LPTMR0 Interrupt

Course: ENGG*3640 Microcomputer Interfacing

Instructor: Radu Muresan

Student Names/Numbers:

Bilal Ayyache (0988616)

Emeka Madubuike (0948959)

Mohammed Al-Fakhri (0982745)

Ali Akhdar (1068542)

Date: 24/10/2018

Contents

1. Introduction

Lab 3 introduces the use of the LPTMR interrupt, and configuration of multiple control registers within the TWR-K60D100M. The main objective in this lab is to develop an interrupt handler routine using assembly and the concepts learned through previous labs. An interrupt or IRQ is an exception usually signaled by a peripheral, but in this lab the interrupt is generated by a software request to help understand how Interrupt handlers can be used. Exceptions are handled through the processor using interrupt service routines, fault handlers, and system handlers. These handlers are accessed using the vector table. Through Lab three, new concepts such as interrupt handlers and bit-banding were presented as well as further enhancing assembly coding practises.

1.1. Lab Description

The purpose of this lab was to investigate the functionality of a Low Power Timer(LPTMR0) interrupt to count on the terminal every five to ten seconds. In part one of the lab, the program was executed using only C-Programming. The program counts on the terminal every five seconds. The use of a callback function was restricted to help understand how interrupts can be handled using C-Programming only.

In the second part of the lab, changes were performed to the program such that the LPTMR0 IRQ service routine was completely implemented using Assembly. The routine cleared the interrupt flag and kept track of the number of interrupts using a counter which was later printed out every ten seconds of the program. To generate the appropriate address in the interrupt vector table the aliased memory location of the bit had to be calculated using the following formula: Bit-band alias = bit-band-aliased base + (byte offset × 32) + (bit number ×4). Through part three of the lab, the C-project was completely transitioned to Assembly and can now handle interrupts as seen in Figure A. 5 in the appendix.

1.2. System Requirements

During this lab, the tools and equipment that were used are enumerated below:

- Kiel Uvision Program was used to create instructions to the K60 Microcontroller
- FreeScale TWR-K60D100M Microcontroller was used to implement instructions sent
- A Hi Speed USB 2.0 Connection cable between PC and board was used to connect the Microcontroller to the PC.
- Putty displayed the results by receiving commands through the COM port in which the USB 2.0 was connected to.

2. Background

2.1. Equipment

- **Kiel Uvision Program**: The Kiel Uvision program is an IDE that combines project management, source code editing, program debugging, and run-time environment into a single powerful environment. Using this environment, the user is able to easily and efficiently test, verify, debug, and optimize the code developed. During the third lab, the debug functionality helped in understanding how the code is acting.

- **K60 Microcontroller**: The K60 Microcontroller (Figure 2.1) from NXP contains a low power MCU core ARM Cortex-M4 that features an analog integration, serial communication, USB 2.0 full-speed OTG controller and 10/100 Mbps Ethernet MAC. These characteristics makes this Microcontroller suitable to preform task in a very efficient and fast manner. A USB

Figure 2.1 1K60 Microcontroller

connection was used to sync the microcontrollerwith the Keil uVision software that that was provided by the teaching assistant. Through Lab 3, The main feature that was used was the NVIC component.

3. Implementation

As an implementation overview of this application, its required to develop an interrupt handler routine. The implementation of this lab was divided to two parts. The first one is the software part that discuss how the code implementation was done including the configuration part of all the registers and the interrupt service routine ISR or what is called an interrupt handler, many obstacles were encountered in this part, and to overcome these obstacles, the code outline was written on a paper and debugged before typing into uVision Keil for execution. The second part is basically discussing the hardware components used in the implementation of this application.

3.1. Lab Implementation Overview

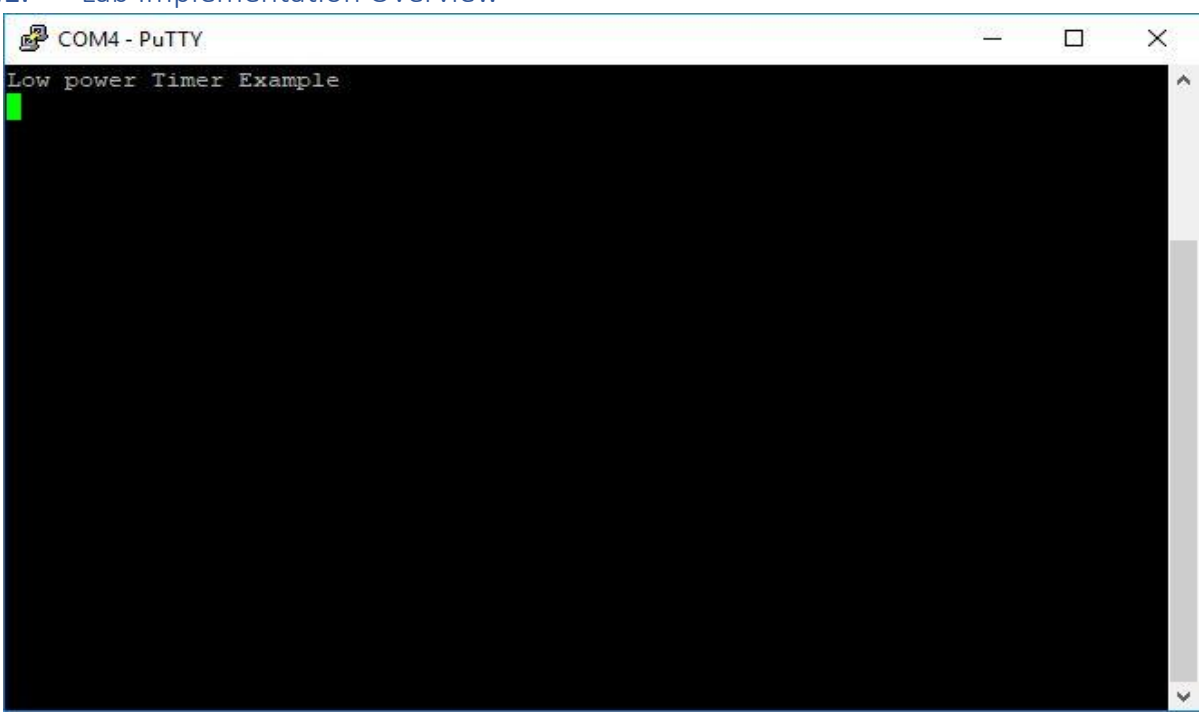


Figure 3.1 Running the Code

Figure 3.1 is considered as a startup statement that initializes the software part, this statement is a result of the first printf in the code, it describes the start of the simulation which will show that the program will begin counting the interrupts and displaying them on the screen after.

3.1.1. Software Implementation

In part one of lab three, the interrupt hander was implemented using C-Programming as seen in Figure A. 3 in the Appendix section. The Low power timer (LPTMR0) was first configured and activated. To initialize the LPTMR module, the LPTMR DRV Init LPTMR_DRV_Init() function was called. In order to use this function, lptmrUserConfig had to be defined first. This contains the LPTMR user configuration option. To set the timer's period, the function LPTMR DRV SetTime LPTMR_DRV_SetTimerPeriodUs rPeriodUs was called. This function configures the LPTMR time period in microseconds, while the LPTMR is working as a time counter. After the time period, the callback function is called. This function cannot be called while the LPTMR is working as a pulse counter. The value in microseconds (10^{-6} of a second, μs) should be an integer multiple of the clock source time slice. A LPTMR_DRV_InstallCallback function was used. This function installs the user-defined callback in the LPTMR module. When an LPTMR interrupt request is served, the callback is executed inside the ISR.

After the program in part one was debugged and properly functioning, the next task was to take out the callback function out and use assembly to implement the IRQ service routine. The routine cleaned the interrupt flag and kept track of the number of interrupts buy incrementing a counter that was later printed as value in seconds. For part three software implementation, the LPTMR0 and the hardware interrupt service rerouting was completely set up using assembly.

3.1.2. Hardware Implementation

During the third lab, Implementation of hardware was minimal. The TWR-K60D100M microcontroller was to demonstrate the code implementation. Putty terminal was used to display the results on screen

as was it communicating serially with the board this display was possible. The internal LPTMR clock was set to 1kHz with 5000 counts loaded in to interrupt every 10 seconds. No other hardware was used for this lab.

3.2. Simulation Results

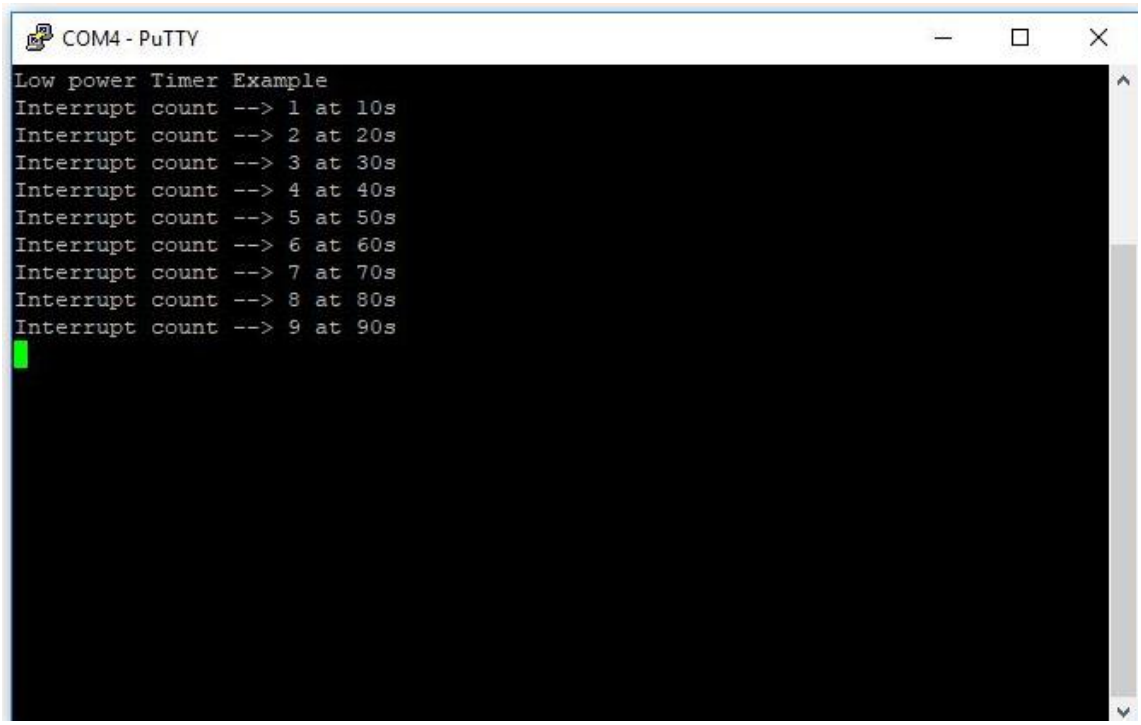


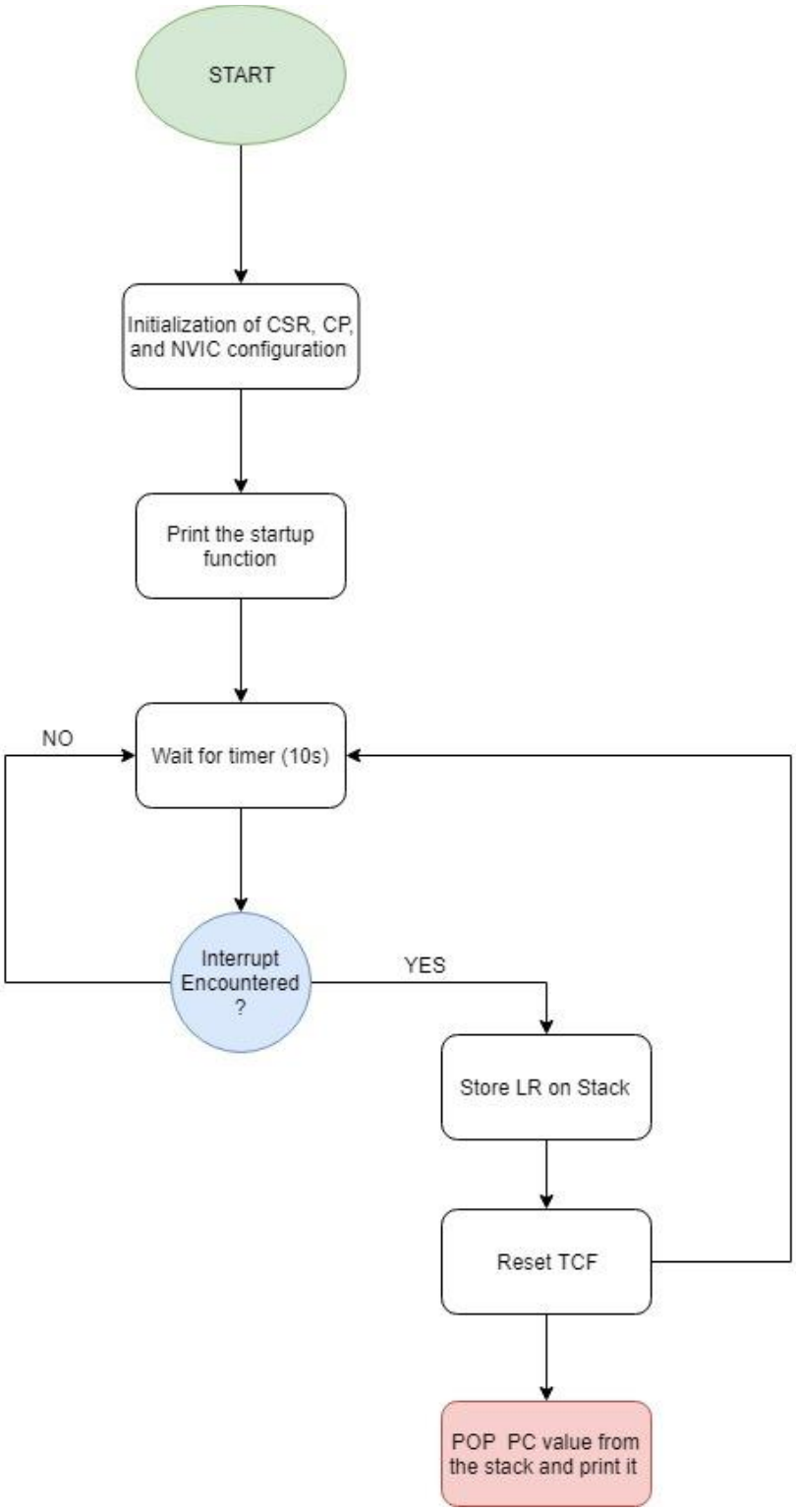
Figure 3.2. 1Results of running code with inputs

As it is shown in Figure 3.2, after we run the code, the startup sentence will pop up alone. After that the value is displayed every ten seconds and keeps going until the user resets the program.

3.3. Block Diagram

The first thing that the code does is initialize the register addresses needed to run the LPTMR, after that it prints the start-up string. Then it will wait until the interrupt is encountered which is around 10 seconds. When the interrupt is triggered, the LR is stored onto the stack then resets the TCF value. Lastly

the PC value is popped from the stack and the prints “Interrupt count → 1 at 20s ” and keeps going until the program is rested from the user .



3.4. Lab Requirements

3.4.1. Lab Requirement 1

With the given C code lptmr_example project in the examples, debugging to understand exactly how the code worked began. Through an exhaustive debugging process, it was discovered the best first step was to remove the callback function and add a line to clear the interrupt flag before incrementing the counter so that the process of the program was not affected. It was also found that changing the timer period changed the duration of time that it takes to print the interrupt statement.

3.4.2. Lab Requirement 2

With the given lptmr_example modified, run and working, requirement two asked for a change of code that would compel the IRQ service routine to be implemented completely in assembly. This change in code can be seen in Figure B.1 found in the appendix. To implement the IRQ handler the timer compare flag(TCF) in the LPTMR0 Control Status Register(LPTMR0_CSR) needs to be cleared. The address of the CSR register was calculated using the bit-band alias formula and the TCF found there is cleared. The interrupt counter is then updated to be printed out later. All of this is done as soon as an exception takes place and hence it occurs between the stacking and unstacking of the link register and program counter respectively.

3.4.3. Lab Requirement 3

For the final requirement the transition of the code to Assembly from C was to be completed. The main parts of the code; initializing the LPTMR0, setting the timers period and starting the timer were completely written in assembly. Access to the LPTMR0 was initialized by accessing the memory address where the LPTMR0 was stored and the timer's period was set by modifying the value in the CMR

register. Then the NVIC and interrupts are enabled at an IRQ number of 85 (for LPTMR0). After the completion of this the entire code is now running and functioning in assembly.

4. Conclusion

Using a LPTMR0 with the assembly code application, a fully configured working interrupt handler that displays the number of interrupts as an output was successfully implemented and tested. As a starting point the LPTMR0 was configured first by enabling the correct bits in the CSR, PSR, CMR and CNR registers making us familiar with those registers and their various functions in the LPTMR0 IRQ implementation. As it was also required to enable the system clock by enabling the bits to gain permissions and access the board peripherals, the concept of clearing or setting a bit in a peripheral device control register was introduced and understood. To conclude the configuration section, it was required also to configure the Nested Vectored Interrupt Controller (NVIC) that was responsible for handling the interrupts. An infinite loop was implemented to keep the code busy and waiting for the interrupt to happen. Finally an Interrupt Service Routine(ISR) was implemented to handle the actual interrupt, the memory address of the LR was stored in the stack first, continues the execution of the handler code till completion and then pops out the stored LR on the stack into the PC register so the program will continue from where it left off before the subroutine was called.

References

[1] Radu Muresan, "ENGG3640: Microcomputer Interfacing Laboratory Manual, Version 2", University of Guelph, July 2016.

Appendices

A. Main.c

```c
// Standard C Included Files
#include <stdio.h>
// SDK Included Files
#include "fsl_lptmr_driver.h"
#include "board.h"
#include "fsl_debug_console.h"

////////////////////////////////////////////////////////////////////////////////
// Definitions
////////////////////////////////////////////////////////////////////////////////

#define LPTMR_INSTANCE     0U

////////////////////////////////////////////////////////////////////////////////
// Variables
////////////////////////////////////////////////////////////////////////////////

volatile uint32_t lptmrCounter=0;

////////////////////////////////////////////////////////////////////////////////
// Code
////////////////////////////////////////////////////////////////////////////////

/*!
 * @brief LPTMR interrupt callback
 */
/*void lptmr_isr_callback(void)                COMMENTED OUT
 {
  LPTMR_DRV_IRQHandler(0U);
  lptmrCounter++;
 }*/

/*!
 * @brief The example uses LPTMR to generate interrupt each 1 second.
 *        When interrupt occurs, LED1 changes status & print to terminal
 */
void LPTMR0_IRQHandler(void){

  lptmrCounter++;
  LPTMR_DRV_IRQHandler(0U);
 }
```

Figure A. 1Main.c

LPTMR0 IRQ

```c
int main (void)
{

    lptmr_state_t lptmrState;
    uint32_t      currentCounter = 0;
    // Configure LPTMR.
    lptmr_user_config_t lptmrUserConfig =
    {
        .timerMode            = kLptmrTimerModeTimeCounter, /*! Use LPTMR in Time Counter mode */
        .freeRunningEnable    = false, /*! When hit compare value, set counter back to zero */
        .prescalerEnable      = false, /*! bypass prescaler */
        .prescalerClockSource = kClockLptmrSrcLpoClk, /*! use 1kHz Low Power Clock */
        .isInterruptEnabled   = true
    };

    // Init hardware.
    hardware_init();

    LED1_EN;
    // Initialize LPTMR
    LPTMR_DRV_Init(LPTMR_INSTANCE, &lptmrState, &lptmrUserConfig);

    // Set the timer period for 1 second
    LPTMR_DRV_SetTimerPeriodUs(LPTMR_INSTANCE,5000000);

    // Specify the callback function when a LPTMR interrupt occurs
    //LPTMR_DRV_InstallCallback(LPTMR_INSTANCE,lptmr_isr_callback);          COMMENTED OUT

    PRINTF("Low Power Timer Example\n\r");

    // Start counting
    LPTMR_DRV_Start(LPTMR_INSTANCE);
    while(1)
    {
        if(currentCounter != lptmrCounter)
        {
            currentCounter = lptmrCounter;
            PRINTF("LPTMR interrupt No.%d \r\n",currentCounter);
            LED1_TOGGLE;
        }
    }
}
```

Figure A. 2Main.c

```c
// Standard C Included Files
#include <stdio.h>
// SDK Included Files
#include "fsl_lptmr_driver.h"
#include "board.h"
#include "fsl_debug_console.h"

////////////////////////////////////////////////////////////////////////////////
// Definitions
////////////////////////////////////////////////////////////////////////////////

#define LPTMR_INSTANCE     0U



volatile uint32_t lptmrCounter=0;
 uint32_t counter = 0;                    //PART 2 ADDED


void LPTMR0_IRQHandler(void);



int main (void)
{

    lptmr_state_t lptmrState;
    uint32_t      currentCounter = 0;
    // Configure LPTMR.
    lptmr_user_config_t lptmrUserConfig =
    {
        .timerMode            = kLptmrTimerModeTimeCounter, /*! Use LPTMR in Time Counter mode */
        .freeRunningEnable    = false, /*! When hit compare value, set counter back to zero */
        .prescalerEnable      = false, /*! bypass prescaler */
        .prescalerClockSource = kClockLptmrSrcLpoClk, /*! use 1kHz Low Power Clock */
        .isInterruptEnabled   = true
    };

    // Init hardware.
    hardware_init();


    LED1_EN;
    // Initialize LPTMR
    LPTMR_DRV_Init(LPTMR_INSTANCE, &lptmrState, &lptmrUserConfig);
```

Figure A. 3 myMain.s

```c

    // Set the timer period for 1 second
    LPTMR_DRV_SetTimerPeriodUs(LPTMR_INSTANCE,5000000);

    // Specify the callback function when a LPTMR interrupt occurs
    //LPTMR_DRV_InstallCallback(LPTMR_INSTANCE,lptmr_isr_callback);          COMMENTED OUT

    PRINTF("Low Power Timer Example\n\r");

    // Start counting
    LPTMR_DRV_Start(LPTMR_INSTANCE);
    while(1)
    {
        if(currentCounter != counter)                //PART 2 CHANGED
        {
            currentCounter = counter;
            PRINTF("LPTMR interrupt No.%d \r\n",currentCounter);
            LED1_TOGGLE;
        }
    }
}

/*****************************************************************************
 * EOF
 *****************************************************************************/
```

Figure A. 4 myMain.s

```c

// Standard C Included Files
#include <stdio.h>
// SDK Included Files
#include "board.h"
#include "gpio_pins.h"
#include "fsl_debug_console.h"
#include "fsl_lptmr_driver.h"


////////////////////////////////////////////////////////////////////////////////
// Code
////////////////////////////////////////////////////////////////////////////////

extern void asmmain(void);


int main(void)
{
    hardware_init();

    printf("Low power Timer Example\n\r");

    asmmain();
}
void Myprintf(int d){

    //char z = 0;
    //z = x * 10;
    printf("Interrupt count --> %d at %ds\r\n",d,d*10);

}


/*****************************************************************************
 * EOF
 *****************************************************************************/
```

Figure A. 5 myMain.s

B. myMain.s

```asm
    PRESERVE8
    AREA MyCode, CODE, READWRITE
    EXPORT asm_lptmr_irq
        import counter

LPTMR0_IRQHandler EQU asm_lptmr_irq+1
    EXPORT LPTMR0_IRQHandler    ;the vector table must contain odd addresses for the Cortex processor

asm_lptmr_irq
    PUSH {lr}           ; store LR
    LDR r2,=0x40040000      ; 0x40040000, clear interrupt flag
    LSL r2,r2, #5
    LDR r3,=0x42000000      ; 0x42000000,
    ADD r2,r2,r3
    MOV r3,#7       ; 7, position bit for TCF
    ADD r2,r2,r3,LSL #2
    MOV r1, #0x1
    STR r1,[r2] ; clear interrupt flag


    LDR r1, =counter        ; increment the irq count
    LDRB r0,[r1]
    ADD r0,#1
    STRB r0,[r1]

    POP {pc}            ; Exit from ISR



    ALIGN
    AREA MyString, DATA, READWRITE



    END
```

Figure B. 1 myMain.s after Requirement 2

C. myMain.s

```
    PRESERVE8
    AREA MyCode, CODE, READWRITE
    EXPORT asmmain
    import Myprintf


; LOW POWER TIMER REGISTERS ADDRESS

my_LPTMR0_CSR EQU 0x40040000    ;CSR register address  CONTROL STATE REGISTOR: Provides a set pending bit for the NON-Maskable Interrupt exception
my_LPTMR0_PSR EQU 0x40040004    ;PSR register address
my_LPTMR0_CMR EQU 0x40040008    ;CMR register address
my_LPTMR0_CNR EQU 0x4004000C    ;CNR register address
my_SIM_SCGC5  EQU 0x40048038    ;System Clock Gate Control Register (bit 0)
NVIC EQU 0xE000E100

; CODE
asmmain

    LDR r2,=my_SIM_SCGC5    ; 0x40048038, enable software access to LPTimer
    MOV r1,#0x00000001      ;Mask
    LDR r0,[r2]
    ORR r0,r0,r1
    STR r0,[r2]


    LDR r2,=my_LPTMR0_CSR
    LDR r2,=0x40040000
    MOV r1,#0x00000040      ;initialize CSR value
    STR r1,[r2]
    ADD r2,#0x4

    MOV r1,#0x00000005      ;set psr
    STR r1,[r2]
    ADD r2,#0x4

    MOV r1,#0x2710              ;set counter value (change later to make the timer correct)
    STR r1,[r2]
    SUB r2,#0x8

    MOV r1,#0x00000001      ;Mask
    LDR r0,[r2]
    ORR r0,r0,r1
    STR r0,[r2]

NVICConfig

    MOV r2, #0x00200000 ;0x00200000 sets IRQ 85
```

Figure C. 1. myMain.s after Requirement 3

```
    LDR r1, =NVIC ;0xE000E100 address of nvic reg
    STR r2,[r1,#0x8]

 loop B loop

  AREA myarea, CODE, READONLY
  EXPORT asm_lptmr_irq
 LPTMR0_IRQHandler EQU asm_lptmr_irq+1
  EXPORT LPTMR0_IRQHandler   ;the vector table must contain odd addresses for the Cortex processor


 asm_lptmr_irq
    PUSH {lr}            ; store LR
    LDR r2,=my_LPTMR0_CSR       ; 0x40040000, clear interrupt flag   0x40040000
    LSL r2,r2, #5
    LDR r3,=0x42000000       ; 0x42000000,
    ADD r2,r2,r3
    MOV r3,#7        ; 7, position bit for TCF
    ADD r2,r2,r3,LSL #2
    MOV r1, #0x1
    STR r1,[r2] ; clear interrupt flag

    ; RESET OUR CNR HERE VALUE FOR IT TO WORK



    LDR r1, =counter         ; increment the irq count
    LDRB r0,[r1]
    ADD r0,#1
    STRB r0,[r1]

    bl Myprintf
    POP {pc}             ; Exit from ISR


    ALIGN
    AREA MyCode, DATA, READWRITE

 counter  DCD    0x00

    END
```

Figure C. 2. myMain.s after Requirement 3