****************************************************************************
//
// Delay for the specified number of seconds.  Depending upon the current
// SysTick value, the delay will be between N-1 and N seconds (i.e. N-1 full
// seconds are guaranteed, along with the remainder of the current second).
//
//*****************************************************************************
void
Delay(uint32_t ui32Seconds)
{
    //
    // Loop while there are more seconds to wait.
    //
    while(ui32Seconds--)
    {
        //
        // Wait until the SysTick value is less than 1000.
        //
        while(ROM_SysTickValueGet() > 1000)
        {
        }

        //
        // Wait until the SysTick value is greater than 1000.
        //
        while(ROM_SysTickValueGet() < 1000)
        {
        }
    }
```

```c
}

//*****************************************************************************
//
// Configure the UART and its pins.  This must be called before UARTprintf().
//
//*****************************************************************************
void
ConfigureUART(void)
{
    //
    // Enable the GPIO Peripheral used by the UART.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    // Enable UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure GPIO Pins for UART mode.
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Use the internal 16MHz oscillator as the UART clock source.
    //
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioConfig(0, 115200, 16000000);
}

//*****************************************************************************
//
// This example demonstrates the use of bit-banding to set individual bits
// within a word of SRAM.
//
//*****************************************************************************
int
main(void)
{
    uint32_t ui32Errors, ui32Idx;

    //
    // Enable lazy stacking for interrupt handlers.  This allows floating-point
    // instructions to be used within interrupt handlers, but at the expense of
    // extra stack usage.
    //
    ROM_FPULazyStackingEnable();
```

```c
//
// Set the clocking to run directly from the crystal.
//
ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);

//
// Initialize the UART interface.
//
ConfigureUART();

UARTprintf("\033[2JBit banding...\n");

//
// Set up and enable the SysTick timer.  It will be used as a reference
// for delay loops.  The SysTick timer period will be set up for one
// second.
//
ROM_SysTickPeriodSet(ROM_SysCtlClockGet());
ROM_SysTickEnable();

//
// Set the value and error count to zero.
//
g_ui32Value = 0;
ui32Errors = 0;

//
// Print the initial value to the UART.
//
UARTprintf("\r%08x", g_ui32Value);

//
// Delay for 1 second.
//
Delay(1);

//
// Set the value to 0xdecafbad using bit band accesses to each individual
// bit.
//
for(ui32Idx = 0; ui32Idx < 32; ui32Idx++)
{
    //
    // Set this bit.
    //
    HWREGBITW(&g_ui32Value, 31 - ui32Idx) = (0xdecafbad >>
                                             (31 - ui32Idx)) & 1;

    //
    // Print the current value to the UART.
    //
    UARTprintf("\r%08x", g_ui32Value);
```

```
        //
        // Delay for 1 second.
        //
        Delay(1);
    }

    //
    // Make sure that the value is 0xdecafbad.
    //
    if(g_ui32Value != 0xdecafbad)
    {
        ui32Errors++;
    }

    //
    // Make sure that the individual bits read back correctly.
    //
    for(ui32Idx = 0; ui32Idx < 32; ui32Idx++)
    {
        if(HWREGBITW(&g_ui32Value, ui32Idx) != ((0xdecafbad >> ui32Idx) & 1))
        {
            ui32Errors++;
        }
    }

    //
    // Print out the result.
    //
    if(ui32Errors)
    {
        UARTprintf("\nErrors!\n");
    }
    else
    {
        UARTprintf("\nSuccess!\n");
    }

    //
    // Loop forever.
    //
    while(1)
    {
    }
}
```

MPU

```c
//*****************************************************************************
//
// mpu_fault.c - MPU example.
//
// Copyright (c) 2012-2016 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 2.1.3.156 of the EK-TM4C123GXL Firmware Package.
//
//*****************************************************************************

#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_nvic.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/mpu.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

//*****************************************************************************
//
//! \addtogroup example_list
//! <h1>MPU (mpu_fault)</h1>
//!
//! This example application demonstrates the use of the MPU to protect a
//! region of memory from access, and to generate a memory management fault
//! when there is an access violation.
//!
//! UART0, connected to the virtual serial port and running at 115,200, 8-N-1,
//! is used to display messages from this application.
//
```

```c
//*****************************************************************************

//*****************************************************************************
//
// Variables to hold the state of the fault status when the fault occurs and
// the faulting address.
//
//*****************************************************************************
static volatile uint32_t g_ui32MMAR;
static volatile uint32_t g_ui32FaultStatus;

//*****************************************************************************
//
// A counter to track the number of times the fault handler has been entered.
//
//*****************************************************************************
static volatile uint32_t g_ui32MPUFaultCount;

//*****************************************************************************
//
// A location for storing data read from various addresses.  Volatile forces
// the compiler to use it and not optimize the access away.
//
//*****************************************************************************
static volatile uint32_t g_ui32Value;

//*****************************************************************************
//
// The error routine that is called if the driver library encounters an error.
//
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
}
#endif

//*****************************************************************************
//
// The exception handler for memory management faults, which are caused by MPU
// access violations.  This handler will verify the cause of the fault and
// clear the NVIC fault status register.
//
//*****************************************************************************
void
MPUFaultHandler(void)
{
    //
    // Preserve the value of the MMAR (the address causing the fault).
    // Preserve the fault status register value, then clear it.
    //
    g_ui32MMAR = HWREG(NVIC_MM_ADDR);
    g_ui32FaultStatus = HWREG(NVIC_FAULT_STAT);
```

```c
    HWREG(NVIC_FAULT_STAT) = g_ui32FaultStatus;

    //
    // Increment a counter to indicate the fault occurred.
    //
    g_ui32MPUFaultCount++;

    //
    // Disable the MPU so that this handler can return and cause no more
    // faults.  The actual instruction that faulted will be re-executed.
    //
    ROM_MPUDisable();
}

//*****************************************************************************
//
// Configure the UART and its pins.  This must be called before UARTprintf().
//
//*****************************************************************************
void
ConfigureUART(void)
{
    //
    // Enable the GPIO Peripheral used by the UART.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    // Enable UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure GPIO Pins for UART mode.
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Use the internal 16MHz oscillator as the UART clock source.
    //
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioConfig(0, 115200, 16000000);
}

//*****************************************************************************
//
// This example demonstrates how to configure MPU regions for different levels
// of memory protection.  The following memory map is set up:
//
```

```
// 0000.0000 - 0000.1C00 - rgn 0: executable read-only, flash
// 0000.1C00 - 0000.2000 - rgn 0: no access, flash (disabled sub-region 7)
// 2000.0000 - 2000.4000 - rgn 1: read-write, RAM
// 2000.4000 - 2000.6000 - rgn 2: read-only, RAM (disabled sub-rgn 4 of rgn 1)
// 2000.6000 - 2000.7FFF - rgn 1: read-write, RAM
// 4000.0000 - 4001.0000 - rgn 3: read-write, peripherals
// 4001.0000 - 4002.0000 - rgn 3: no access (disabled sub-region 1)
// 4002.0000 - 4006.0000 - rgn 3: read-write, peripherals
// 4006.0000 - 4008.0000 - rgn 3: no access (disabled sub-region 6, 7)
// E000.E000 - E000.F000 - rgn 4: read-write, NVIC
// 0100.0000 - 0100.FFFF - rgn 5: executable read-only, ROM
//
// The example code will attempt to perform the following operations and check
// the faulting behavior:
//
// - write to flash                         (should fault)
// - read from the disabled area of flash   (should fault)
// - read from the read-only area of RAM    (should not fault)
// - write to the read-only section of RAM  (should fault)
//
//*****************************************************************************
int
main(void)
{
    unsigned int bFail = 0;

    //
    // Enable lazy stacking for interrupt handlers.  This allows floating-point
    // instructions to be used within interrupt handlers, but at the expense of
    // extra stack usage.
    //
    ROM_FPULazyStackingEnable();

    //
    // Set the clocking to run directly from the crystal.
    //
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);

    //
    // Initialize the UART and write status.
    //
    ConfigureUART();

    UARTprintf("\033[2JMPU example\n");

    //
    // Configure an executable, read-only MPU region for flash.  It is a 16 KB
    // region with the last 2 KB disabled to result in a 14 KB executable
    // region.  This region is needed so that the program can execute from
    // flash.
    //
    ROM_MPURegionSet(0, FLASH_BASE,
                     MPU_RGN_SIZE_16K | MPU_RGN_PERM_EXEC |
                     MPU_RGN_PERM_PRV_RO_USR_RO | MPU_SUB_RGN_DISABLE_7 |
```

```c
                 MPU_RGN_ENABLE);

//
// Configure a read-write MPU region for RAM.  It is a 32 KB region.  There
// is a 4 KB sub-region in the middle that is disabled in order to open up
// a hole in which different permissions can be applied.
//
ROM_MPURegionSet(1, SRAM_BASE,
                 MPU_RGN_SIZE_32K | MPU_RGN_PERM_NOEXEC |
                 MPU_RGN_PERM_PRV_RW_USR_RW | MPU_SUB_RGN_DISABLE_4 |
                 MPU_RGN_ENABLE);

//
// Configure a read-only MPU region for the 4 KB of RAM that is disabled in
// the previous region.  This region is used for demonstrating read-only
// permissions.
//
ROM_MPURegionSet(2, SRAM_BASE + 0x4000,
                 MPU_RGN_SIZE_2K | MPU_RGN_PERM_NOEXEC |
                 MPU_RGN_PERM_PRV_RO_USR_RO | MPU_RGN_ENABLE);

//
// Configure a read-write MPU region for peripherals.  The region is 512 KB
// total size, with several sub-regions disabled to prevent access to areas
// where there are no peripherals.  This region is needed because the
// program needs access to some peripherals.
//
ROM_MPURegionSet(3, 0x40000000,
                 MPU_RGN_SIZE_512K | MPU_RGN_PERM_NOEXEC |
                 MPU_RGN_PERM_PRV_RW_USR_RW | MPU_SUB_RGN_DISABLE_1 |
                 MPU_SUB_RGN_DISABLE_6 | MPU_SUB_RGN_DISABLE_7 |
                 MPU_RGN_ENABLE);

//
// Configure a read-write MPU region for access to the NVIC.  The region is
// 4 KB in size.  This region is needed because NVIC registers are needed
// in order to control the MPU.
//
ROM_MPURegionSet(4, NVIC_BASE,
                 MPU_RGN_SIZE_4K | MPU_RGN_PERM_NOEXEC |
                 MPU_RGN_PERM_PRV_RW_USR_RW | MPU_RGN_ENABLE);

//
// Configure an executable, read-only MPU region for ROM.  It is a 64 KB
// region.  This region is needed so that ROM library calls work.
//
ROM_MPURegionSet(5, (uint32_t)ROM_APITABLE & 0xFFFF0000,
                 MPU_RGN_SIZE_64K | MPU_RGN_PERM_EXEC |
                 MPU_RGN_PERM_PRV_RO_USR_RO | MPU_RGN_ENABLE);

//
// Need to clear the NVIC fault status register to make sure there is no
// status hanging around from a previous program.
//
g_ui32FaultStatus = HWREG(NVIC_FAULT_STAT);
```

```c
HWREG(NVIC_FAULT_STAT) = g_ui32FaultStatus;

//
// Enable the MPU fault.
//
ROM_IntEnable(FAULT_MPU);

//
// Enable the MPU.  This will begin to enforce the memory protection
// regions.  The MPU is configured so that when in the hard fault or NMI
// exceptions, a default map will be used.  Neither of these should occur
// in this example program.
//
ROM_MPUEnable(MPU_CONFIG_HARDFLT_NMI);

//
// Attempt to write to the flash.  This should cause a protection fault due
// to the fact that this region is read-only.
//
UARTprintf("Flash write... ");
g_ui32MPUFaultCount = 0;
HWREG(0x100) = 0x12345678;

//
// Verify that the fault occurred, at the expected address.
//
if((g_ui32MPUFaultCount == 1) && (g_ui32FaultStatus == 0x82) &&
   (g_ui32MMAR == 0x100))
{
    UARTprintf(" OK\n");
}
else
{
    bFail = 1;
    UARTprintf("NOK\n");
}

//
// The MPU was disabled when the previous fault occurred, so it needs to be
// re-enabled.
//
ROM_MPUEnable(MPU_CONFIG_HARDFLT_NMI);

//
// Attempt to read from the disabled section of flash, the upper 2 KB of
// the 16 KB region.
//
UARTprintf("Flash read... ");
g_ui32MPUFaultCount = 0;
g_ui32Value = HWREG(0x3820);

//
// Verify that the fault occurred, at the expected address.
//
if((g_ui32MPUFaultCount == 1) && (g_ui32FaultStatus == 0x82) &&
```

```
    (g_ui32MMAR == 0x3820))
{
    UARTprintf(" OK\n");
}
else
{
    bFail = 1;
    UARTprintf("NOK\n");
}

//
// The MPU was disabled when the previous fault occurred, so it needs to be
// re-enabled.
//
ROM_MPUEnable(MPU_CONFIG_HARDFLT_NMI);

//
// Attempt to read from the read-only area of RAM, the middle 4 KB of the
// 32 KB region.
//
UARTprintf("RAM read... ");
g_ui32MPUFaultCount = 0;
g_ui32Value = HWREG(0x20004440);

//
// Verify that the RAM read did not cause a fault.
//
if(g_ui32MPUFaultCount == 0)
{
    UARTprintf(" OK\n");
}
else
{
    bFail = 1;
    UARTprintf("NOK\n");
}

//
// The MPU should not have been disabled since the last access was not
// supposed to cause a fault.  But if it did cause a fault, then the MPU
// will be disabled, so re-enable it here anyway, just in case.
//
ROM_MPUEnable(MPU_CONFIG_HARDFLT_NMI);

//
// Attempt to write to the read-only area of RAM, the middle 4 KB of the
// 32 KB region.
//
UARTprintf("RAM write... ");
g_ui32MPUFaultCount = 0;
HWREG(0x20004460) = 0xabcdef00;

//
// Verify that the RAM write caused a fault.
//
```

```c
    if((g_ui32MPUFaultCount == 1) && (g_ui32FaultStatus == 0x82) &&
       (g_ui32MMAR == 0x20004460))
    {
        UARTprintf(" OK\n");
    }
    else
    {
        bFail = 1;
        UARTprintf("NOK\n");
    }

    //
    // Display the results of the example program.
    //
    if(bFail)
    {
        UARTprintf("Failure!\n");
    }
    else
    {
        UARTprintf("Success!\n");
    }

    //
    // Disable the MPU, so there are no lingering side effects if another
    // program is run.
    //
    ROM_MPUDisable();

    //
    // Loop forever.
    //
    while(1)
    {
    }
}
```