ECE/CSE 474

Rania Hussein

Ishan Dane & Anael Aguayo-Chong

10/31/2022

LAB 2 Report

**Section 1: Procedure**

**Task1:**

In Task 1a we had to turn the onboard LEDs on and off in a periodic pattern. The LEDs change at a rate of 1 sec by polling a GPTM. We started by setting up the header file. We configured the RCGCTIMER register, GPTMCTL register, GPTMCFG register, GPTMTAMR register, and GPTMTAILR register utilizing the datasheet to find the addresses and corresponding offsets. To activate the LEDs, we used our code from Lab 1 to set up the RCGCGPIO and ports F and N, where port F was for LED3 and 4 and port N was for LED1 and 2. We used the configured registers in our main file to initialize the timer and used the timer to poll the GPTM. We enabled the TIMER0 bit in the RCGCTIMER register by setting it to 0x01. We also disabled the TIMER0 by using the GPTMCTL by setting it to 0x0. We used the GPTMCFG to select the 32- bit mode, which uses Timer A and B. We then configured the GPTMTAMR register to periodic timer mode by setting it to 0x2, which would allow the timer to constantly cycle through the desired time. We set up GPTMTAILR to 0x00F42400 which allows TIMER0 to achieve a 1Hz blink rate. After setting the desired time, we enabled TIMER0 which sets the initialization for the timer. We then enabled Port F and Port N in the PortF\_init method to initialize the LEDs. We set GPIODIR and GPIODEN for both PortF and PortN to their corresponding bits. After initializing the LEDs and the timer, we created a timer handler that cleared the flag on the timer when the timer had timed out using the GPTMRIS and GPTMICR registers. Once the timer runs out, it then exits the while loop and resets the TIMER0\_ICR, which clears the flag for the timer. We used this delay for the alternation between the LEDs in our pattern method. Inside the pattern method we called the timer before setting the LED, and before changing from one LED to the next.

In Task1b, we reused the header file and added another timer (timer1), GPIOs for PortL, GPIOs for pushbutton 1 and 2, and added our methods used in main. We started setting up the registers for timer1 the same way we did previously for timer0, enabling them by going through the datasheet and finding their bases and offsets corresponding to the ports. We decided to use 2 timers to make our timer manipulation easier. We set TIMER1 as a 5-second timer and TIMER0 as a 2-second timer functioning as a one-shot timer. The 5-second timer is being used for alternating between the LEDs causing a delay as a periodic timer. The 2-second timer is used for the button presses, where the buttons must be held down for 2 seconds before activating any LEDs. The way we achieved 5 and 2-second timers were in their initialization methods under the GPTMTAILR. We set this register to 16 MHz to achieve a 1 Hz blink rate, we then multiplied 16MHz by 5 which gave us a 5-second timer, and repeated this for the 2- second timer. After the GPTMs were set up we moved to GPIOs for ports L, F, and G, where PortL is for the LEDs, PortF is for pushbutton 1, and PortG is for pushbutton 2. The GPIOs that were defined include GPIODIR, GPIODEN, GPIOAFSEL, GPIODATA, and GPIOAMSEL. Similar to the GPTMs, we found the GPIO bases and offsets from the datasheet. The last thing we included on the header file was all of the methods used in our main, in order to initiate them before using them in the main. After we set up the header file, we moved on to the main. We created a method that would activate the clock for ports L, G, and F as well as configure the ports to the LEDs and switches. From our previously configured TIMER0, we initialized it once more but changed the 5-second TIMER0 to TIMER1 and set up another timer initialization for TIMER0 which is our 2-second timer. We then implemented them into our previous traffic light system using timer interrupt handlers, that were used to clear the flag, when the timer would run out. This allowed us to reset the timer every time the method was called on within our logic. From the FSM seen in Fig.1, we were able to identify where to include the timer resets according to the current state we were in, and to what state we wanted to go. We would start in our idle state (LEDs off, waiting for the on/off button) once it was clicked it would start in the stop state. If the system recognized that it was being turned off, then it would go back to idle, if the system did not receive any command during the 5 seconds it would move to the go state. While in the go state there are 3 possibilities: nothing and it goes back to the stop state, the on/off button pressed and held would turn the system off, and if the pedestrian button was pressed and held it would go to warn state. We included a TIMER1 reset every time we would go from state to state so it would start the 5-second countdown, before changing to another state. With everything configured, the logic in our methods allows for the system to function properly.

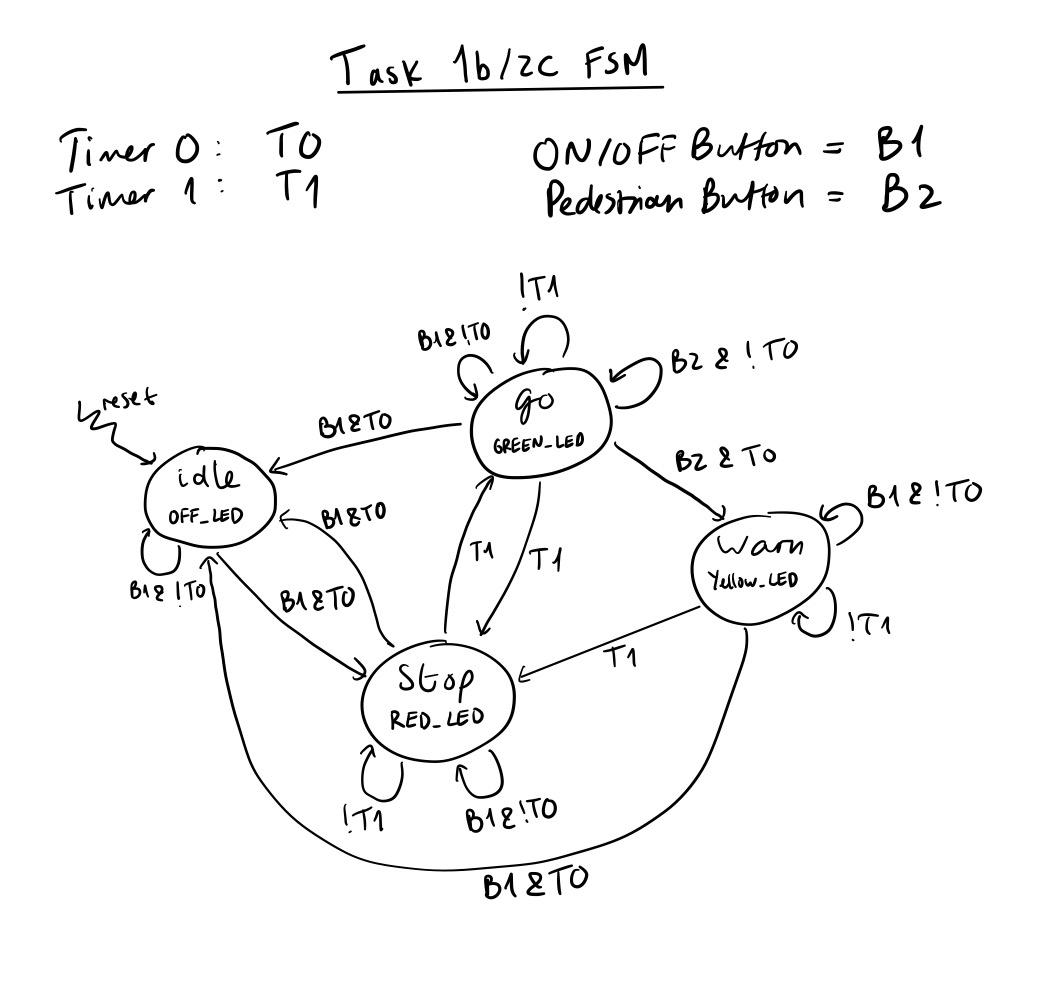


Figure 1: FSM for tasks 1b and 2c

**Task2:**

In Task 2a we want to turn on and off the LEDs in a periodic pattern, but use interrupts as the delays. We first started with the vector table code that was provided for us and added the timer interrupt handlers 1A and 0A and ports F and N. The vector table requires that you place the handlers and the ports according to their vector numbers and addresses for each interrupt. Once all of the ISRs are set we saved the cstartup\_M.c file to the project, and debugged the file. Then building on our previous code for the header file we added the NVIC EN0 interrupt for Port F and Timer 0. We also needed to add the Priority register and GPIOMIS register for PortN interrupts. Like before we used the GPIO ports F and N for the LEDs portF for LEDs 3 and 4 and portN for LEDs 1 and 2. Once that was set, we moved on to the main, where we were to implement logic to allow the LEDs to periodically alternate from one to the other. We defined the LEDs at the top by setting them to their corresponding bit callout. To start, we first enabled PortF and PortN GPIOs and initialized the ports and LEDs in our initialization IO. After initializing this we initialized the timer, which is set to a one-second blink rate and includes interrupt NVIC EN0, GPTMIMR, and GPTMCTL. Having everything configured allowed us to move on to the logic. Where we set it in Timer0A interrupt handler, so when a LED is detected to be on it will then turn off and change to the next LED. This allowed for the LEDs to alternate in a periodic pattern.

For Task 2b we are controlling the timer from a user switch. The LED1 will blink at a rate of 1 second and SW1 and SW2 will interrupt the system. When SW1 is pressed, the timer should stop counting down, and LED2 should turn on. When SW2 is pressed, LED2 will turn off and the timer should resume counting down and the alternation will resume. After adding the interrupts to the vector table we moved on to the header file. Reusing our previous code from Task 2a all we added were PortJ GPIOs and PortN GPIOs. This port is being utilized for user switches 1 and 2. As in task 2a when setting up the GPIO registers we also set up the Priority registers and GPIOMIS. After configuring the needed ports and interrupts, we had to define the LEDs and switches to their appropriate pins. Our next step was to initialize the interrupts, the ports, and the timers. Once this was set up, we added logic into the timer0A which clears the timer flag and used exclusive OR to create the blinking effect. The logic in the PortJ handler has two different functions depending on which button is pressed. If switch 1 is activated then the LED2 will turn on, the interrupt will be turned off, and then clear the switch 1 flag. If switch 2 is activated then the LED1 will be turned on in its blinking periodic state, the timer interrupt will resume, and the switch 2 flag is cleared. With this logic, the system will work as described above.

For Task 2c, we were tasked with repeating the same traffic control system as Task 1 only this time with interrupts. In task 1 we used cases to control the LED behavior, instead now we are setting our logic in our PortF, PortG, Timer0A, and Timer1A interrupts. The first step was including Port G and TIMER1A into our vector table as they were new interrupts to our system. Port F and Timer0A were already configured and included in the vector table from previous tasks. We then added Port G GPIO and Timer 1A configurations and interrupt configurations into our header file using the datasheet provided to retrieve the values necessary. We set up our system to use Port F PF3 as the ON/OFF button and Port G PG0 as the PEDESTRIAN button much like we did in task 1b. From our knowledge of interrupts, we knew that we could essentially design the entire traffic light system through our port handlers and timer handlers and therefore making the system more efficient as nothing happened in the while(1) loop. Our first step was creating the timer initializations for timer 1 and timer 0. We decided to set timer 0 as our 2 second timer and timer 1 as our 5 second timer. We chose to set timer 1 as periodic due to the nature of its function: alternating between red and green every 5 seconds and yellow to red if the pedestrian button was clicked and held. For timer 0, we decided to configure it as a one shot timer as we only needed it to run when a button was clicked. Therefore we also decided to leave it off in the configuration as it would only be turned on when necessary (button clicked). Using information from the datasheet, we configured and initialized both timers and their interrupts so that they would interrupt the system and run their respective handler methods when countdown was finished. After initializing the timers, our next step was to set up the timer handlers and to ensure that they worked. We first set up Timer 1’s handler to clear the timer flag and then use the appropriate logic to check which LED was currently being lit up. Our configurations were: If the RED LED was on when interrupted, the handler would turn off the RED LED and turn on the GREEN LED. If the GREEN LED was on when interrupted, the handler would turn off the GREEN LED and turn on the RED LED. This created the alternation between the go and stop state that we wanted. We then included logic for the pedestrian button case where: If the YELLOW LED was on when interrupted (if the pedestrian button click was successful), the system would transition to the RED LED and then the alternation would begin as normal from our previous logic. We then tested our logic to ensure it worked by manually setting the LEDs to begin in the stop state in the main and seeing if the alternation between Green and Red occurred which it did. Our next step was first to set up the port handlers without the 2 second button press to turn off and on the system, and switch to the warn state when the pedestrian button is pressed (like in task 1b). We had to ensure that the buttons worked before implementing the 2 second button hold. In both PORTF (ON/OFF) and PORTG (PEDESTRIAN) handlers, we used the GPIO MIS register to check whether an interrupt has occurred by their respective buttons. If an interrupt occurred in PORTF (ON/OFF BUTTON), we would then transition to the RED LED state if the system was off or turn off all LEDs if the system was on, and then clear the interrupt flag. If an interrupt occurred in PORTG (PEDESTRIAN BUTTON), we would transition to the YELLOW LED state, restarting the 5 second timer, and then back to the RED LED state after the 5 seconds was done. After we tested and debugged our logic for the LEDs and button presses without the 2 second timer, we were ready to move to implementing it. To create the 2 button press hold for the system, we first changed the button port handlers to: 1) Initialize and start timer 0 on either button press and clear the interrupt flag. All our handler logic had to now go into our Timer 0 handler which would act when the timer was done and the system was interrupted. Upon Timer 0’s interrupt, the handler would check whether the button was still being pressed. If the pedestrian button was still being pressed, the handler would then transition into the YELLOW LED state, clear the button interrupt, turn off the 2 second timer and then reset the 5 second timer. If the ON/OFF button was still being pressed, the handler would clear the button interrupt, and then check whether the system was in the OFF state (All LEDs off) or the ON state (An LED is on). If the system was in the OFF state, the handler would then turn on the RED LED, restart the 5 second timer and turn off the 2 second timer. If the system was in the ON state, the handler would then turn off all LEDs and then turn off both timers. Through this logic, and setup of handlers, we were able to implement the same FSM system we previously had in Task1b and allow it to work exactly the same but with much less code, no polling and only interrupts.

**Section 2: Procedure**

**Task1:**

In task 1a we were making the LEDs alternate in a periodic pattern. Where we had the LEDs change at a rate of 1sec by polling a GPTM. We started by making a timer handler that would check if the timer had run out, if it hadn't then it would stay in the while loop until the timer ran out. This would then leave the while loop and go into the next step, ultimately causing a delay. Since we wanted to create an alternating pattern of the LEDs we created a method called pattern. That would cycle through the LEDs and when it would be alternated it called the timer handler to cause a delay of one second.

In task1b we were repeating the timed traffic system we had previously done in lab 1, only this time using timers. We decided to set two separate timers so it would be easier to manipulate and use in our code. We set a 5-second timer to use for the alternating LEDs from the go state (green LED) to the stop state (red LED) and if we were in the warn state it would take 5-seconds to change to the stop state. The other timer was used for the button hold, if it did not satisfy this condition it would not move on to the next state. Within our states we had logic defining what the LED behavior would be. We initialized the timers and immediately went into the idle state, where it would sit until the on/off button was detected. Once the button was pressed and held for 2-seconds it would go into the stop state. If none of the buttons were pressed the 5-second timer would run out and switch to the go state. If the system detected the on/off button was clicked and held then it would go back into the idle state. If the system detected the pedestrian button was clicked and held it would not do anything for the system would already be in the stop state. If the timer ran out and no button was clicked and held for 2-seconds it would then go into the go state. Once in the go state if the system timed out from the 5-seconds and no button was clicked it would go into the stop state. If it detected the on/off button was clicked and held for 2-seconds it would go into the idle state turning the system off. If the pedestrian button was pressed and held for 2-seconds, the system would go into the warn state and wait for 5-seconds. After the 5 seconds and the on/off button was not clicked and held it would go to the stop state.

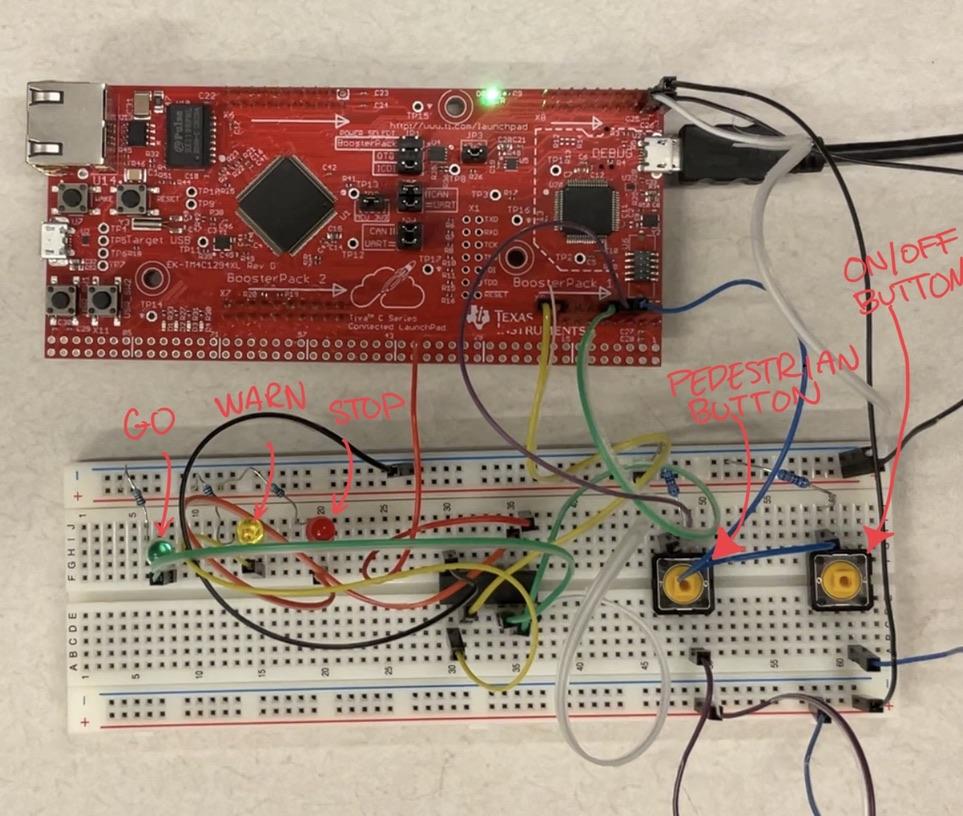
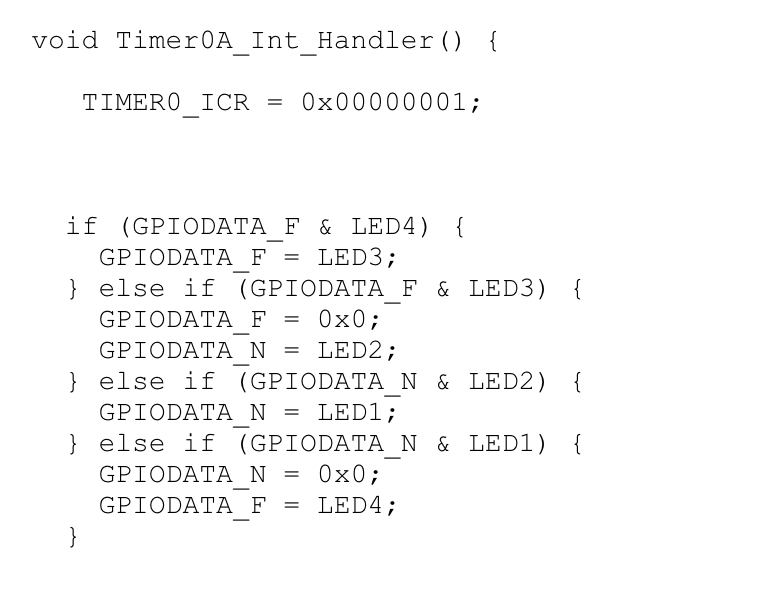
****

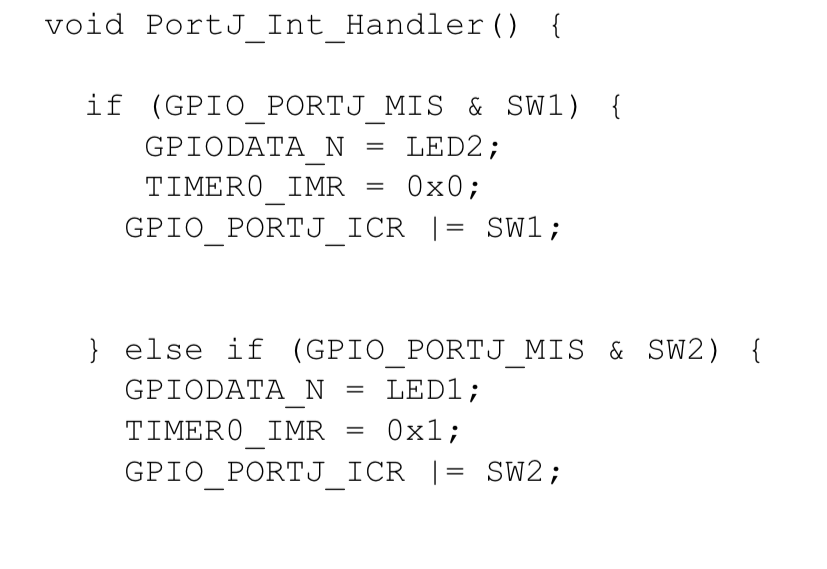
Fig. 2: Traffic light system

**Task 2:**

In task 2a we were turning on and off the LEDs in a periodic pattern using interrupts. We first set up the vector table with the required interrupts. Once the vector table was done and debugged we set up the incorrect handlers for the switches and realized we did not need this. We then got rid of them and set our logic in the Timer0A handler that took care of creating the periodic pattern. As seen in our code in Fig.3.

Figure 3: Timer0 handler creating a periodic pattern.

In task 2b we controlled the timer from the user switch. The LED1 is blinking at a rate of 1 second. When SW1 is pressed, the timer will stop counting down, and LED2 should turn on. When SW2 is pressed, LED2 will turn off and the timer should resume counting down and the alternation will resume. We did not experience many complications other than logic debugging. Once we figured out our POrtJhandler as seen in Fig.4 the system was able to work.

Figure 4: PortJ handler logic.

In Task 2c, we encountered problems when it came to initializing and re-initializing our timers. We had initially thought that we could just turn off the individual timers using GPTMCTL and then turn them back on. However, this caused the timers to turn back on keeping the same count and not restarting. To fix this problem, we realized that we had to reinitialize the entire timer and reload it from the start to reset it. Our next problem was we were unable to figure out how to implement the 2 second button hold. We had placed most of our transition logic in our port handlers and then were stuck on trying to transition after the 2 second button hold. Through reading the datasheet and testing different things, we realised that by putting our transition logic inside the timer itself and starting the 2 second timer on button interrupt, we were able to check if the button was still pressed after 2 seconds and transition to the appropriate state. Besides these issues, we did not face too many other ones besides small debugging mistakes and syntax errors. This was because by task 2c, we had become much more comfortable with using interrupts through handlers.