EE474: Laboratory Report #1

Section 1: Procedure

Task 1

Part 1 – Making a header file

The header file *tm4c1294ncpdt.h* provided by the course staff was removed as instructed. Then I began to write my *tm4c1294ncpdt.h* header file. Here are the major details:

- Includes <*stdint.h*> to standardize the size of integer datatypes, which is required to ensure that each variable's length is known. Furthermore, it helps to define structures for register definitions. For instance, I wrote a register structure definition for GPIO which requires each datatype to be 4 bytes long to ensure that the compiler produces correct addresses for each register. Here is a snippet of the GPIO register structure definition:

```
1.  typedef struct {
2.      __R uint32_t __res0[1];
3.      __RW uint32_t PIN0;
4.      __RW uint32_t PIN1;
5.      __R uint32_t __res1[1];
6.      __RW uint32_t PIN2;
7.      __R uint32_t __res2[3];
8.      __RW uint32_t PIN3;
9.      __R uint32_t __res4[7];
10.     __RW uint32_t PIN4;
11.     __R uint32_t __res5[15];
12.     __RW uint32_t PIN5;
13.     __R uint32_t __res6[31];
14.     // ... More registers
15. } GPIO_RegDef_t;
```

Which then make it possible to define a pointer to a *GPIO_RegDef_t* as follows:

```
-   #define GPIOA           ((GPIO_RegDef_t *) (GPIO_PORTA_BASE))
```

- Includes "tm4c1294ncpdt_config.h" which is a configuration file that I might add configuration macros if I need to. For example, I will also be writing drivers for the peripherals that we will be learning during this quarter, therefore I have written a configuration macro that tells the preprocessor to enable or disable *assert()* calls to aid in debug.

```
-   #define ASSERT_ENABLED              (0U)
```

Then *tm4c1294ncpdt.h* can include <*assert.h*> based on the value of *ASSERT_ENABLED*:

```
1.  #if (ASSERT_ENABLED)
2.  #include <assert.h>
3.  #endif
```

Similarly, driver files can enclose assert statements in *#if* preprocessor directives to enable or disable assertions:

```
#if ASSERT_ENABLED
assert(GPIO_IS_GPIO(pGPIO));
assert(IS_COMMAND(Command));
#endif
```

- Contains base addresses of registers and pointers to registers. These can be obtained by reading the datasheet and finding the correct addresses:

```
#define SYSCTL_BASE                     0x400FE000UL

#define NVIC_BASE                       0xE000E000UL

#define GPIO_PORTA_BASE                 0x40058000UL

#define SYSCTL_RCGCGPIO            (*((__vo uint32_t *) (SYSCTL_RCGCGPIO_ADDR)))
#define SYSCTL_SRGPIO              (*((__vo uint32_t *) (SYSCTL_SRGPIO_ADDR)))
#define SYSCTL_PRGPIO              (*((__vo uint32_t *) (SYSCTL_PRGPIO_ADDR)))
```

- It also contains important constants used in the laboratory solutions that are specific to the tm4c1294ncpdt board. For example, masks commonly used through registers or the port of specific peripherals:

```
#define GPIO_PIN0_MASK              (0x1U << 0U)
#define GPIO_SW_PORT                GPIOJ
#define GPIO_D1_D2_PORT             GPION
```

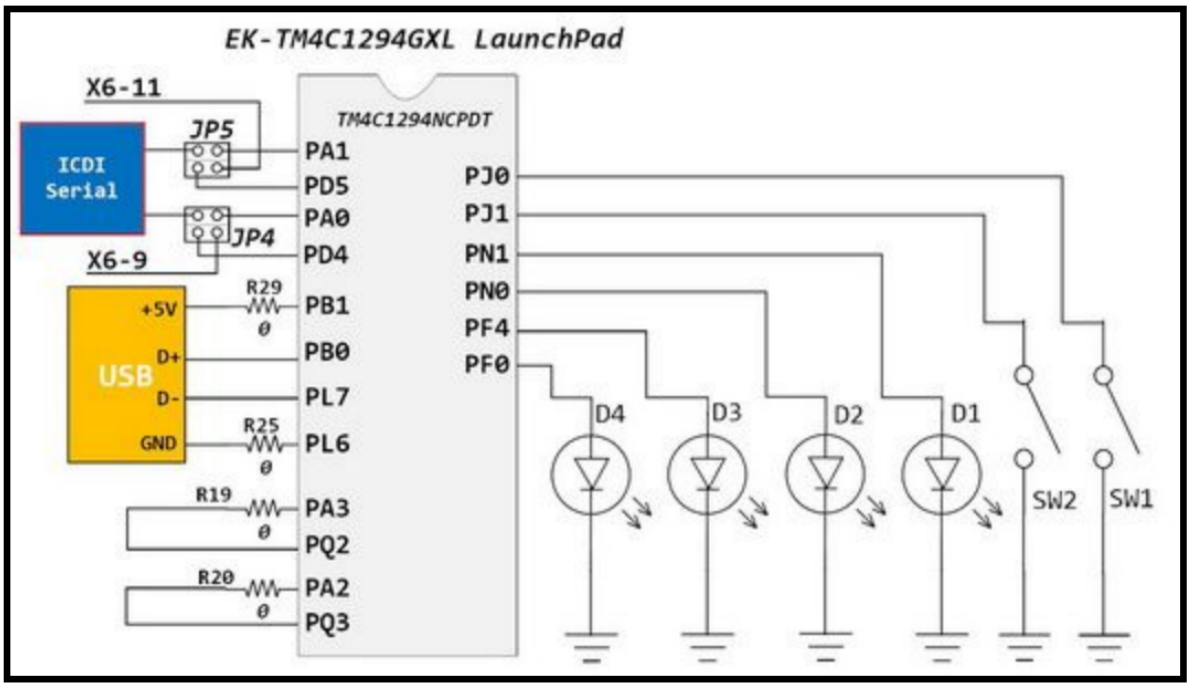
Part 2 – Modifying the provided program to turn on and off the LEDs in a periodic pattern.

Given that at this point the laboratory document had guided us on how to turn on the LEDs, the next thing to do is to decide on a periodic pattern. I decided to turn the LEDs in increasing order based on their LED number (D_1, D_2, D_3, D_4), and then turning them off in reverse order with some delay in-between. For that, I first refactored the code into a *lab001.c* file that chooses which task to run, a *task1.h* header file to define the constants and function declarations used in this task, and a *task1.c* file that implements these functions. Then I have declared a list that contains pointers to the GPIO ports that contain each LED as well as the pin number corresponding to each LED. This list is traversed through a *for* loop after the GPIO ports and pins related to the LEDs have been initialized through *Task1_Part2_Init()*. Additionally, some delay is provided by keeping the CPU busy through a *Delay()* function.

Another important observation is the use of the *PRGPIO* register in the System Control *(SYSCTL)*. Since there might be a small delay after the clock to a GPIO port is enabled, I noticed that the *PRGPIO* register provides information when a port is ready to be accessed. Therefore, I am reading this register before initializing the GPIO ports used in this task.

Part 3 – Controlling the LEDs using the user switches.

The following figure provided in the laboratory document shows the GPIO pins associated to each user button:



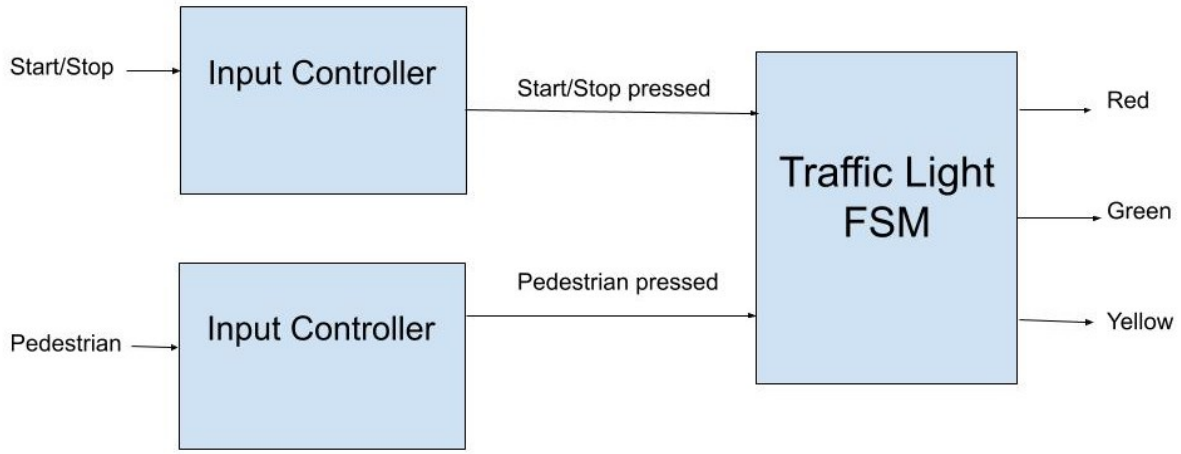
Observe from the figure that the pin associated with each user button is floating when the switch is open. Therefore, it is necessary to use a pull-up resistor to provide a normally-high state to the pin and then it gets pulled to ground when the switch is closed (the button is pressed). Upon inspection of the datasheet, it became clear that the PUR register can enable internal pull-up resistors for any desired GPIO pin.

The code for this section includes a *Task1_Part3_Init()* initialization function that configures the GPIO pins associated with LEDs D1 and D2 as digital outputs and the GPIO pins associated with the switches SW1 and SW2 as digital inputs with internal pull-up resistors. Then the principal section *Task1_Part3()* is simply an infinite loop that reads the value of the switches and turns on the LED D1 if switch SW1 is pressed and LED D2 if switch SW2 is pressed.

Task 2

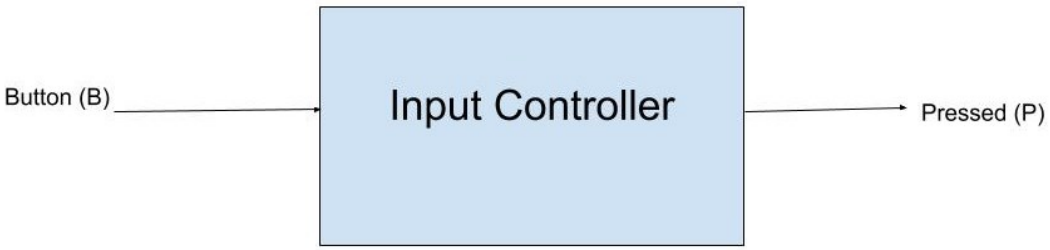
Let us start with the conceptual design of this machine. First, the block diagram of this machine is shown:

Which can be further decomposed into a combination of two blocks:

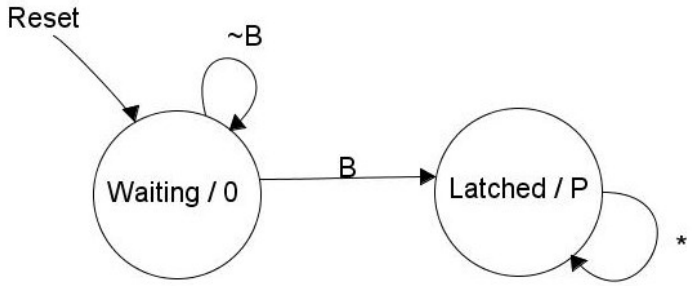


The responsibility of the input controller is to make sure that the FSM gets valid and clean input. Then the Traffic Light FSM takes care of changing the overall state of the system and producing the correct output (combination of lights). Hence the Input Controller acts like a filter.

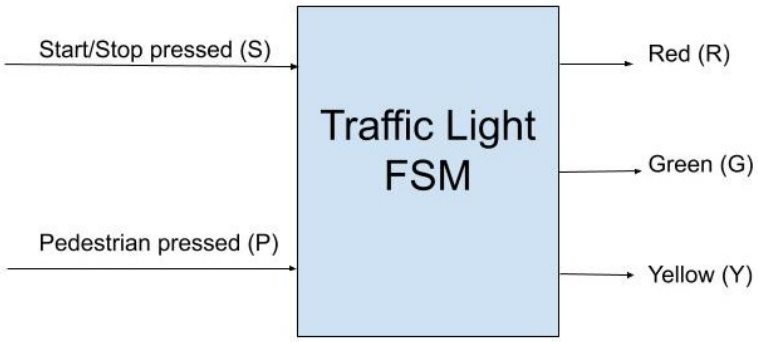
The input controller looks for rising edges on its input signal *B*. When a rising edge is detected, this value is latched until the next cycle. For the input controller a cycle is abstract, but it must be defined by the implementation (the C code). In this implementation, a cycle for the FSM will consists of several iterations through a *while* loop, which will be kept track of with a variable called *ticks*. When *ticks* exceeds an arbitrarily chosen number, the system moves to the next state and the input controller is reset and can start looking for the next rising edge.

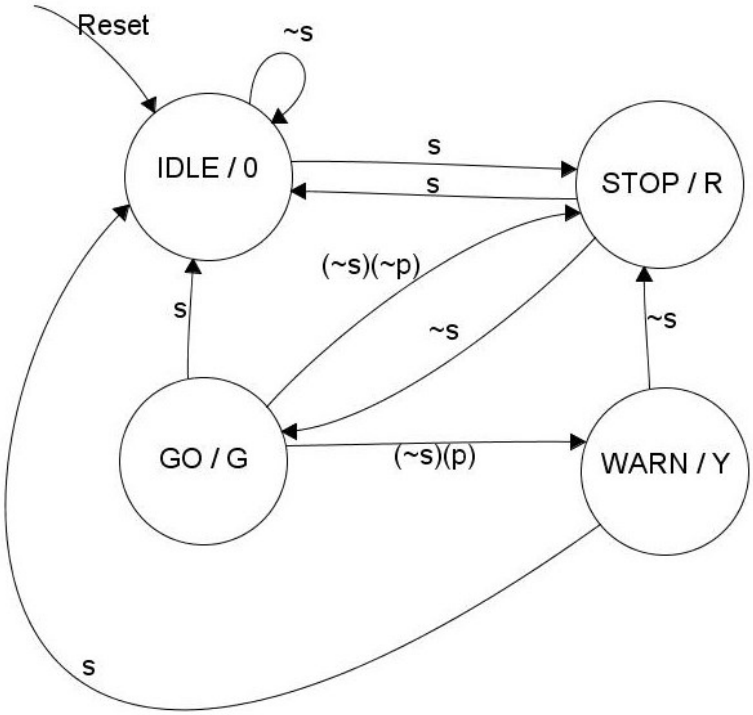


Conceptually, the input controller is a simple finite state machine:



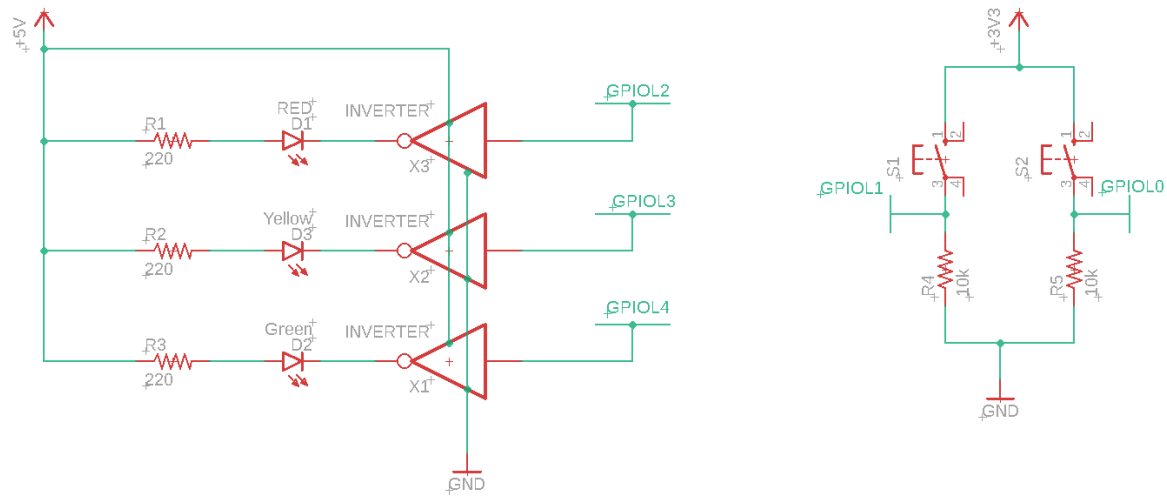
Next is the Traffic Light FSM:





- When in the IDLE state, the machine is waiting for the start/stop button to be pressed and go to the STOP state. I selected the STOP state for safety. If it started in any other state, I would not expect drivers nor pedestrians to be ready.
- In the STOP, GO, and WARN state if the start/stop button is pressed, the machine will go to the IDLE state (the system is disabled). Otherwise, the following applies:
- In the STOP state, the machine is waiting to transition to the GO state.
- In the GO state, either it goes back to the STOP state if the pedestrian button is not pressed. Otherwise it will go to the WARN state if the pedestrian presses their button.
- In the WARN state, the machine is waiting to move to the STOP state.

Having the Input Controller and the Traffic Light FSM designed, it became a matter of implementing the machine. First, the following circuit is built:



The C code is a matter of implementing a Finite State Machine in software:

- A *ticks* variable is kept track of to decide when to move to the next state. It was found experimentally that 100 thousand loops was a reasonable value for this laboratory. When *ticks* reaches this value, it gets reset to zero.
- The input controller keeps track of the previous and new value of the buttons to determine there has been a rising edge. This allows the system to register a button press as a single press (the user must release the button and press again for the system to count a second press). Hence the system is guarded against debouncing issues.
- The next state and present stare are stored in variables. Switch statements on the present state variable combined with logic on the buttons allow the system to decide the value for next state. The next state is adopted when *ticks* reaches its maximum value.
- Since this is a Moore-type Finite State Machine, the output only depends on the present state. Therefore, a switch statement on the present state variable allows the system to produce the correct output.
- When *ticks* reaches its maximum value, the input counter is reset so that it can look for the next rising edge.

Section 2: Results

Task 1

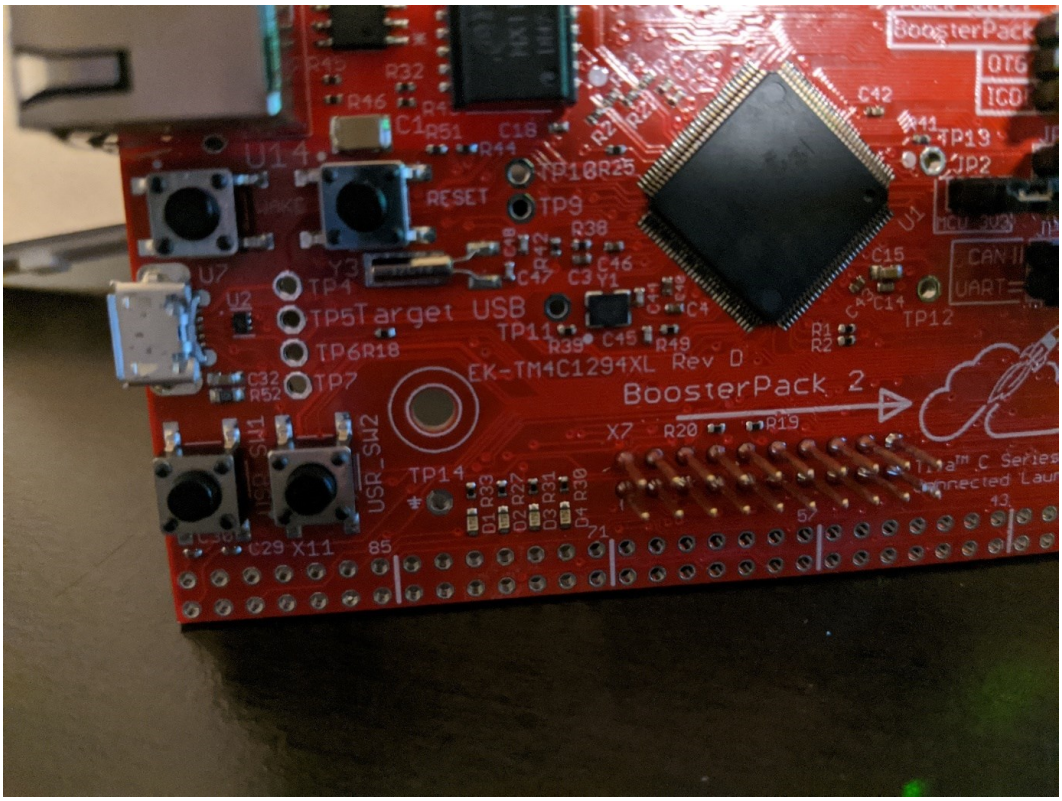
Part 1 – Making a header file

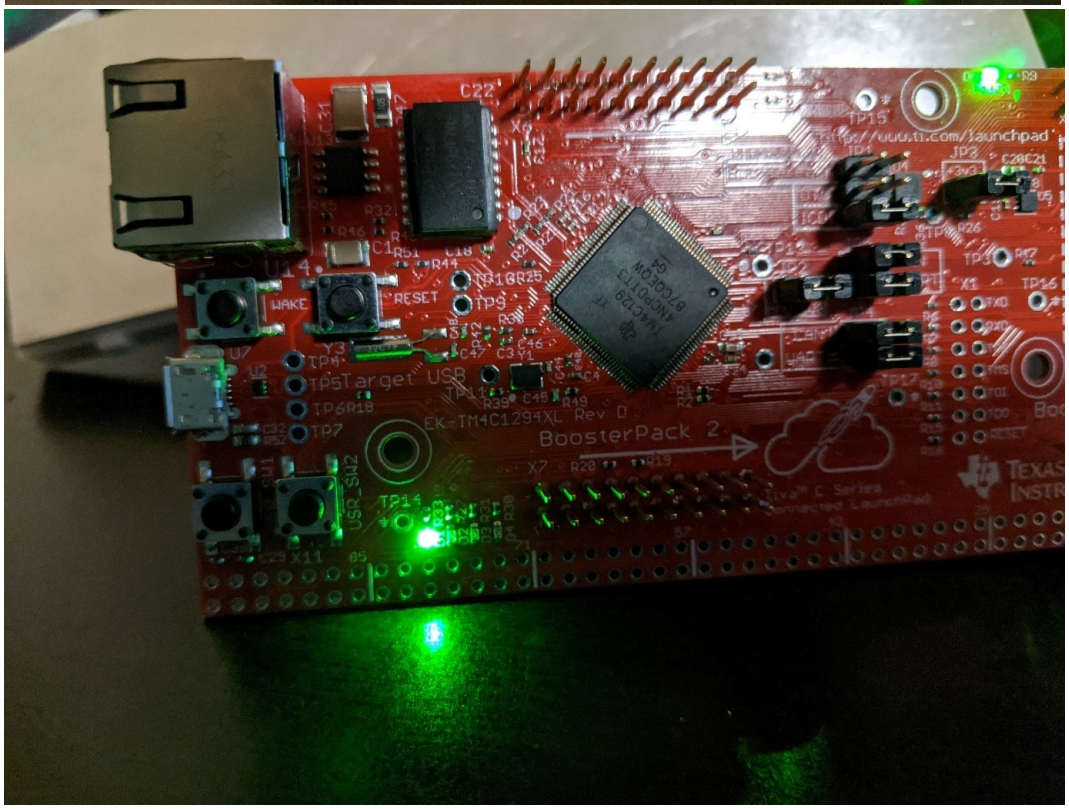
The header file *tm4c1294ncpdt.h* has been found to be a reliable tool for the rest of task 1 and for task 2. Including this header file has allowed for more readability and less code repetition. The fact that tasks that depended on this header file were successful also increase confidence that the header file is working correctly.

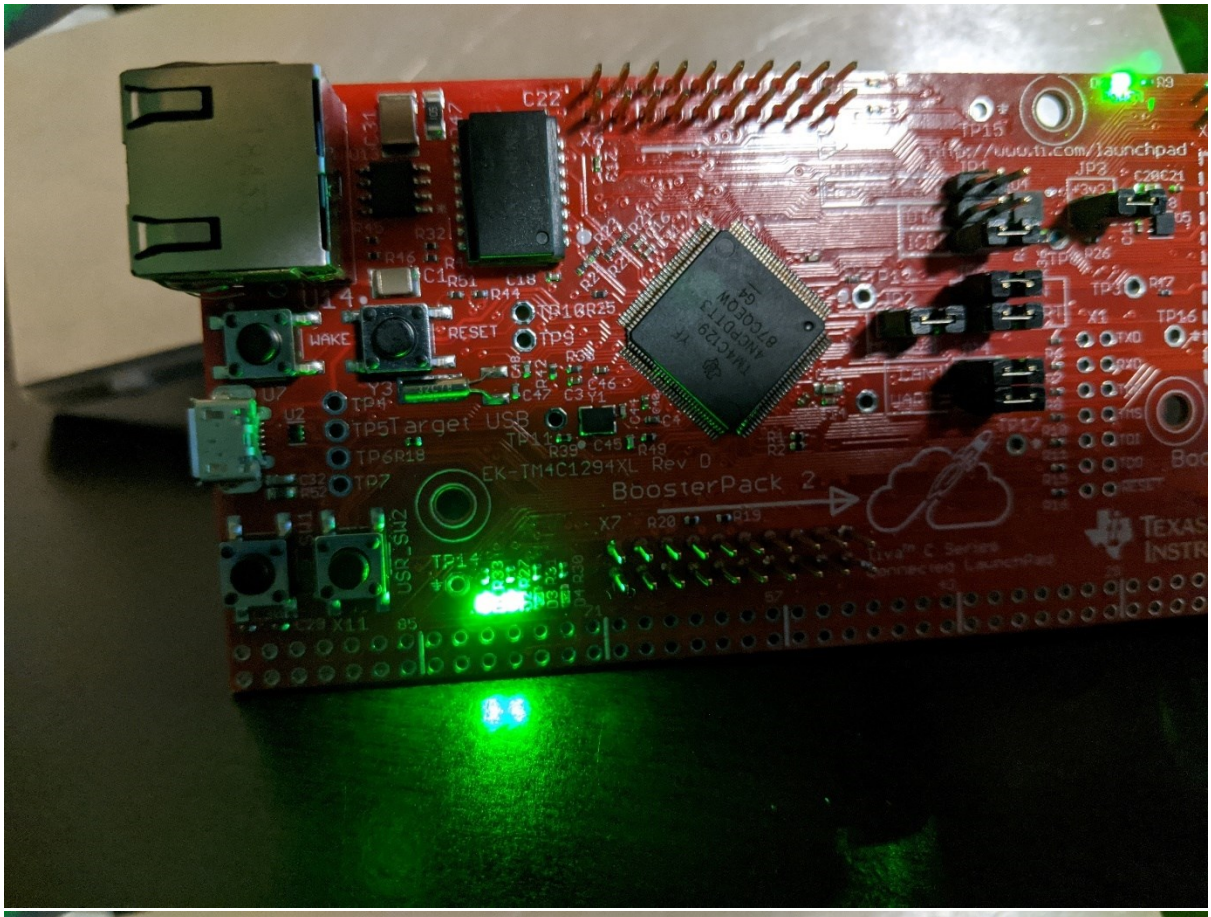
Part 2 – Modifying the provided program to turn on and off the LEDs in a periodic pattern.

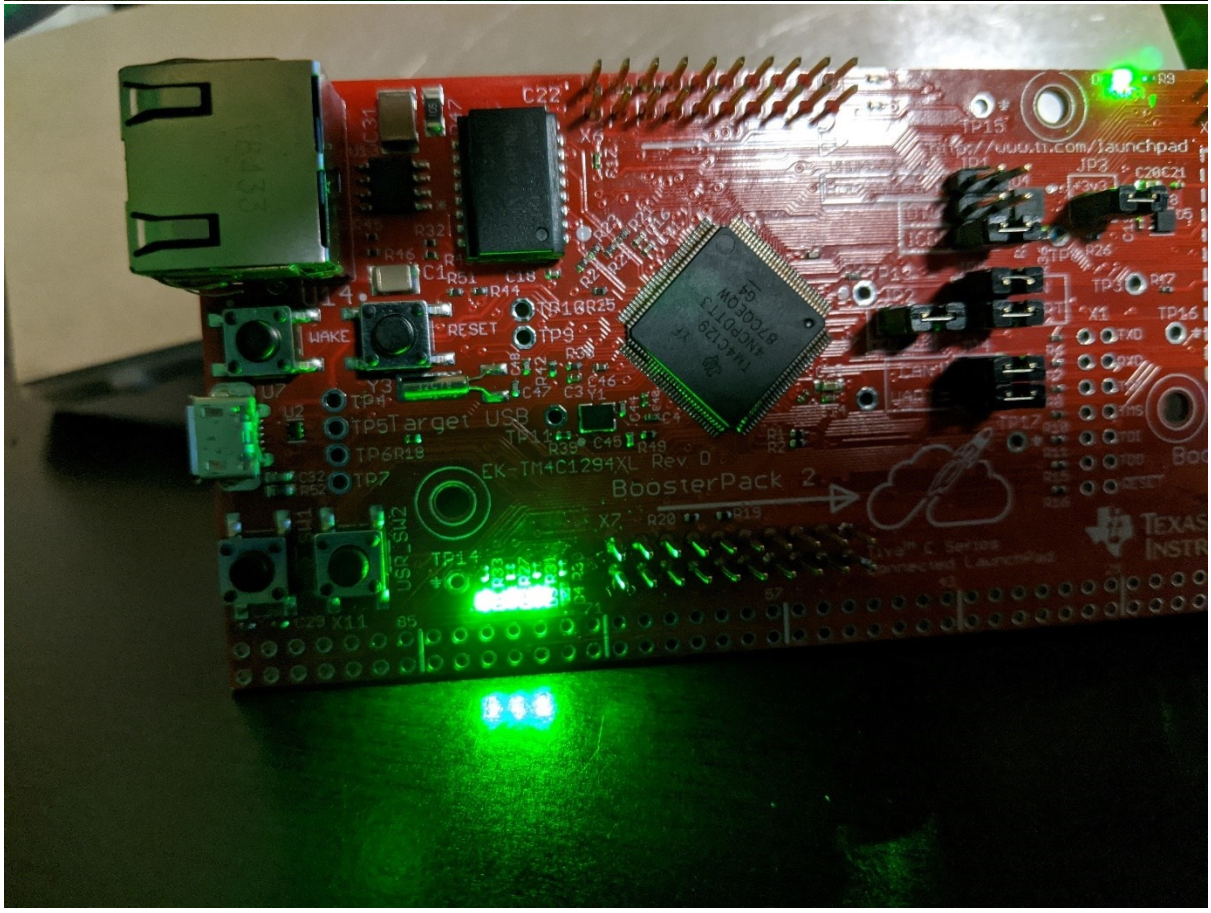
The system meets the specifications derived from its design. The LEDs turn on in order of increasing LED number (D_1, D_2, D_3, D_4) with some intended delay in-between. Then

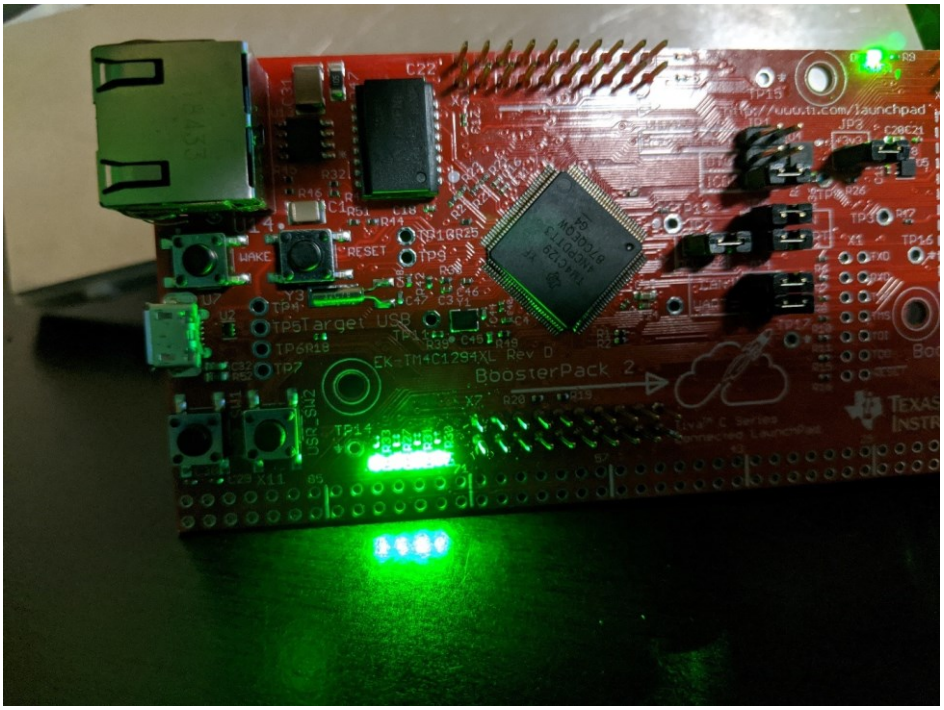
the LEDs proceed to be turned off in the opposite order (D_4, D_3, D_2, D_1). This cycle occurs periodically and indefinitely while the system is powered. The following images show the visual output of the system:





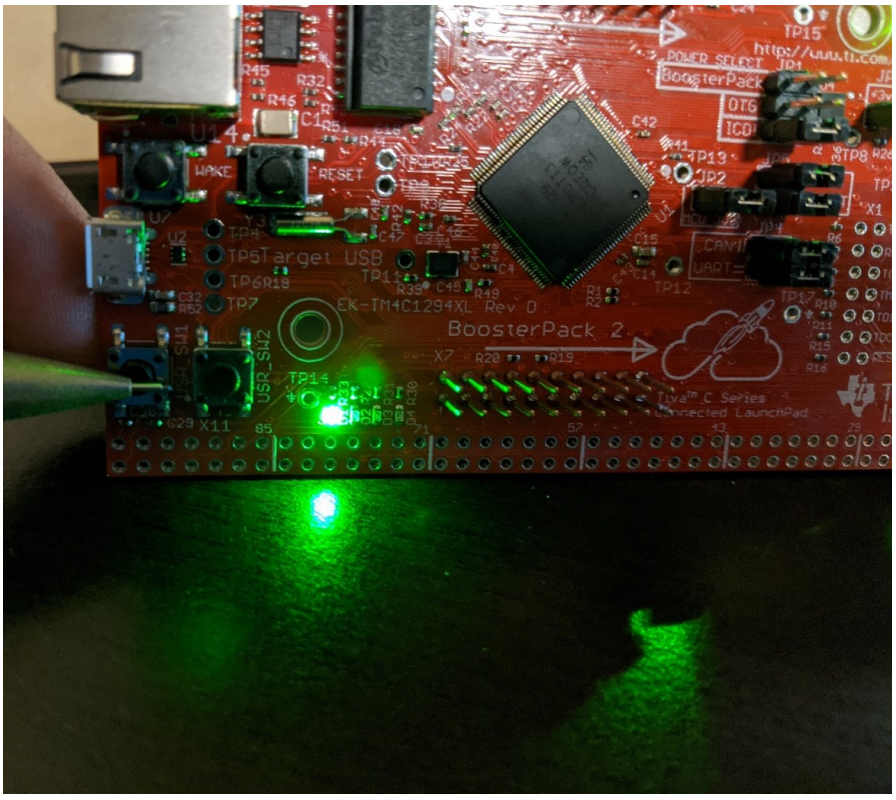

WAKE
RESET
Target USB
EK-TM4C1294XL Rev D
BoosterPack 2
USR_SW2

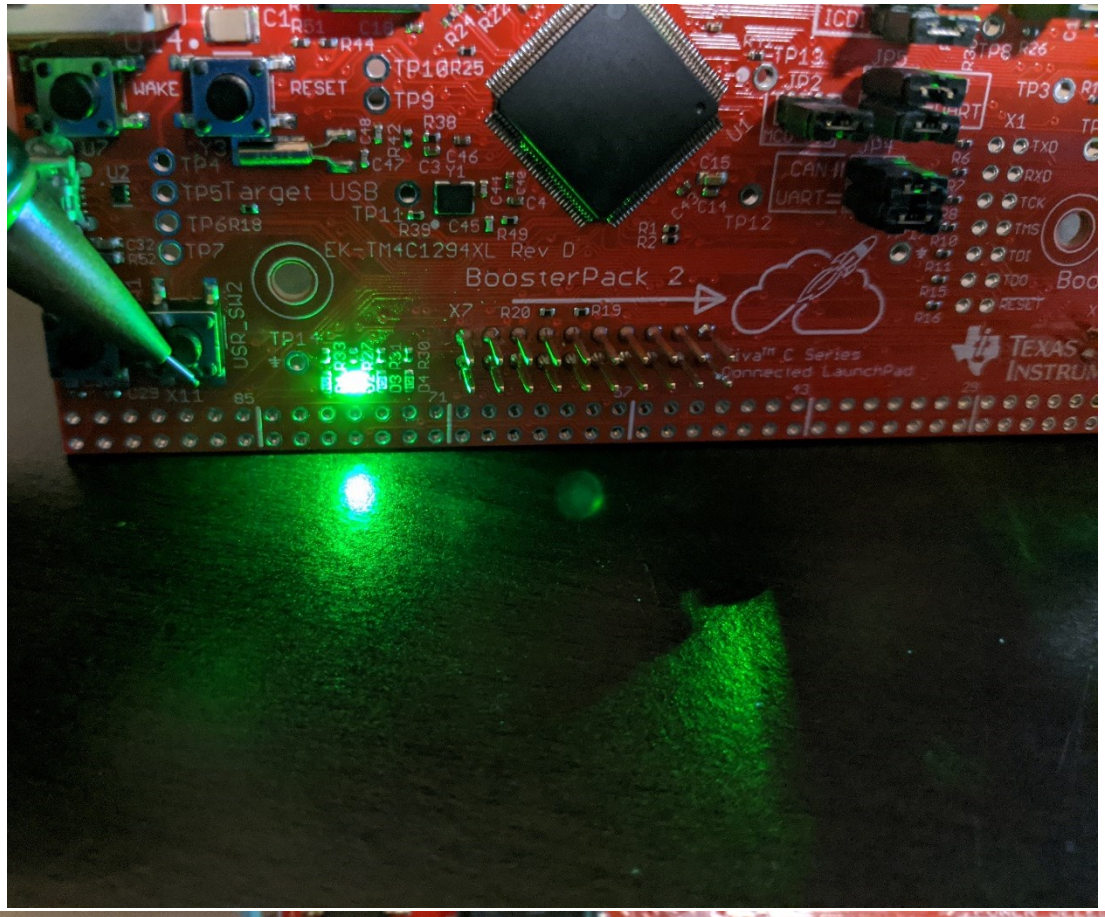

WAKE
RESET
Target USB
EK-TM4C1294XL Rev D
BoosterPack 2
USR_SW2

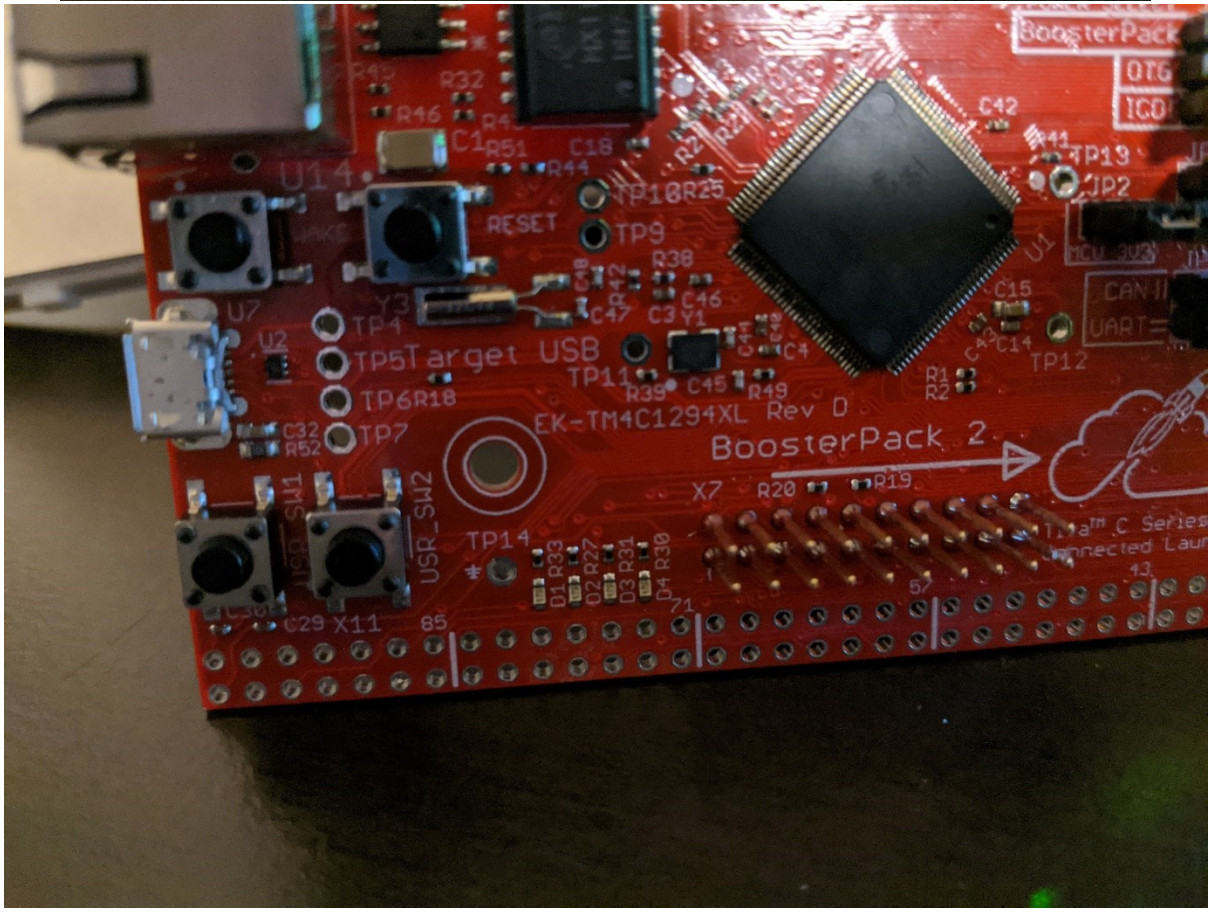


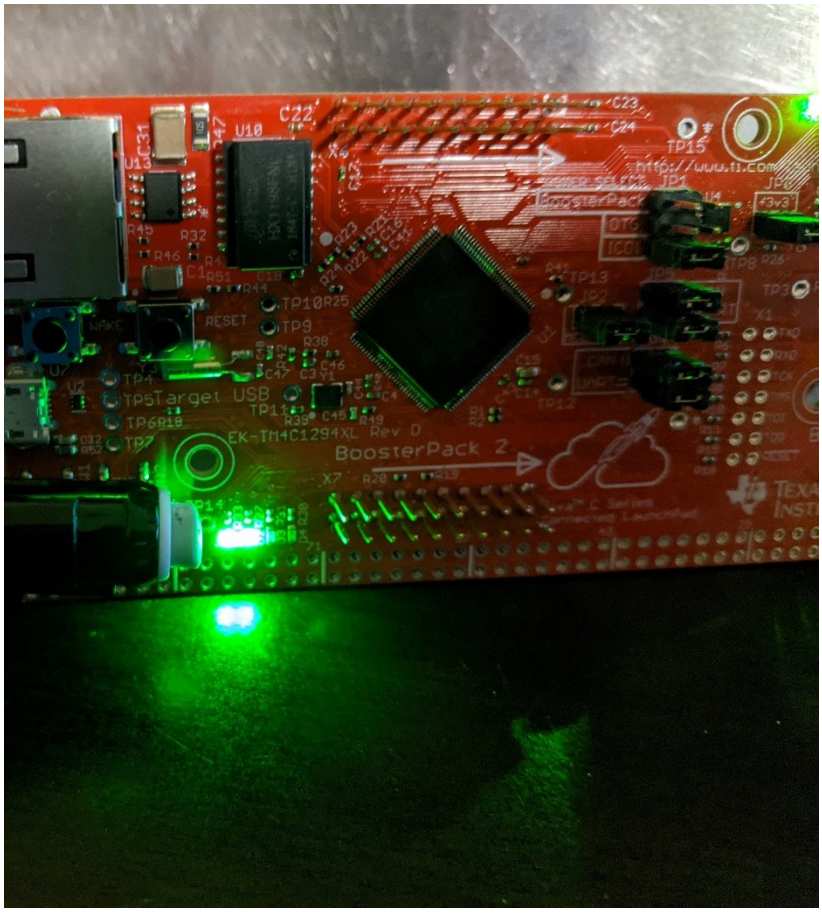
Part 3 – Controlling the LEDs using the user switches.

The system meets the specifications. LED D1 is ON if and only if the switch SW1 is being pressed. Similarly, LED D2 is ON if and only if the switch SW2 is being pressed. The following images show the machine during operation:




WAKE
RESET
TP10
R25
TP9
R38
TP4
TP5
Target USB
TP11
TP6
R18
TP7
EK-TM4C1294XL Rev D
BoosterPack 2
USR_SW2
TP14
X7
R20
R19
TP12
TP13
JP2
TP3
X1
TXD
RXD
TCK
TMS
TDI
TDO
RESET
CAN II
UART
ICDI
TEXAS INSTRUMENTS


BoosterPack
OTG
ICDI
U14
RESET
TP10
R25
TP9
R38
TP4
TP5
Target USB
TP11
TP6
R18
TP7
EK-TM4C1294XL Rev D
BoosterPack 2
USR_SW1
USR_SW2
TP14
X7
R20
R19
TP19
JP2
TP12
CAN II
UART



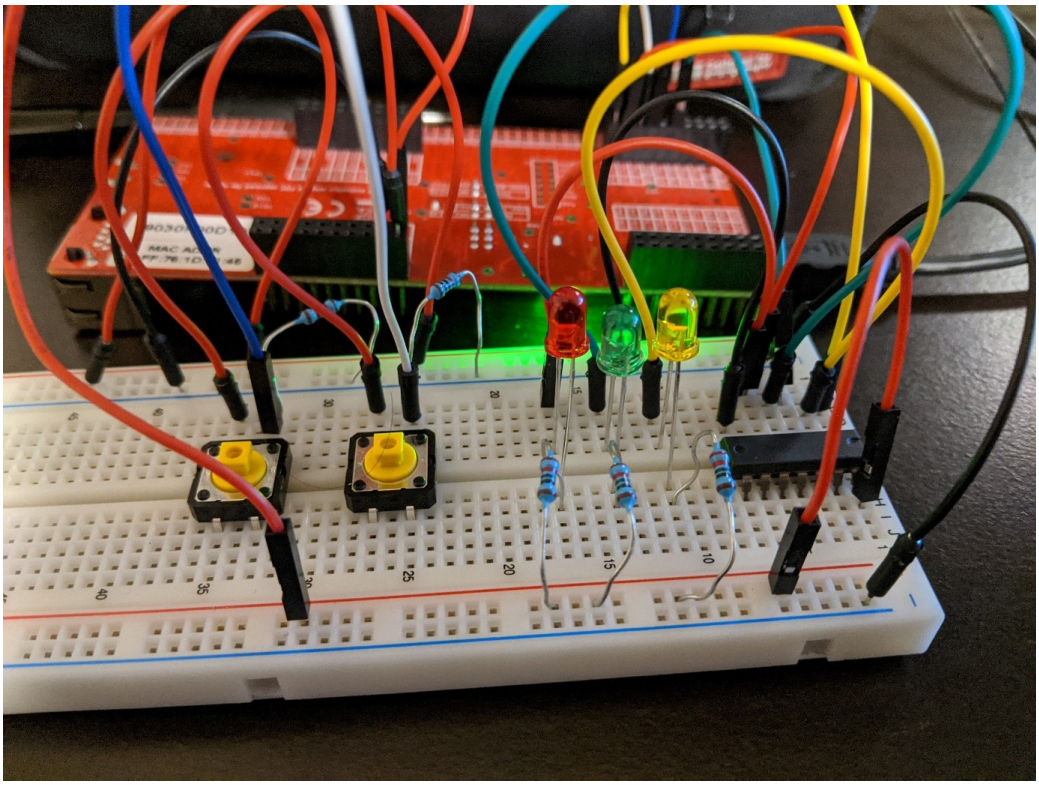
Task 2

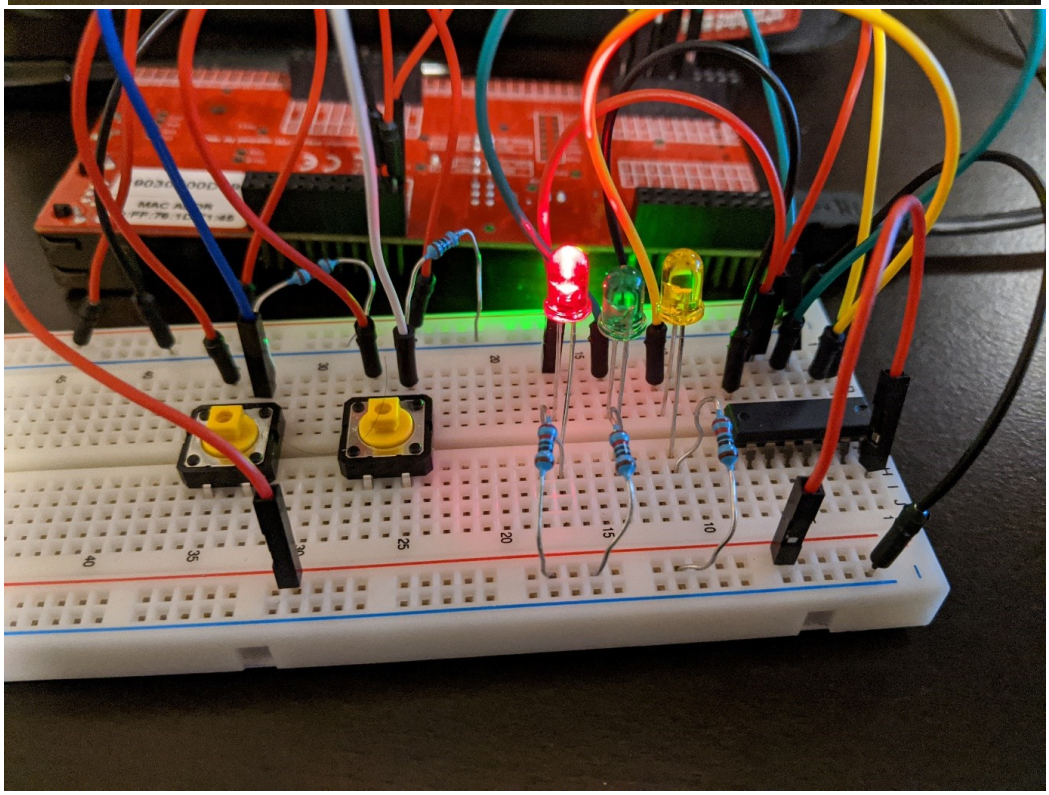
The system simulates a traffic light controller that meets the specifications in the original laboratory document using external peripherals (LEDs and Pushbuttons).

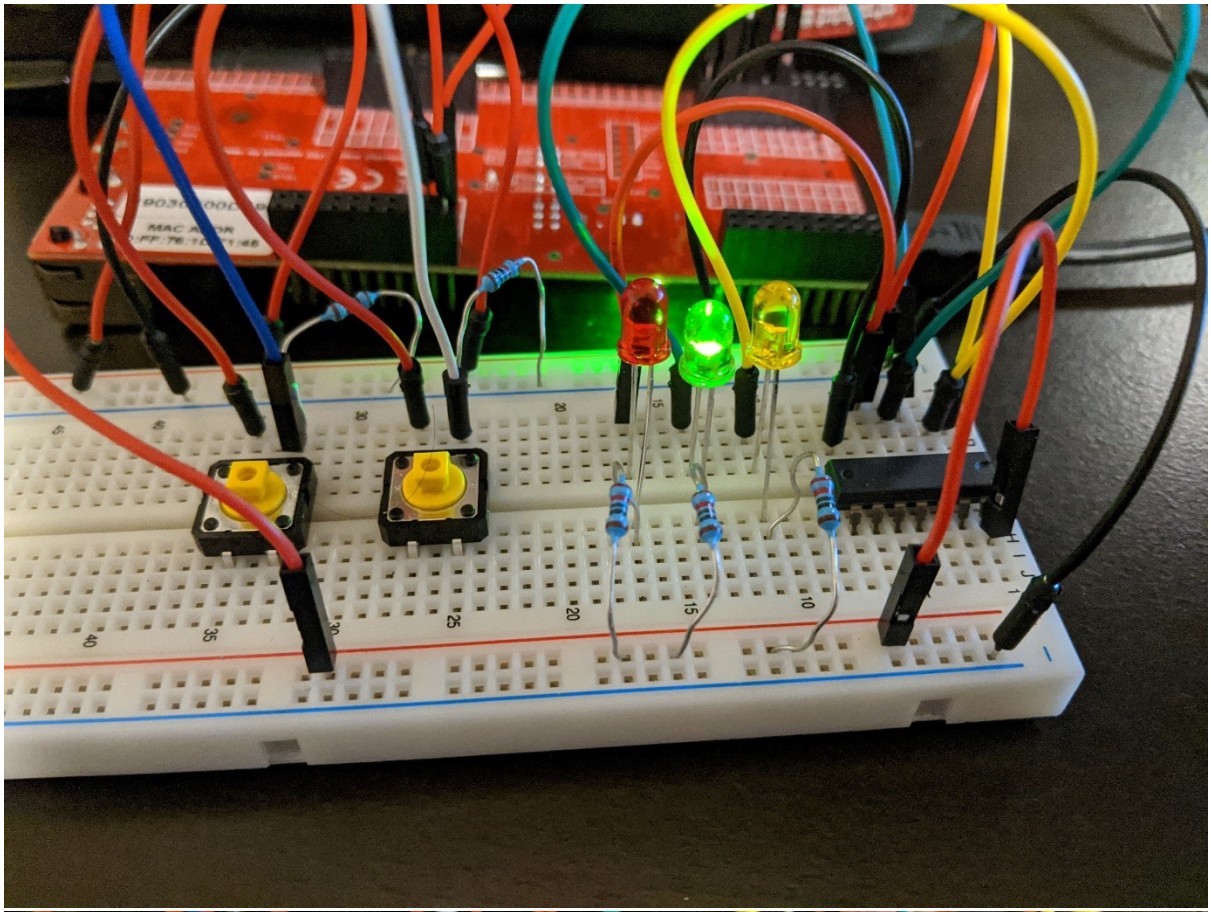
- The start/stop button will start the system when the system is off, and it will turn the system off when the system is running. The system will always start in the STOP state for safety.
- Pressing the pedestrian button during the STOP or WARN state or when the system is turned off or being turned off has no effect. Otherwise the pedestrian button signals the system to go through the WARN state if pressed during a green light (GO state).
- The only way to leave the IDLE state is to press the start/stop button. Entering IDLE can be done by pressing the start/stop button while the system is running or by a reset.
- Pressing the start/stop button outside of the IDLE state will always stop the system and bring it to IDLE.
- The WARN state is always followed by the STOP state unless the system is being turned off (brought to IDLE).
- If neither button is pressed and the system is running, it will alternate between STOP and GO.
- The system correctly turns on a red LED if and only if the state is the STOP state, a green LED if and only if the state is the GO state, and a yellow LED if and only if the state is the WARN state.

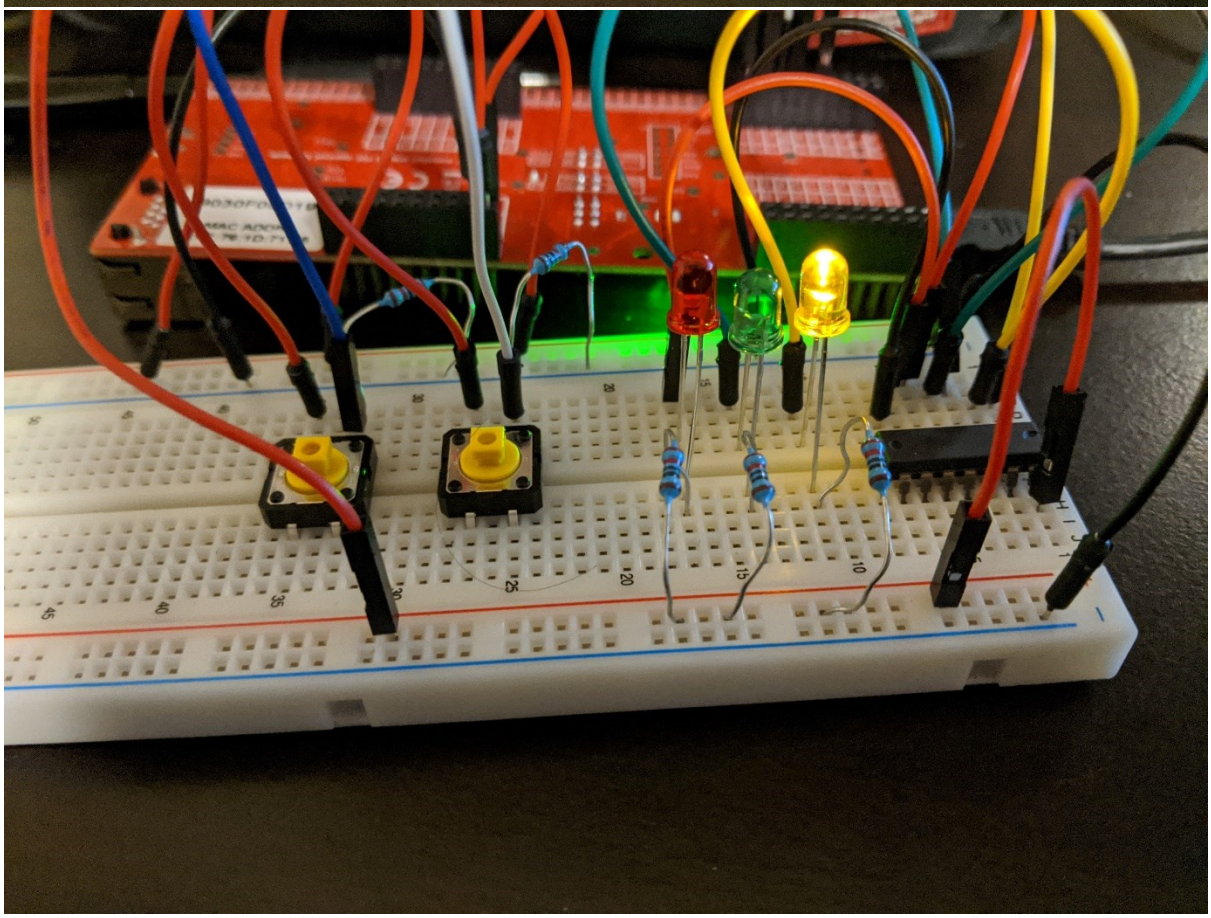
- The system is able to handle all inputs and deterministically decide the next state.
- The system guards itself against debouncing issues by only reacting to rising edges.

The following images show the system in action:









Section 3: Problems Faced & Feedback

I believe the biggest challenge of the present laboratory are:

1. Protecting the system from glitchy input or debouncing issues.
2. Timing and state transitions.

Protecting the system from debouncing issues or glitchy input is necessary since our machine will run through the loop of our machine way too fast. One problem is that if you choose to slow down the system using delays then your buttons will be unresponsive, and the system will fail to capture the input. Another problem is that it should not matter if the button is pressed at the beginning or the end of a state as long as the input arrives before the next transition.

The resolution to this problem was to make the system detect rising edges and count that as an input instead of simply looking for high/low signals. The idea is that if you capture a rising edge then the input counts as a single button press. If the system only looked at the value of the signal, then a button press that occurs right at the transition between two states would be counted twice and it would count as an input for both states. In contrast, looking at the rising edge and capturing the input until the next transition means that the system will count the input as a single press and it will be counted as an input to the state in which it arrived, effectively solving the problem.

The second problem of timing has to do with the fact that our machine will be running very fast and it would be impossible for us to see the changes or to provide proper input. The resolution was to only update the state transitions once a certain number of iterations through the loop have occurred to simulate a timer (we will use real timers in future laboratories). The number of iterations was obtained experimentally to be one hundred thousand iterations per state. It is also possible to customize the length of each light by adding a switch statement on the present state based on government regulations.

The feedback for the present laboratory is positive. I believe the assignment allows every student to get accustomed to reading datasheets, and interfacing with the tm4c1294ncpdt board and its GPIO peripheral to drive small digital signals (such as LEDs) and to read simple digital signals (such as buttons).