LAB #2
ECE 3140/CS 3420

Divyansh Garg (dg595) and Caitlin Stanton (cs968)

INTRODUCTION

The point of this lab was to work with a timer to produce a certain output (an LED flash of a certain color) once a condition was met (the timer ran out). That conditional was checked through polling and interrupts. In order to properly understand how to complete the lab, there had to be understanding of the aforementioned polling and interrupt techniques, as well as memory-mapped I/O and writing C programs.

PART 1

The first part of the lab included accomplishing the following: setting up a timer, checking to see if the timer ran out, and, when it did, toggling a red LED to turn on and off so that the LED would change its state approximately once a second.

This timer was a periodic interrupt timer, specifically PIT[0]. The clock first had to be enabled to the PIT module, and this was done through the System Integration Module (SIM) at SCGC6. We also had to set the MCR register to 0x00 and the load value of the clock to a some value which was equivalent to 1 sec of real time. We finally set the TCTRL register to PIT_TCTRL_TEN_MASK to enable the timer. These register values and the general setup for the timer were written in the void function ***timer()***.

An additional part of the setup involved enabling the red LED such that it could be toggled on and off as necessary. This was written in the void function ***setup()***. Much like the PIT timer, the clock had to be enabled through the SIM, specifically to Port B (the macro for which was stored as SIM_SCGC5_PORTB_MASK). Within Port B, the specific pin for the red LED is PTB22, and that had to be enabled as GPIO pin (essentially making it an output pin on the board). The appropriate control signals were set using the pin control register (PCR) and PDDR, ensuring that the 22nd pin (red LED) would be known as an output.

We also wrote separate functions to turn the red LED on and off. For ***LEDRed_On()***, we set the 22nd bit of the PCOR register and indicated, using a global variable "**red_led**", that the LED had been turned on by setting it equal to 1. For ***LEDRed_Off()***, we set the 22nd bit of the PSOR register and set "**red_led**" equal to 0 to show that the LED was turned off.

In our ***main()***, we used the "**red_led**" variable in our polling. Polling is a protocol that uses a loop to constantly check a value and see if it fulfills a certain condition; it's simple to write, it's hard to scale and can be very slow and wastes CPU cycles. Polling was implemented with a while loop that runs forever and some nested conditionals, which check to see if the timer flag (TFLG) was set. If the flag was set, that meant that the timer had expired and the flag needed to be cleared for the next iteration of the timer. To check whether the LED had to be turned on or off, the **red_led** variable from earlier was checked: if equal to 1, the LED had to be turned off; if equal to 0, the LED had to be turned on.

The combination of these helper functions and their use in our ***main*** created a program that used repetitive checking of a one-second timer to toggle a red LED on and off.

PART 2

Similar to Part 1, this section of the lab used a PIT timer to toggle several LEDs to flash. However, one LED (blue) was toggled based on the running of a loop, and one LED (green) was toggled using interrupts.

Starting with the ***setup()*** and ***timer()*** functions, they had nearly identical functionalities to those of Part 1. In ***setup()***, the blue and green LEDs were prepped to be toggled through enabling the clock to Port B and Port E, and making PTB21 and PTE26 GPIO pins, respectively. The same registers for those steps were used -- PCR and PDDR -- and the PSOR was set for each pin so that the LEDs were maintained at an off position from the start. For ***timer()***, the only difference between the methods in Part 1 and 2 was that in Part 2 the TCTRL register had to be set equal to TIE (to enable interrupts) and OR'd with the value TEN to enable the timer.

The blue LED had to be separately toggled every second through a loop, so in ***blue_toggle()***, we used a for loop with an extremely large counter value in order to repetitively turn the LED on (setting the 21st bit of the PCOR), keep it on for around a second (using a while loop function ***space()***), and turn the LED off (setting the 21st bit of the PSOR).

This section required using interrupts in order to toggle the green LED to be on for a tenth of a second once every second, when the PIT interrupt is triggered. Interrupts are performed through the hardware, which wait for a specific trigger and then handles that trigger accordingly. In this case, the trigger was the countdown of the timer. The interrupt was handled in the ***PIT0_IRQHandler()*** function, enabled with the line *NVIC_EnableIRQ(PIT0_IRQn);* in the ***main()***.

Within the handler, first the TFLG for the PIT timer was cleared, and then a global variable "**flag**" was checked (which indicated the status of the green LED). If it was equal to 0, then the green LED would be turned on and the LDVAL for the PIT timer would be modified to be the equivalent of one tenth of a second, and if it was equal to 1, then the green LED would be turned off and the LDVAL for the PIT timer would be modified to be the equivalent of nine-tenths of a second. The LED was turned on and off by setting the 26th bit of the PCOR and PSOR registers of PTE, respectively.

To modify the load value of the timer in between of its cycle, we need to disable it, modify LDVAL, and again enable it. To disable the timer, the TCTRL register was AND'd with ~PIT_TCTRL_TEN_MASK. Similarly, to enable it we OR'd it PIT_TCTRL_TEN_MASK.

WORK DISTRIBUTION

The first step we took was to make certain that we knew the differences in structure and implementation between polling and interrupts. After reading through the assignment, we were aware that the first part would use polling and the second part, which was a bit trickier since it dealt with two LEDs, would use interrupts.

We determined a general overview of the functionality that each program needed to fulfill, and relied on pair programming to complete the project. One of us would be coding while the other was combing through the various reference manuals for the FRDM board and C programming. The work was, for the most part, completed with both of us working together, but to read through code and catch any errors that one of us missed, we shared code over GitHub before meeting up later.