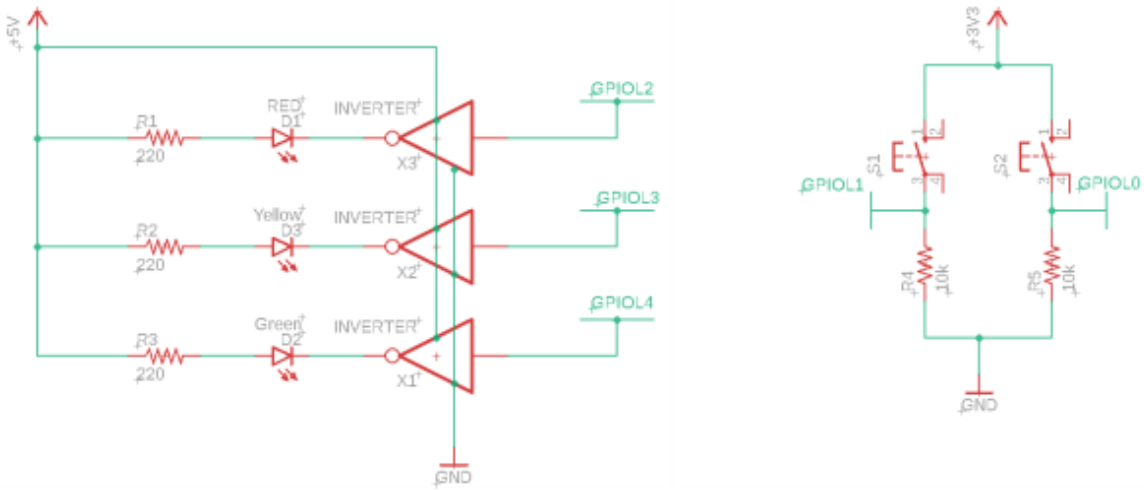
EE474: Laboratory Report #2

Section 1: Procedure

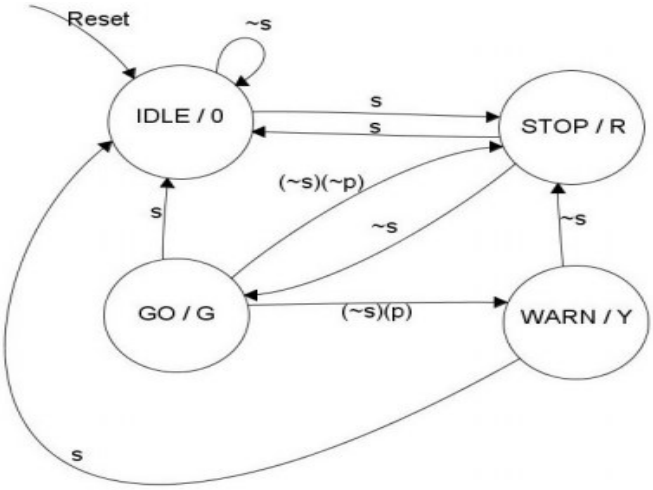
Task 1A

After learning how to use the timers in the *TM4C1294NCPDT* board from the laboratory document, task 1A reduced to replacing the *delay()* function implemented in laboratory #1. First, a timer (*TIM0*) is configured in periodic down-counting mode with a frequency of 1Hz. Then, the *delay()* function from laboratory #1 is replaced by a *while* loop that polls the timer's *raw interrupt status* until it has timed-out. The time-out flag is cleared and the next step in the LED sequence is taken. This process is repeated for every step in the sequence indefinitely.

Task 1B



The traffic light controller from laboratory #1 is modified for this task. First, three timers are initialized. One with the purpose of measuring the time when the pedestrian button is held, another for when the start/stop button is held, and finally one for measuring the time since the last transition. Each timer is configured as periodic down-counters, the timers related to buttons have a period of 2 seconds, and the one for measuring time since last transition has a period of 5 seconds. A state transition is normally produced after 5 seconds after the last transition, unless (i) it is caused by pressing *pedestrian* button while in the *GO* state for 2 seconds, which causes an immediate transition or (ii) the start button is held for 2 seconds during *IDLE* which causes an immediate transition that makes the machine more responsive than before.



Since now the requirement is that each button is registered as an input only if it has been held for two seconds, the question of what happens when a button starts being hold before a state-transition and continues being hold after the transition. I noticed that holding the start/stop button for two seconds is never ambiguous since when the system is not *IDLE* the effect is always to stop the system. Therefore, the *start/stop* button counter is not reset during a transition. However, holding the pedestrian button can be ignored during *STOP* but must be considered during *GO* which makes the situation ambiguous. I chose to reset the *pedestrian* button timer to avoid ambiguity, thus the user must strictly press the button for two seconds during the *GO* state to trigger a transition into *WARN*.

Task 2A

First, the initialization procedure is similar to task 1A, with the addition that the timer is configured to produce an interrupt signal after each time-out event. The ISR related to this timer keeps track of the next step to take in the LED sequence, and performs one step after each time-out. The main loop of this task is simply waiting for the next interrupt. I initially had an empty *while* loop for waiting, but then I added a *WFI* instruction using in-line assembly to reduce power consumption.

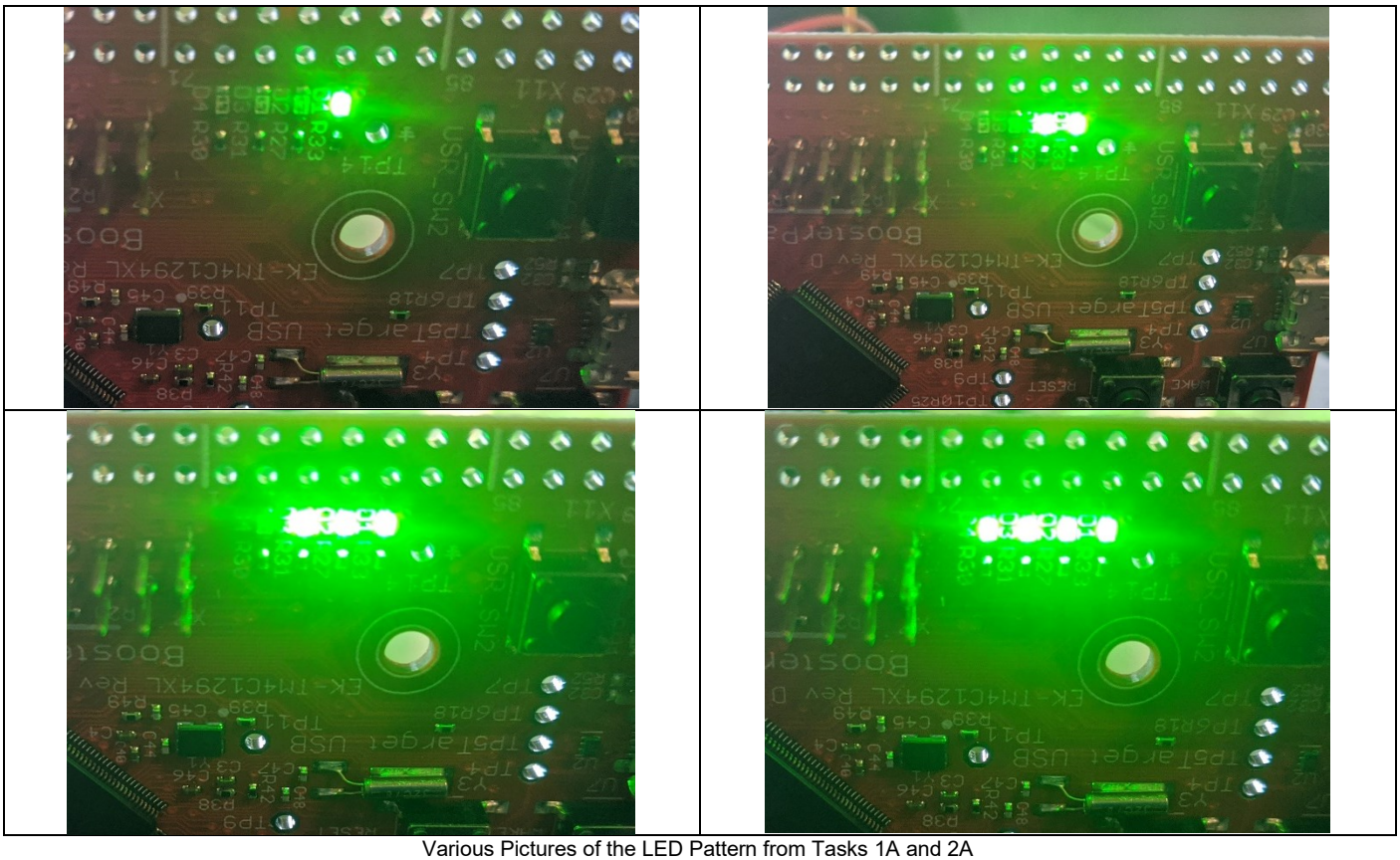
Task 2B

For this task, a timer is configured as a periodic down-counter with a frequency of 1Hz which triggers an interrupt on time-out events. The ISR of this timer is used to toggle LED D1 to produce the blinking behavior. The GPIO pins and ports that connect to SW1 and SW2 are configured to read the push buttons and to trigger interrupts based on falling edges. When SW1 is pressed, the ISR runs and stops the timer from counting, turns D1 OFF and D2 ON. Pressing SW2 has the reverse effect of letting the timer count so that D1 has the blinking behavior (toggles at each time-out with a frequency of 1Hz), while D2 is OFF. To avoid having nested interrupts, the priority is the same for both the timer and the GPIO ISR for this task. This way, the software will behave as if each input arrived one after the other, avoiding bad interleaving.

Task 2C

Task 1B is modified first by configuring each timer to produce an interrupt based on time-out events. The ISR for each button (start/stop and pedestrian) will execute if and only if the button has been held for two seconds straight. Therefore, these ISR will register the corresponding button as pressed and clear the corresponding interrupt flag. The timer ISR for measuring the time since the last transition will simply request a transition to the FSM. The FSM thus receives a transition request and moves to the next state. Just like before, a transition can also be triggered immediately by holding the pedestrian button for two seconds during the *WARN* state, or by holding the start/stop button for two seconds during the *IDLE* state. Observe also that the priority is again set to be the same for each ISR. However, in this case there is no possibility of a bad interleaving since each ISR does not affect the other nor puts the system on a partially incorrect state at any point, thus having different priorities is totally fine. I just found no reason to give a different priority to each ISR in this task.

Section 2: Results



Various Pictures of the LED Pattern from Tasks 1A and 2A

Task 1A

The system is successfully displaying the pattern required by the laboratory document. The on-board LEDs are being turned on sequentially and in order of increasing LED number (D1, D2, D3, D4), and then being turned off in the opposite order. The frequency is 1Hz, thus each step in the sequence is taken after one second. The task was done by polling a general-purpose timer.

Task 1B

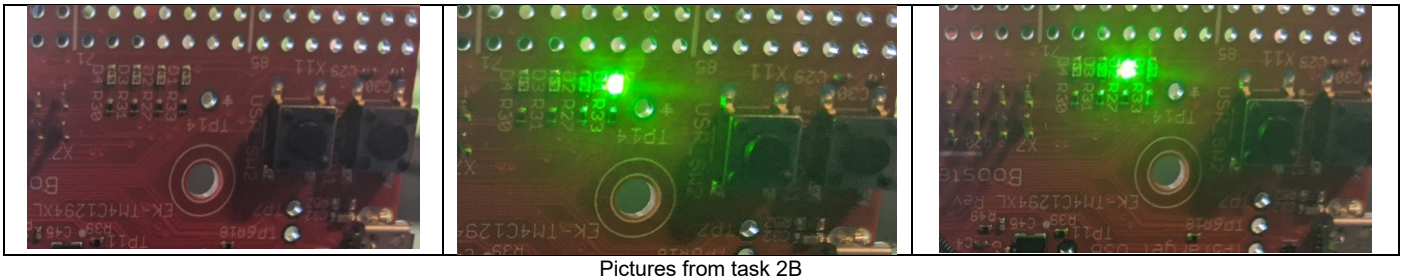
The system is successfully meeting the design specifications and simulates a traffic-light controller. Each button is registered as input after two seconds of being held. The system after being started oscillates through the *GO* and *STOP* state, and each transition takes 5 seconds. The start/stop button has the effect of turning off the system (Any state→ IDLE), or turning ON the system (IDLE→STOP). The pedestrian button has the effect of triggering a transition to the *WARN* state immediately when registered as an input in the GO state. Then it stays in *WARN* for five seconds before moving to the STOP state. This task is done by polling general purpose timers.

Task 2A

The system successfully performs the periodic pattern described in the laboratory document. The LEDs are turned on sequentially as in Task 1A. However, this task was developed using a timer interrupt whose ISR manages the LED behavior. The main code is simply putting the microcontroller into sleep until the next interrupt wakes up the processor.

Task 2B

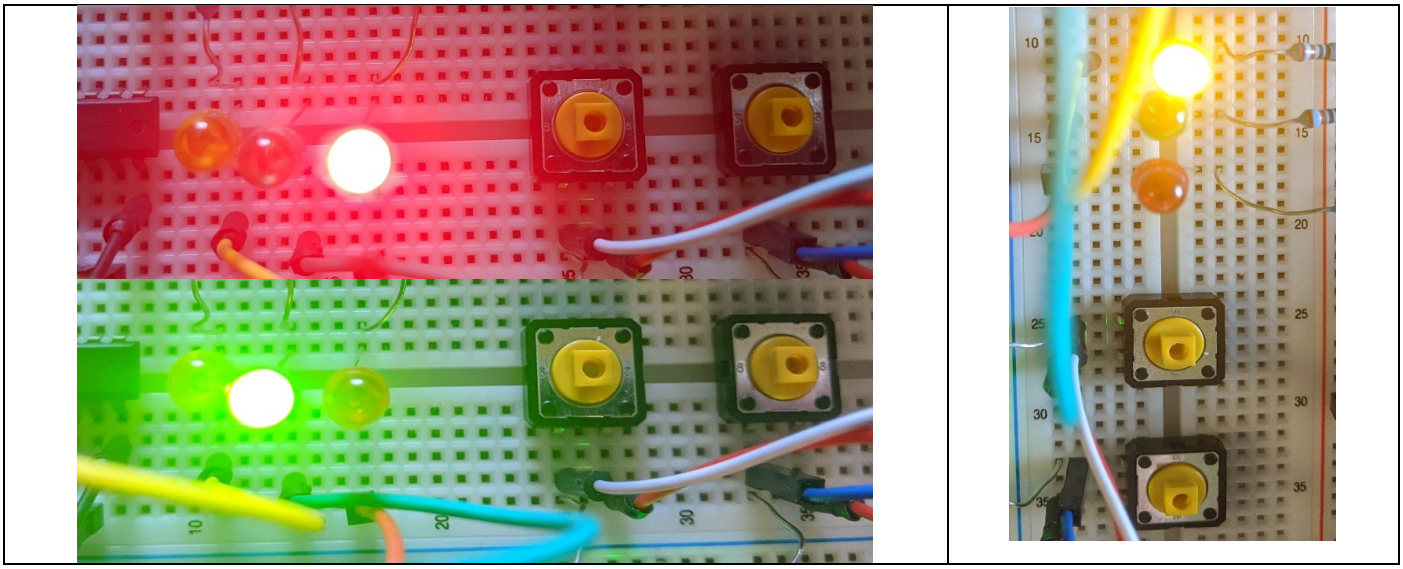
The system correctly meets the specifications and requirements. The system uses timers to blink LED D1 at a frequency of 1Hz. When SW1 is pressed, the timer stops counting and thus the blinking behavior stops and turns LED D2 on. Then when SW2 is pressed, the system starts again the blinking behavior of LED D1 and turns of LED D2.



Pictures from task 2B

Task 2C

The system has success replicating the behavior of task 1B simulating a traffic-light controller. This time, the task is performed using timer interrupts instead of polling the timers. One can see and test that each button has to be held for two seconds, that the start/stop button causes the system to start in the STOP state where it oscillates with the GO state which gets switches every 5 seconds. The start/stop button also turns of the system, and the pedestrian button causes an immediate move to the WARN state if registered as input during the GO state. Therefore the task was completed satisfactorily.



Pictures of the traffic-light controller from tasks 1B and 2C

Section 3: Problems Faced & Feedback

One issue that I had during the laboratory was that sometimes the pedestrian button for the traffic-light controller was having the correct effect, but not always. I was able to overcome this issue by debugging through my program using the TASTALL bit to stop the timers from counting while the processor is halted by the debugger. I realized that sometimes the program would come back to the main FSM after it had requested a transition, but since it had already evaluated the code for the next-state transition then the system wouldn't go to WARN. The issue was that I had originally planned for the ISR related to the pedestrian button to register the input, but I ended up programming logic that would request a state-transition. Having fixed the issue, now the ISR simply registers the input as described previously in the present document, and the next-state logic is what triggers the premature state-transition request and the transition is then guaranteed to occur after the next-state logic is decided.

Feedback about the laboratory assignment is positive. I believe we had a very good introduction to learning about timers and how they can be used to solve problems. We also had an effective introduction about interrupts which can be used to avoid polling. Once again, we had more opportunities to reading the datasheet and learning how to program our TM4C1294NCPDT board. I think in particular having this ability will make us effective engineers in the embedded field.

One tip or trick that I can remember for future laboratories is that some peripherals have debug options that can be enabled. For instance, I used the TASTALL bit to stop the timers from counting when the processor is halted due to the debugger. This made debugging the programs developed for this laboratory much easier.