# EE474: Laboratory Report #4

## Section 1: Procedure

### Task 1A

The first function that I completed was *LCD_GPIOInit()*. Using the knowledge that we have acquired this quarter, I was able to follow the comments and initialize the GPIOs as appropriate. The only difference is that since the supplied drivers make reference to the original "tm4c1294ncpd.h" file provided by TI, we were allowed to include such file for this single task. Fortunately, having experience manipulating registers from the scratch makes it very easy to use TI's file. As a note, I re-named TI's file to "official_tm4c1294ncpdt.h" to avoid a name conflict with my customized header file "tm4c1294ncpdt.h," which is used for the rest of the laboratory.

The second function was *LCD_PrintFloat()*. The first thing to do is to understand that this function will be printing real values with two decimals. To achieve this, *LCD_PrintFloat()* multiplies the input by 1000 and added five for rounding, then divided by 1000 using integer arithmetic to obtain the whole part of the input. Then, the lowest three digits of the input multiplied by 1000 are extracted to obtain the first 3 decimal places of the input. The result is printed with two decimals making use of *LCD_PrintInteger()* to print the whole and fractional parts, and *LCD_PrintChar()* to print the decimal dot.

### Task 1B

The first thing to do is to learn how to print with the correct format using the LCD screen. This was done in isolation. After learning how to print on the LCD screen in isolation, the task became a matter of integrating the LCD screen to extend the laboratory #3 assignment. First, all the components for UART are removed. Due to the modularity of my implementation, integrating the LCD was as simple as adding initialization code to the initialization function, and then changing the call to *UART_SendData()* for equivalent *LCD_PrintString()*, *LCD_PrintFloat(),* and *LCD_Printf()* calls as well as modifying the string format to satisfy the requirements.

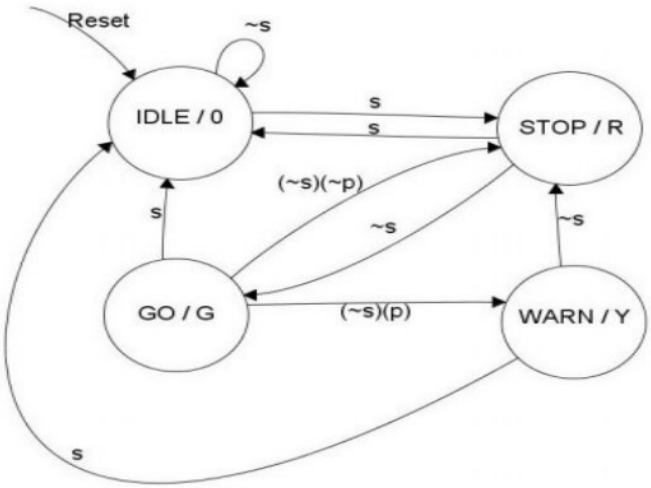
### Task 1C

I started by testing the LCD touch-screen functionality in isolation. To do so, I started by drawing circles to represent buttons until I found a position that I thought was appropriate. Then I proceeded to label the buttons so that the user knows which button implements which functionality. Lastly, I printed the coordinates being read on the touch-screen, so that I could decide the region covered by each button. Knowing how to interface with the touch-screen in isolation, all I had to do was to integrate with the result from task 1B. Again, due to the structure of the code, it became as simple as removing the components for the switches, adding initialization for the touch-screen, and then replacing the mechanism for detecting button presses. Now, instead of having an interrupt handler for detecting button presses, the virtual buttons are checked at the top of the main loop for this task.

### Task 2A

After having experience with the LCD and its touch-screen functionality, the main focus is on integrating these to the traffic light controller from laboratory #2. As in task 1C from the present laboratory, the first thing was to draw on the LCD the start/stop button, the pedestrian button, and the lights in appropriate positions and with reasonable colors. Then, I had to test again the limits of the buttons by printing coordinates of the current touch. After deciding on the visual features, I went ahead and integrated this into the traffic light controller. First, I removed the code for the buttons and the lights. Then, I extended

the initialization function to initialize the LCD and its touch-screen functionality. Now, I changed the output of the FSM to draw/clear virtual buttons instead of the GPIO pins for the LEDs. Finally, the main loop for this task reads from the touch-screen to determine if a virtual button is being pressed, thus the rest of the logic remains the same as the original traffic light controller. The FSM is shown below.




**Task 2B**

I started by reading the provided *main.c* file and its comments to have a general understanding of the structure of this program. Then I proceeded to look up the documentation of the FreeRTOS APIs called within *main.c*. Now I was able to follow the comments in the provided *main.c* to implement each of the FreeRTOS tasks required for this assignment. The selected priority was the same (0) for each task, and pre-emption was disabled. The priority is the same because none of our tasks has a deadline that is tighter than other tasks. Regarding disabled pre-emption, the idea is that there is no need for any task to pre-empt each other in this simple system, so the over-head of pre-emption is avoided. This also simplifies logic. For instance, if pre-emption were enabled and a button task pre-empts the control task it might raise the question of whether it might become ambiguous if the button press belongs to the previous or next state. The FSM is the same task 2A, thus the same diagram is used (see above).

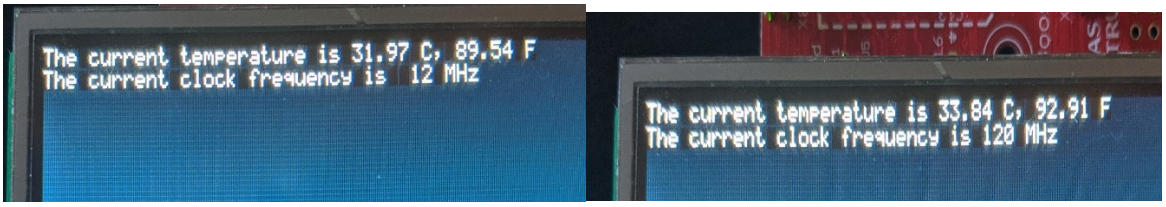
## Section 2: Results

**Task 1A**

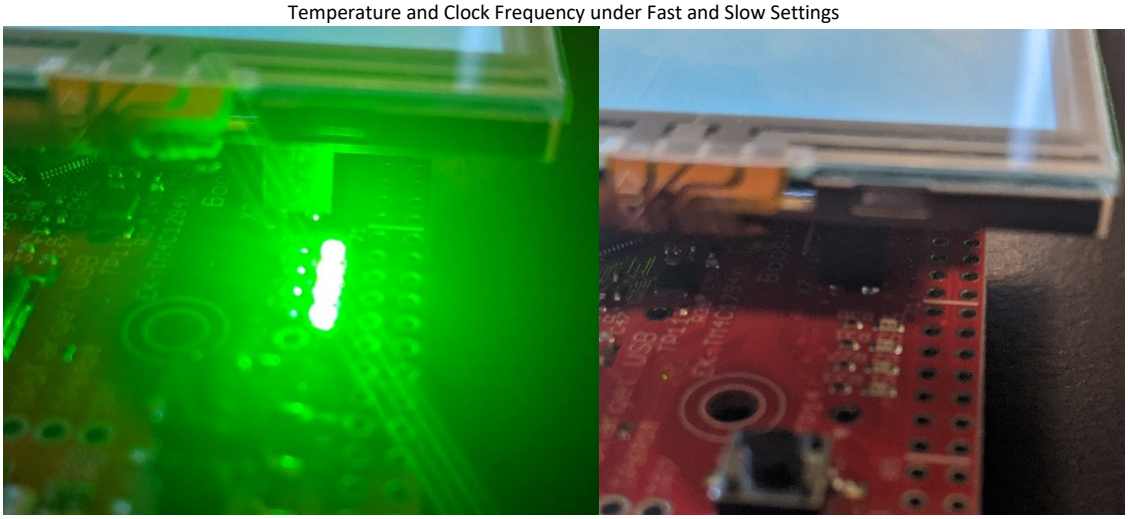
The LCD driver was completed successfully. The function *LCD_GPIOInit()* is able to initialize the GPIOs utilized by the LCD, which can be evidenced in the success of the next tasks. Similarly, the function *LCD_PrintFloat()* is able to print float values with two decimals. This can be evidenced through the success of tasks 1B and 1C.

**Task 1B**

This task was completed successfully. The temperature sensor is read and then displayed on the LCD display, in the format “The current temperature is {temp in C} C, {temp in F} F” and the clock frequency is also displayed as “The current clock frequency is {freq} MHz”. The temperatures are displayed as floating-point numbers with two decimal places, and the frequency is an integer. All the functionality of the equivalent task in laboratory #3 is still preserved, as one can see the LEDs flashing depending on the current temperature.



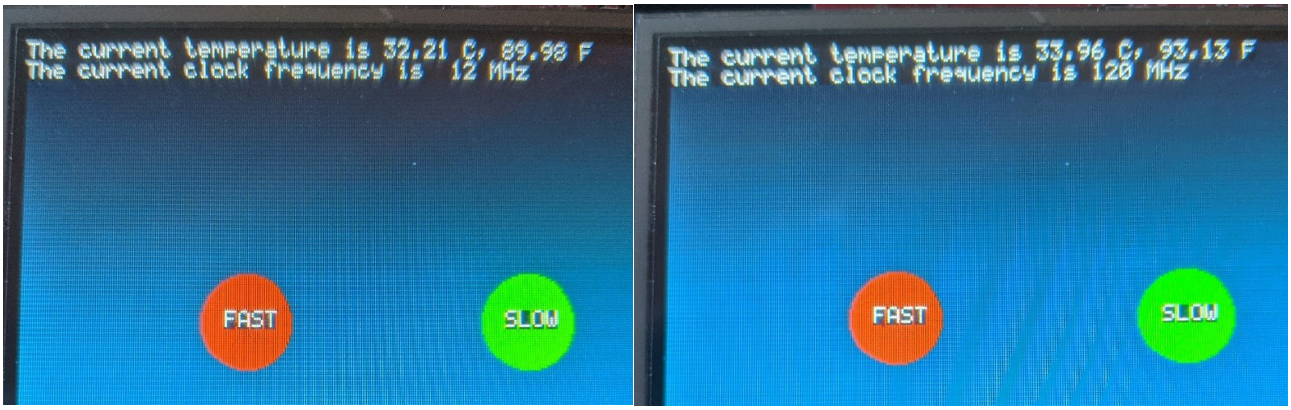

Temperature and Clock Frequency under Fast and Slow Settings



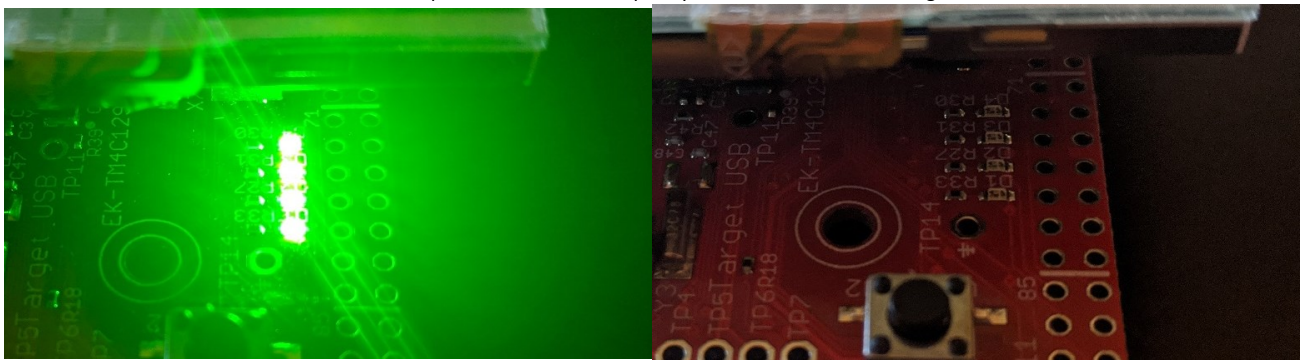
The LEDs are still flashing based on temperature

**Task 1C**

The task was completed with success. The touch screen was integrated and now the temperature and clock frequency are displayed on the LCD screen, and the buttons are now virtual. The user can press the virtual buttons and the system will use the selected clock frequency. All the functionality from task 1B and the equivalent laboratory #3 assignment is still preserved, as one can still see the LEDs flashing depending on the current temperature.




The temperature and Clock Frequency under Slow and Fast Settings

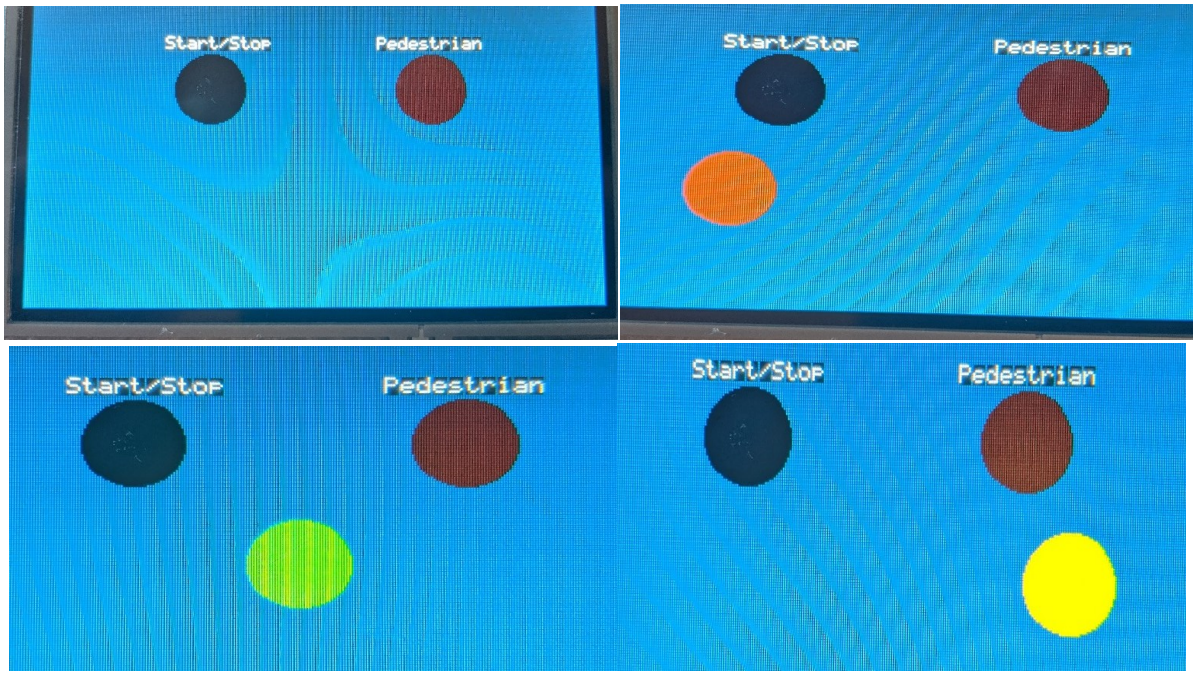


The LEDs flashing based on temperature

**Task 2A**

The system meets the specifications outlined in the laboratory document with success. The virtual traffic light controller behaves as the original traffic light controller from laboratory 2, but uses virtual traffic

lights and virtual buttons. Each button must be pressed for two seconds to count as input. Pressing the start/stop button will start the system or stop the system if it was already running. The pedestrian button will move the system to the WARN state with the yellow virtual light if the system was in the GO state with the green virtual light. Otherwise, the system keeps cycling between STOP (red light) and GO, staying 5 seconds in each.




The Virtual Light Traffic Controller. When seen counter-clockwise starting at the top-left: Idle, Stop, Go, and Warn states.

**Task 2B**

The system replicated the behavior of the traffic light controller with success while using FreeRTOS. The system has virtual buttons and virtual lights. Each button must be pressed for 2 seconds to count as an input. Pressing the start/stop button will start or stop the machine, pressing the pedestrian button will put the machine in WARN state (yellow light) if it was on the GO state (green light). Otherwise, it cycles between STOP (red) and GO (green), staying at each for five seconds. The images are essentially the same as task 2A, and so are not repeated here.

## Section 3: Problems Faced & Feedback

The biggest challenge from this laboratory was to select an area for the buttons. These screens are resistive and are of appropriate quality for our purposes, but most of us are used to high-quality screens (i.e.: because of our personal cellphones/tablets). Thus we assumed at first that we would be able to press and get a very accurate reading with our fingers. I was printing the values read and I noticed that they would vary a lot. I googled and was able to understand that it was expected, that these values would depend on the pressure with which you press, and that you would preferably use a fine-tip with these screens. In addition, the TAs also made similar suggestions on the Piazza board when more students asked. Therefore, I was able to overcome this problem by using the stylus from my surface and by printing the coordinate values on the screen so that I was able to choose appropriate values.

Feedback about the laboratory assignment is positive. We had the opportunity to work with touch screens which I think is a really useful device. In addition, we had the opportunity to use an RTOS (FreeRTOS), which is an essential skill for the embedded systems engineer.

Since this is the last laboratory, I wouldn't have tips for myself for future laboratories. However, an important tip for the future could be to practice and don't forget the content learned this quarter.