Lab 2 Report

Helen Liang (hl973)
Raghav Kumar (rk524)

Work Distribution

Design

In designing each part of our program, we came up with subtasks we wanted to complete first and therefore laid out a plan before we began coding.

For the first part, we started off with the guidelines provided in the lab manual and got familiar with the different ports, pins and their structure. Even before starting part two, we had configured and enabled all three LEDs, making sure they were running all right. We created different functions to organize and maintain our source file, hence we have functions like ‘set_up_RED’, ‘LEDRed_on’, ‘LEDRed_Off’ etc. Starting from the main function, we first enabled the clock to the PIT module. Although the PIT module contains several channels, we primarily worked with Channel 0. Using macros from the ‘fsl_device_registers.h’ header file, we enabled the clock in Channel 0 and loaded it with some time that we thought would amount to about a second. We then went on to set up the red led using the ‘set_up_RED’ function. This function enabled the port (port B) that the red LED was connected to. We also had to initialise the 22nd pin of Port B and we did that using the macro that was made available to us. Since we were a bit unsure of bit shifting, we resorted to using macros for part one. To toggle our red LED, we used a local variable called ‘state’. Before entering the loop, we initialized ‘state’ to 0. This corresponded to the ‘On’ state and the function ‘toggleRed’ would call ‘LEDRed_on’ to turn on the red led. Similarly, if the ‘state’ was 1, ‘toggleRed’ would call ‘LEDRed_off’ and the LED would turn off. Going back to the main function, inside our while loop, we kept checking for the TFLG flag. If the 0th bit of the flag was a 1, it meant that the timer did time out and an interrupt should be raised. As it can be observed, the following interrupt would be raised using the classical polling type interrupt checking because inside our while loop, we continuously check for the TFLG flag each time we enter the loop. If the TFLG flag is not set, we don’t do anything and just loop back to check again. If the flag is set however, we write one to the TFLG flag clearing it in the process. Now our TFLG flag is back to how it was before the interrupt was raised, ready to receive a new interrupt. Using the 0th bit of the TCTRL register, we enable and disable the timer so that the interrupt generating process can continue. Finally, we toggle the LED while also changing the state so that each time an interrupt occurs, the LED either lights on or off. We tried setting the timer value such that interrupts got raised every second meaning that the LED changed state every second.

For the second part of the lab, we had to toggle both the blue and green lights. We decided to implement and test it incrementally, so we first wrote the code to set up blue and green lights and turning them on and off. To do this, we created separate ‘set_up_BLUE’ and ‘set_up_GREEN’ functions as well as separate ‘LEDBlue/Green_On/Off’ functions. Because we realized that we would need to toggle green and blue lights rather than directly calling on/off, we created toggle functions ‘LEDBlue/Green_toggle’ for each light, which took in an integer state as a parameter. For the green LED, we decided to make the variable holding the green state as a global variable, so we could later change/access it in the interrupt handler as well as the timer countdown. Therefore, we were able to change the physical state of the LED light based on what the current integer state is in our code. We tested these functions to ensure they were working properly before starting the actual second part of the lab.

In order to toggle the blue LED light once every second, we created a local variable that held the state of the blue light in our main function, since the body of the loop for the blue light is in main. As the lab release required, we ran a for loop as time delay, and on the last iteration of the loop, the blue LED light is toggled to either on or off. In order to keep the for loop running forever, we wrapped it inside of an infinite while loop so that the blue state toggles once every second forever. To determine the number of loops needed for a one-second delay, we tested various values of different magnitudes until we found a value that was about one second.

For the green LED, we first had to enable everything to get the timer set up. This consisted of enabling the clock to the PIT module and enabling the standard timers by setting the first bit of MCR to 0. We then called our ‘set_up_GREEN’ function we had completed beforehand to set up the green LED light. The initial green state is set to off so that it only turns on after the timer has ended and an interrupt has been generated.

Noticing that we needed a timer to countdown a duration of both 1 second and 1/10 of a second, we created a ‘timerCountdown’ method that was capable of counting both 1 second and 1/10th of a second. Depending on what the state of the green LED light was, the countdown timer would load the corresponding value of 1 second or 1/10th of a second. If the state was on, then the timer would countdown for 1/10th of a second, and if the state was off, the timer would countdown for 1 second. We also made sure to clear the interrupt flag as well as enable the timer interrupt and timer again in ‘timerCountdown’ so we can receive new interrupts.

Whenever the timer reaches 0, an interrupt would be generated and the ‘PIT0_IRQHandler’ would handle this interrupt. This signifies the end of a state, so inside of our IRQ handler, we toggled the state of the green LED and then called ‘LEDGreen_toggle’ to display the state. At the end of our IRQ handler, we called our timer countdown again in order to restart the timer.

Coding

We used the discussion slides, lab handout, and the Kinetis K64 manual as directions for coding. We also referred to the header file for macros we could use that would prevent us from having to work with the actual bits.

In part 1 of the lab, we used macros we found to enable the clock to PIT module, enable the timer, write a 1 to clear the timer interrupt flag, and set MDIS of MCR (1st bit) to 0. We found the macros to be easier to use than shifting individual bits, as when we shifted individual bits, the red light wasn't turning off.

For part 2 of the lab, however, the macros didn't work. We may have used them incorrectly, but the light wasn't turning on or off. Once we changed the macros to manually changing the individual bits, the program worked. We also made sure to use 'or equals' instead of just 'equals' if we're writing a 1 to a bit so the other bits are maintained. To accomplish this, we shifted a '1' into the correct position, according to the Kinetis K64 manual, that we wanted to change. If we needed to write a zero, we just set the value to 0.

We also decided to separate things into functions as much as possible in order to keep our code clean. Therefore, for part 1, we had the functions 'set_up_RED', 'LEDRed_On', 'LEDRed_Off', and 'toggleRed' in addition to the main function. For part 2, we had the functions 'set_up_BLUE', 'set_up_GREEN', 'LEDBlue_On', 'LEDBlue_Off', 'LEDBlue_toggle', 'LEDGreen_On', 'LEDGreen_Off', 'LEDGreen_toggle', 'timerCountdown', and 'PIT0_IRQHandler' in addition to the main function.

Code Review

For code review, we tried to keep certain things in mind while looking over the code. Since we had to work with 3 different LEDs, it was very easy to mess up the ports or the pins while setting up or toggling the LEDs. So whenever we went over our code, we always made sure that we were referring to the right pins and ports.

We also tried making sure that we didn't mess up on the syntax since it was the first time that we were using C in this class. Missing a semicolon or a type declaration was our most common type of syntax error and we tried looking out for those. We also made sure that we were accessing the pins, ports, and registers in the proper way as suggested by the Kinetis K64 reference manual.

Whenever we wrote a certain segment of code, we made sure to mentally go through the control flow and make sure that our logic was correct. We had to be careful regarding the control flow of toggling the LEDs since we were using the state variable.

While doing code review, we also paid attention to the way we were enabling and setting up the various registers as defined by the reference manual. Sometimes as in part two, the ordering of the statements mattered and hence we paid special attention to such cases.

Testing

For testing, since the two parts of the labs consisted of turning on LEDs for a specified amount of time in a specified way, we tested our code manually by looking at the LEDs. We also tested our code incrementally so that we were able to isolate more easily what part of our code was malfunctioning. For example, we wrote the code to turn on and off all three LEDs before implementing the corresponding sections of the lab to make sure that the LEDs were functioning correctly. Therefore, if they didn't turn on/off, it must have been a problem with the timer or delay and not the LED itself.

We used a stopwatch to verify the time for part 1 red and part 2 blue. Because human reaction time is slow and the LEDs change quickly, once every second, we decided to check over an interval of 10 state changes so that if there was an error, it would be magnified over time. Through guess and check, we were able to find a value to load into the timer that was roughly equivalent to one second.

To debug, we set breakpoints in our newly written code and stepped through each line to see if the program was behaving properly. When we ran into a problem of the LED not turning off for part 1, we also learned to change the load time to see whether the load time was just simply too small and the human eye couldn't detect the changes or that there was an actual bug in our code.

Documenting/Writing

Given that we were working with C statements from the reference manual that we didn't quite understand, it was imperative to maintain good documentation throughout this lab assignment. From day one of starting the lab, we tried commenting on each line of code with its functionality. This not only helped us catch bugs but also reminded us of our original logic when we came back to the code later on.

For the lab report, we worked on it together. We brainstormed for about ten minutes and jotted down the broad points and headers that we wanted the report to be sectioned in. After deciding on the broad sections, both of us added material to each section before we actually started writing it. Once the skeleton of the report was laid out, we took individual sections and completed them. Since Raghav had done most of the physical coding for part one, he wrote the first part of the design section. Helen physically coded part two of the lab assignment and hence she wrote the second part of the design section. We then equally contributed to writing all the other sections.

Collaboration

For collaboration, we implemented the peer programming method we used in lab 1. One person would code while the other one said what to code. We always met up together to work on the lab, and we never worked on the lab alone without the other person in order for both of us to understand each part of the lab.

First, we would both read through the lab write-up individually and then we would discuss and create a rough online of all the tasks we needed to do. For example, for part 1, we determined that we needed to first figure out how to turn on the red LED, then how to turn off the red LED, then how to step up the clock and enable the timers, how to clear the flag, and lastly find the right load value. We came up with this to-do list together, and then we worked together to figure out how to implement this. This consisted of bouncing ideas back and forth and trying different things.

Specifically, for peer programming, Raghav typed up the first part of the lab while Helen said what to type while Helen typed up the second part of the lab while Raghav said what to type. We found this method highly effective as both the typer and speaker had to be fully aware of what's going on with the code. If we had any problems, we also went to office hours together so both of us could understand the problem and solution.

Since only one of us had the board, we always met together to do the coding and testing. This way we also ensured that both of us were involved in the entire lab and knew what was going on. We communicated over messenger and explained any new ideas before meeting up again. We also used messenger to decide on what time worked for both of us.

Finally, both of us went through the Kinetis K64 manual as well as the discussion slides together whenever we hit a roadblock. Whoever found something useful would convey it to the other. Sometimes one would code while the other would have the manual open so that he or she could quickly tell for example what port or pin to address in that particular line of code.