**EEPROM & MPU Lab**

**Video with the following name found in Dropbox:**

VIDEO.mp4

**Part 1: EEPROM Example**

**#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];

uint32\_t pui32Read[2];

pui32Data[0] = 0x12345678;

pui32Data[1] = 0x56789abc;

SysCtlClockSet(SYSCTL\_SYSDIV\_5|SYSCTL\_USE\_PLL|SYSCTL\_XTAL\_16MHZ|SYSCTL\_OSC\_MAIN); //set clock to 40 MHz

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOF); //enable port F

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3); //set PF1, PF2, P3 as outputs

GPIOPinWrite(GPIO\_PORTF\_BASE,GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3, 0x00); //clear PF1, PF2, PF3

SysCtlDelay(20000000); //delay about 2 seconds

FlashErase(0x10000); //erase block of flash starting at 0x10000

FlashProgram(pui32Data, 0x10000, **sizeof**(pui32Data)); //program the array pui32Data to beginning of block

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_1 | GPIO\_PIN\_2 | GPIO\_PIN\_3, 0x02); //light red LED to indicate success

SysCtlDelay(20000000); //delay about 2 seconds

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_EEPROM0); //turn on EEPROM peripheral

EEPROMInit(); //perform recovery if power failed during previous operation

EEPROMMassErase(); //erase entire PROM

EEPROMRead(pui32Read, 0x0, **sizeof**(pui32Read)); //read erased values into pulRead (offset address)

EEPROMProgram(pui32Data, 0x0, **sizeof**(pui32Data)); //program the data array to the beginning of the EEPROM

EEPROMRead(pui32Read, 0x0, **sizeof**(pui32Read)); //read data into array puRead

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_1 | GPIO\_PIN\_2 | GPIO\_PIN\_3, 0x04); //turn off red LED and turn on blue LED

**while**(1)

{

}

}

**Part 2: MPU Example**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

// mpu\_fault.c - MPU example.

//

// Copyright (c) 2012-2015 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 2.1.1.71 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.

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

// 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);

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 |

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);

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.

//

**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)

{

}

}