EE 474 Lab 5: Human Computer Interface - Muscle Sensor

Students: Rikuo (Ricky) Sato
Devyansh Gupta

Introduction:

For our final project we wanted to use a sensor that would give us an analog input so that we could get some practice taking real world data and converting it into digital computations. To do this we found the myoware sensor (pictured right) which can read the electrical signals muscles create when they flex. The sensor rectifies and amplifies this small voltage so that it can be easily read through an ADC. On the digital side we wanted to do something cool yet simple that the sensor's output would control. Originally we thought we could use the sensor to play a game of pong, but would later choose the simpler t-rex game that is hidden in google chrome (pictured below).

T-Rex, Run!

A hidden Google dinosaur game(T-rex game) now can be played online for free!
Press **Space** key or **Tap** the T-rex to jump.

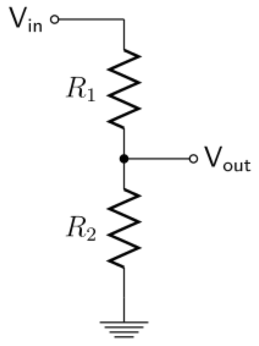
Procedure:

We broke this task into two parts, one for each of us. While Ricky worked on understanding the myoware sensor and getting its output to be read by the board's ADC, Devyansh worked on interpreting the data sent into the computer through the UART so that it could control the computer.

Ricky: Myoware Input into the ADC and UART for data communication

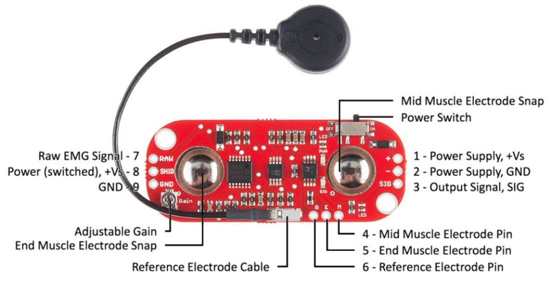
For my section I took a step by step approach. I wanted to make sure that I could reliably measure an analog signal through the board and display the value on the computer's PUTTY terminal. After which I wanted to increase the sampling rate of the ADC as much as possible so that I could get the most accurate input before finally moving on to tackle the myoware sensor itself. This way I would be able to reliably test the myoware sensor without worrying about the rest of the program being inaccurate.

Thankfully I kept every single bit of my code so I did not have to worry about the UART's communication with the computer. I did test this of course to be sure it worked before moving on to configuring the ADC to take in a voltage from a GPIO pin. To test this part of the program I used an oscilloscope and a simple voltage divider where the output voltage was controlled by a potentiometer. To safely test this with the tiva board I used the 3.3 V as the supply for this voltage divider so that there was no chance of destroying a GPIO pin. (In the voltage divider to the right V_{in} = 3.3 and R_2 was a 200 ohm potentiometer while R_1 was only 100 ohms or so) With this I could easily vary the voltage so I could see in real time how well the ADC could read the changing voltage. I could also test how accurate the program was by comparing the program's output to the oscilloscope reading.



After confirming that my program could interpret different voltages accurately I saw that it's sampling was sluggish so I sped up the program by keeping the trigger for the ADC as the timer but loading the timer with 1 instead of 16M. I also overclocked the board to 80MHz and set the maximum sample rate of the ADC to its highest value. Through this I managed to get the ADC to have a very high sample rate. I then tested the program using a function generator to see if it could accurately track a rapidly changing wave form. From my tests I confirmed that it could sample 8 times along a sine wave before it repeated. Unfortunately I did not record the frequency I tested so I can not calculate the ADC sampling rate but that is how I would approximate it.

With all of this done it was time to figure out the myoware sensor. This took me a spectacularly long time but in essence the sensor amplifies the signal using whatever voltage supplied as the V_{DD}. It also does some fancy rectifying and filtering which allows us to get a very easy to use output from the SIG pin. The other pins are not useful to us in this project. When the muscle the sensor is attached to is flexed the sensor outputs a higher voltage - never going over the supply. When the muscle is relaxed the output drops to near zero.


Mid Muscle Electrode Snap
Power Switch
Raw EMG Signal - 7
Power (switched), +Vs - 8
GND - 9
1 - Power Supply, +Vs
2 - Power Supply, GND
3 - Output Signal, SIG
Adjustable Gain
End Muscle Electrode Snap
Reference Electrode Cable
4 - Mid Muscle Electrode Pin
5 - End Muscle Electrode Pin
6 - Reference Electrode Pin

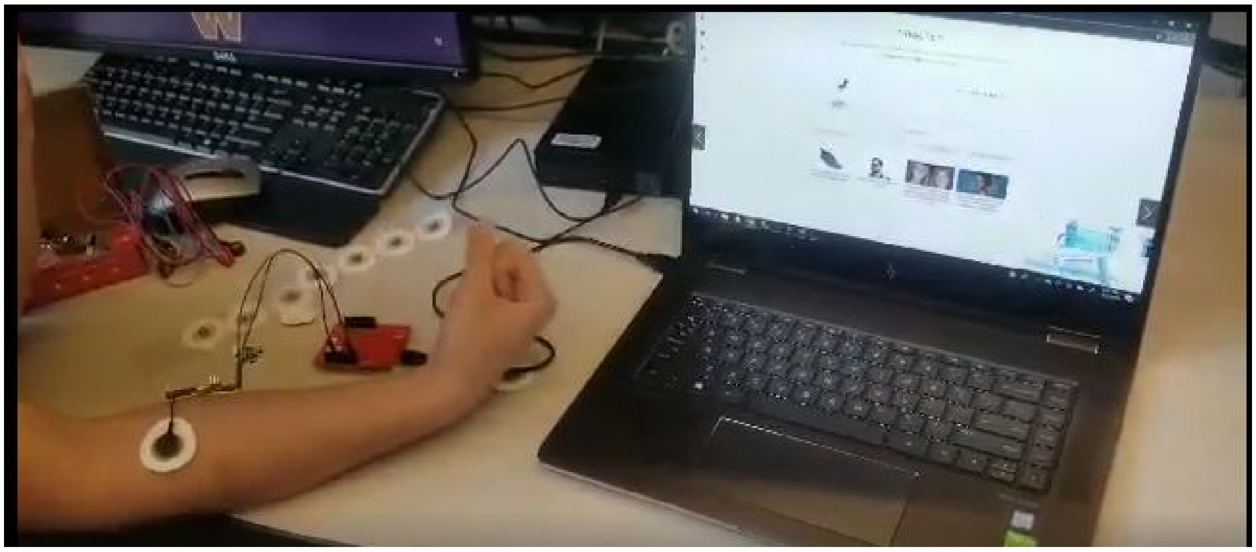
The myoware sensor uses three electrodes. Two of which must be placed linearly over the muscle while the third reference electrode is placed away from the measured muscle. There are various shield attachments that can be added but we only used the main board. Through my experimentation and research I eventually realized that to be used properly the female-male breakaway pins must be soldered into place - a step Devyansh performed. After that it was simple to plug in 3.3V from the board as a power supply (so that the maximum output voltage would be safe for the GPIO pins), after which we simply connected the SIG to the analog input pin I specified in the program.

Devyansh: Processing the data from UART and writing the Python script
Devyansh was provided with data coming in from the sensor to PuTTY and then process it to play the game. To perform these tasks he first had to learn how to read in the serial data in python from the UART and by googling he found out about the library **PySerial** which could be used to read data from the UART. One interesting thing that he learnt while doing so was that if the data was already flowing in through PuTTY then python won't be able to communicate with the sensor as the protocol was UART and could only communicate with one program at a time. To combat this issue, he had to load the complete program to the Tiva board via IAR workbench and then start reading the data using python instead of PuTTY. After reading the data in python successfully, he set up a threshold value after testing the sensor. Whenever this threshold was crossed (depending on muscle flex) the computer was instructed to press spacebar. To map the keys with the python script he used another library **autogui**.

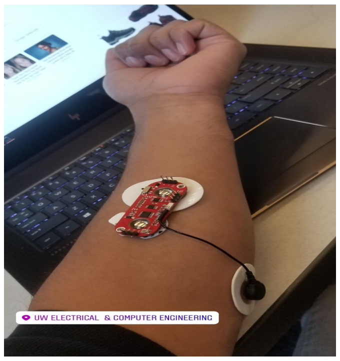
After successfully writing the script, the user just had to start the script once before playing the game and have endless hours of fun.

Results:

After we were done with our separate parts we could take analog inputs from the myoware sensor and use them to control the cursor and keyboard through the UART connection. To demonstrate this we chose a game that only required one input (the space key). We created the program in such a way that each time the myoware sensor sensed the muscle tense the program would press the computer's space key and would not do this again until the muscle was relaxed. We chose the dinosaur game as it only required the space key to play and reset. Below is a picture of our project in action. In particular you should be able to see the dinosaur jumping over a set of trees after the arm is flexed.



Below is a better picture of how to set up the myoware sensor's electrodes. Notice that the reference electrode is placed far away and perpendicular to the targeted muscle and that the two electrodes directly on the board are placed linearly along the muscle.



Short User Guide:

1) Connect the Myoware sensor's electrodes to the desired muscle by placing the two board electrodes linearly along the muscle and the reference far away.
2) Download the program onto the Tiva Board.
3) Connect the 3.3V output of the board to the positive voltage supply pin and the ground pin to the negative. Then connect the SIG pin to pb4.
4) Open the correct PUTTY terminal and a program that can run the python script. You should see the voltage output from the myoware sensor on the PUTTY terminal.
Speed: 9600, Data bits: 8, Stop bits: 1, Parity: None, Flow control: XON/XOFF
5) After opening up the game (which can be found here https://elgoog.im/t-rex/) run the python script, select the game's window, and you should be all set to go!
6) To play the game you flex the muscle to start the game, make the dinosaur jump, and restart the game once you lose. The goal is to dodge all of the obstacles as the dinosaur runs to the right and you lose once the dinosaur hits any obstacle.

Problems Encountered/Feedback:

Myoware Sensor's Pins:

Since the myoware sensor came with breakaway pins instead of soldered in connections, and neither of us had experience with breakaway pins and assumed they could be gently pushed into place, we initially struggled with making the myoware sensor work as any shift in the pins would stop the sensor and cause the program to crash. Ricky managed to successfully sample from the sensor by carefully aligning the pins and then holding them in place as he ran the program, but we quickly came to the conclusion that to use the sensor we must solder in the pins. Which we did. Also, at the time of soldering, Devyansh soldered the pins in the wrong direction the first time as it was his first time dealing with break away pins. He later had to unsolder all of those pins and then re-solder them in the correct way.

Initial Testing of ADC Analog Input:
As discussed previously we needed a way to test if we were accurately reading an analog input through the ADC. This meant that we needed to create a voltage that we could vary and have some way to accurately measure the voltage so that we could record the actual voltage and compare it to what we got from the ADC. We also needed to be confident that the voltage we were using would never rise above the allowed voltage of the GPIO pins. To do this we set up a voltage divider with a supply of 3.3V and with a potentiometer as the load resistance. (See above in Ricky's procedure for further circuit details) Since the supply was set to 3.3V the output voltage could never be larger than 3.3V which is a safe voltage for the GPIO pins. To get accurate measurements we used an oscilloscope to get accurate measurements in real time so we could compare the values computed by the ADC which were displayed on the PuTTY. The potentiometer allowed us to easily change the output voltage to test the response time of the program, accuracy, and if there were any limits on the voltage.

Slow Processing in Python Script:
The python script was written by Devyansh. It used the library autogui to map the keys to the input received from the TIVA board via UART. Surprisingly, whenever we would flex the muscle the UART would start reading the input at a considerably slower speed and then would stay above the threshold for a while even when we relaxed the muscle. Initially, we thought this was the nature of how muscles work but later after debugging the code and looking on the internet we found out that the autogui library had a PAUSE function which would put a delay between consecutive commands and therefore the input was getting slowed down and on relaxing the values in buffer were coming over for a while. After fixing that error, we started getting smooth flow of inputs.
We used the code: pyautogui.PAUSE = 0.01

Overly Excited Dinosaur:
The python script was written by Devyansh. The issue we faced was that the dinosaur would jump for multiple times even though we only flexed once. This was because till the time the muscle was unflexed the computer would keep on pressing the spacebar. We combated this issue

by placing a flag to make sure that the jump command was only sent once and the command wasn’t sent again until the muscle was flexed at least one time.

Conclusion:

Working on this project we brainstormed several real life applications that we could develop our final product into. In essence, our project was to create a way to control computer functions using only a muscle. This obviously would help anyone with disabilities that would prevent them from properly operating a keyboard and mouse. In theory you could create a simple binary input by only using two myoware sensors. And as we know, binary is all you need to create more complex systems. If we were to pursue this sort of application then we would next need to either make or find a user interface that can be controlled with only binary inputs. We would also want to keep most functionalities limited to 2 or 3 inputs from the user. Beyond the disabled this could also be useful for hospitalized patients or post-surgery patients who cannot move freely. Nurses can be summoned without reaching for a button, or a digital book’s pages could be flipped by a simple flex of the arm.

Outside of helping the disabled we also saw applications in general safety. For example, if we could create a smaller and more accurate muscle sensor then it would be feasible to track a user’s eye movements by tracking the muscles around the face. Tracking the eyes also has applications for helping the disabled but in this case we could use the eye tracking to monitor a driver’s alertness. Though actually tracking eye movement precisely would be difficult, it would not be hard to detect when the eyelid is closed for an extended period of time. While current safety features will alert the driver when they start crossing a line, this safety feature could prevent drivers from falling asleep at the wheel or deploy safety features if the driver does fall asleep.

Appendix:

main.c

```c
#include "lab.h"
#include <stdint.h>

double temp;
int x = 0;
int main()
{
  reset_clk();
  switch_control_set();
  onboard_led_start();
  GPIO_b4_analog();
  fast_timer_start();
  onboard_switches_interrupt_enable();

  UART_Init();

  ADC_Analog_Init();
  ADC_interrupt_enable();


   while(1)
   {
     led_data = 0;
   }
   return 1;
}


//Triggers every cycle according to a timer
void ADC_SS3_Handler(void)
{
  temp = (ADC_FIFO3 * (3.3 / 40960)); //Converts value from ADC into voltage

  //Determines the value to put into the led_data register
  UART_Print(temp);

  ADC_interrupt_clear();

}

void Port_F_Handler(void)
{

  //Switches the system clock and the timer's load according to which switch was
pressed
```

```
    if(switch1 == 0)
    {
      timer_off();
      reset_clk();
      UART_Print(16);
      fast_timer_start();

      delay
      port_f_clear_interrupt();
    }else if(switch2 == 0)
    {
      timer_off();
      set_clk_80MHz();
      UART_Print(80);
      fast_timer_start();

      delay
      port_f_clear_interrupt();
    }

  }
```

lab.h

```
#ifndef lab
#define lab


#include <stdint.h>

//Task 1: Onboard LED and Switches
#define clock_gate_control (*((volatile uint32_t *)0x400FE108)) //Clock gate control register for GPIO ports: 0 to Port A, 1 to Port B... 6-31 are irrrelevant (pg 464)
#define led_dir (*((volatile uint32_t *) 0x40025400)) //(Port F GPIO) Sets the different directions of the GPIO that control the onboard LED. We'll want bits 1 to 3 set which will make them outputs (pg 663)
#define led_digital (*((volatile uint32_t *) 0x4002551C)) //(Port F)Let's us set the GPIO to digital vs analog. Want all digital which means we want the corresponding bits set (pg 682)
#define led_data (*((volatile uint32_t *) 0x400253FC)) //(Port F)Controls the actual values being sent through the GPIO. If we were reading it is where the data would come in. Address acts as a mask. This address actively changes and reads all pins. Bit #s correspond to pins (pg 662)
#define portf_unlock (*((volatile uint32_t *) 0x40025520)) //(Port F)Unlocks write access to port F's GPIOCR. Code to unlock is 0x4C4F.434B (pg 684)
#define portf_commit (*((volatile uint32_t *) 0x40025524)) //(Port F) Allows GPIOPUR to be changed. Without unlocking the corresponding bits nothing written to GPIOPUR sticks. (pg 685)
#define portf_pull_up (*((volatile uint32_t *) 0x40025510)) //(Port F) Allows assignment of pull up resistors to GPIO pins. Necessary for onboard switches. Bit #s correspond to pins (pg 677)

#define change_blue (*((volatile uint32_t *) 0x40025010)) //Allows turning on the blue led without changing the state of the other Port F pins by utilizing the address mask (pg 662)
#define change_green (*((volatile uint32_t *) 0x40025020)) //Allows turning on the green led without changing the state of the other Port F pins by utilizing the address mask (pg 662)
#define change_red (*((volatile uint32_t *) 0x40025008)) //Allows turning on the red led without changing the state of the other Port F pins by utilizing the address mask (pg 662)
#define switch1 (*((volatile uint32_t *) 0x40025040)) //Allows reading of switch 1 without changing the state of the other Port F pins by utilizing the address mask (pg 662)
#define switch2 (*((volatile uint32_t *) 0x40025004)) //Allows reading of switch 2 without changing the state of the other Port F pins by utilizing the address mask (pg 662)

//Task 2: Traffic Light
```

```
#define clock_gate_control2 (*((volatile uint32_t *)0x400FE608)) //Another clock gate register which allows us to connect the clock to the diferrent GPIO ports (pg 340)
#define port_b_control (*((volatile uint32_t *) 0x4000552C)) //Controls the peripheral functions of port b. Nibble #s correspond to MUX selection for each pin. Clear to have pin act as normal GPIO (pg 688)
#define port_b_analogue (*((volatile uint32_t *) 0x40005528)) //Controls the analogue function of port b GPIO pins. Bit #s correspond to pin. Clear to disable analogue functions (pg 687)
#define port_b_direction (*((volatile uint32_t *) 0x40005400)) //Controls whether port b GPIO pins are inputs or outbuts. Bit #s correspond to pin. Clear for input. Set for output. (pg 663)
#define port_b_alternate (*((volatile uint32_t *) 0x40005420)) //Controls alternate functions of port b GPIO pins. Bit #s correspond to pin. Set to make a peripheral signal. Clear to use as GPIO (pg 671)
#define port_b_digital_en (*((volatile uint32_t *) 0x4000551C)) //Enables pin's digital function. Bit #s correspond to pin. Set to enable digital. Clear to disable. (pg 682)
#define port_b_all_data (*((volatile uint32_t *) 0x400053FC)) //Reading/Writing for all data in port b's GPIO pins. Bit #s correspond to pin. (pg 662)
#define port_b_data0 (*((volatile uint32_t *) 0x40005004)) //Reading/Writing for pin 0 only. Bit #s correspond to pin. (pg 662)
#define port_b_data1 (*((volatile uint32_t *) 0x40005008)) //Reading/Writing for pin 1 only. Bit #s correspond to pin. (pg 662)
#define port_b_data2 (*((volatile uint32_t *) 0x40005010)) //Reading/Writing for pin 2 only. Bit #s correspond to pin. (pg 662)
#define port_b_data3 (*((volatile uint32_t *) 0x40005020)) //Reading/Writing for pin 3 only. Bit #s correspond to pin. (pg 662)
#define port_b_data4 (*((volatile uint32_t *) 0x40005040)) //Reading/Writing for pin 4 only. Bit #s correspond to pin. (pg 662)
#define port_b_data5 (*((volatile uint32_t *) 0x40005080)) //Reading/Writing for pin 5 only. Bit #s correspond to pin. (pg 662)


//setting color values. Bit 1 is red, 2 is blue, 3 is green. (will be plugged into data)
#define RED 0x2
#define BLUE 0x4
#define PURPLE 0x6
#define GREEN 0x8
#define YELLOW 0xA
#define TURQUOISE 0xC
#define WHITE 0xF



//Lab 2 Defines
//Task 1
#define clock_control_time (*((volatile uint32_t *)0x400FE604)) //Clock Gate Controller for the Timer Modules (pg 338)
```

```
  #define timer0_control (*((volatile uint32_t *) 0x4003000C)) //Timer Control: Allows timers to be turned on and off. Can only modify certain timer registers while it is off (pg 737)
  #define timer0A_config (*((volatile uint32_t *) 0x40030000)) //Timer Configuration: Determines what mode the timer is in. Concatenated or individual (pg 727)
  #define timer0A_mode (*((volatile uint32_t *) 0x40030004)) //Timer Mode: Continues to configure the different modes of the timer. We are interested in the timer's direction and type (pg 729)
  #define timer0A_load (*((volatile uint32_t *) 0x40030028)) //Timer Load: Determines how long the timer will be by setting setting the bounds of the counting (pg 756)
  #define timer0_interrupt_status (*((volatile uint32_t *) 0x4003001C)) //Raw Interrupt Status: If timer reaches its threshold the corresponding bit will be set (pg 748)
  #define timer0_clear (*((volatile uint32_t *) 0x40030024)) //Clears the timer's status (pg 754)

  //Task 2
  #define timer0_interrupt_mask (*((volatile uint32_t *) 0x40030018)) //Enables interrupts for timer0 (pg 745)
  #define NVIC_interrupt_enable (*((volatile uint32_t *) 0xE000E100)) //Enables the difference indices of the NVIC interrupt table (pg 142)

  #define clock_gate_control2 (*((volatile uint32_t *)0x400FE608)) //Another clock gate register which allows us to connect the clock to the diferrent GPIO ports (pg 340)
  #define port_f_control (*((volatile uint32_t *) 0x4002552C)) //Controls the peripheral functions of port b. Nibble #s correspond to MUX selection for each pin. Clear to have pin act as normal GPIO (pg 688)
  #define port_f_analogue (*((volatile uint32_t *) 0x40025528)) //Controls the analogue function of port b GPIO pins. Bit #s correspond to pin. Clear to disable analogue functions (pg 687)
  #define port_f_direction (*((volatile uint32_t *) 0x40025400)) //Controls whether port b GPIO pins are inputs or outbuts. Bit #s correspond to pin. Clear for input. Set for output. (pg 663)
  #define port_f_alternate (*((volatile uint32_t *) 0x40025420)) //Controls alternate functions of port b GPIO pins. Bit #s correspond to pin. Set to make a peripheral signal. Clear to use as GPIO (pg 671)
  #define port_f_digital_en (*((volatile uint32_t *) 0x4002551C)) //Enables pin's digital function. Bit #s correspond to pin. Set to enable digital. Clear to disable. (pg 682)
  #define port_f_all_data (*((volatile uint32_t *) 0x400253FC)) //Reading/Writing for all data in port b's GPIO pins. Bit #s correspond to pin. (pg 662)
  #define port_f_data0 (*((volatile uint32_t *) 0x40025004)) //Reading/Writing for pin 0 only. Bit #s correspond to pin. (pg 662)
  #define port_f_data4 (*((volatile uint32_t *) 0x40025040)) //Reading/Writing for pin 4 only. Bit #s correspond to pin. (pg 662)

  #define port_f_interrupt_mask (*((volatile uint32_t *) 0x40025410)) //Controls whether port f can genearate interrupts (pg 667)
  #define port_f_interrupt_clear (*((volatile uint32_t *) 0x4002541C)) //Clears the status of port f's interrupt register (pg 670)
  #define port_f_interrupt_sense (*((volatile uint32_t *) 0x40025404)) //Configures what triggers an interrupt in port f (pg 664)
  #define port_f_raw_interrupt (*((volatile uint32_t *) 0x40025414)) //The raw status of port f's interrupt (pg 668)
  #define port_f_IBE (*((volatile uint32_t *) 0x40025408)) //Determines if an interrupt is triggered by both edges or just one (pg 665)
  #define port_f_interrupt_event (*((volatile uint32_t *) 0x4002540C)) //Determines if a low or high triggers port f's interrupt (pg 666)

  #define port_b_interrupt_mask (*((volatile uint32_t *) 0x40005410)) //Controls whether port f can genearate interrupts (pg 667)
  #define port_b_interrupt_clear (*((volatile uint32_t *) 0x4000541C)) //Clears the status of port f's interrupt register (pg 670)
  #define port_b_interrupt_sense (*((volatile uint32_t *) 0x40005404)) //Configures what triggers an interrupt in port f (pg 664)
  #define port_b_raw_interrupt (*((volatile uint32_t *) 0x40005414)) //The raw status of port f's interrupt (pg 668)
  #define port_b_IBE (*((volatile uint32_t *) 0x40005408)) //Determines if an interrupt is triggered by both edges or just one (pg 665)
  #define port_b_interrupt_event (*((volatile uint32_t *) 0x4000540C)) //Determines if a low or high triggers port f's interrupt (pg 666)


  //Lab 3 Defines

  #define RCC (*((volatile uint32_t *) 0x400FE060)) //Initial Clock configuration register
  #define RCC2 (*((volatile uint32_t *) 0x400FE070)) //Second clock configuration register used to get different clock frequencies. Overrides RCC
  #define RIS (*((volatile uint32_t *) 0x400FE050)) //Raw interrupt status of the system control modules (pg 244)
  #define MISC (*((volatile uint32_t *) 0x400FE058)) //Clear for system control interrupts (pg 249)
  #define IMC (*((volatile uint32_t *) 0x400FE054)) //Mask for system control interrupts (pg 247)

  #define ADC_RCGC (*((volatile uint32_t *) 0x400FE638)) //Running Clock Gate control for ADC (pg 352)
  #define ADC_SSEN (*((volatile uint32_t *) 0x40038000)) //Controls which sample sequencers are on (pg 821)
  #define ADC_EVENT (*((volatile uint32_t *) 0x40038014)) //Event Mux: Selects what event triggers the ADC (pg 833)
  #define ADC_SSMUX3 (*((volatile uint32_t *) 0x400380A0)) //Sample Sequencer Input Selector: defines analog input configuration for sample sequencer 3 (pg 875)
  #define ADC_SSCTL3 (*((volatile uint32_t *) 0x400380A4)) //Sample Sequencer Control: Selects the input for sample sequencer 3 (pg 876)
  #define ADC_IM (*((volatile uint32_t *) 0x40038008)) //Interrupt Mask: Controls the interrupts for all the sample sequencers (pg 825)
  #define ADC_IC (*((volatile uint32_t *) 0x4003800C)) //Interrupt Clear: Clears the interrupts for the sample sequencers (pg 828)
  #define ADC_Manual (*((volatile uint32_t *) 0x40038028)) //Sample Sequencer Initiate: allows for manual initiation of a sample sequence (pg 845)
  #define ADC_FIFO3 (*((volatile uint32_t *) 0x400380A8)) //Output of sample sequencer 3 (pg 860)
  #define ADC_RIS (*((volatile uint32_t *) 0x40038004)) //Raw interrupt status
  #define ADC_SAMPLE_RATE (*((volatile uint32_t *) 0x40038FC4)) //Programmed Sample Rate Register (pg 891)
  #define ADC_PP (*((volatile uint32_t *) 0x40038FC0)) //We use this register for setting the maximum sample rate (pg 889)

  #define UART_RCGC (*((volatile uint32_t *) 0x400FE618)) //Runing Clock Gate control for UART (pg 344)
  #define GPIO_RCGC (*((volatile uint32_t *) 0x400FE608)) //Running Clock Gate control for GPIO pins (pg 340)
  #define GPIO_Alternate_A (*((volatile uint32_t *) 0x40004420)) //Controls whether the GPIO pins are used as I/O or alternate functions (pg 671)
  #define GPIO_CTL (*((volatile uint32_t *) 0x4000452C)) //Mux selection for alternate GPIO functions (pg688)
  #define GPIO_LOCK_A (*((volatile uint32_t *) 0x40004520)) //Lock register for port a (pg 684)
  #define GPIO_COMMIT_A (*((volatile uint32_t *) 0x40004524)) //Allows other registers to be written to (pg 685)
  #define GPIO_ENABLE_A (*((volatile uint32_t *) 0x4000451C)) //Enables pin's digital function. Bit #s correspond to pin. Set to enable digital. Clear to disable. (pg 682)
  #define GPIO_DIR_A (*((volatile uint32_t *) 0x40004400)) //Controls whether port b GPIO pins are inputs or outbuts. Bit #s correspond to pin. Clear for input. Set for output. (pg 663)

  #define UART_CTL (*((volatile uint32_t *) 0x4000C030)) //Control register for UART modules (pg 918)
  #define UART_IBRD (*((volatile uint32_t *) 0x4000C024)) //Register for the integer portion of the bard rate equation (pg 914) Number generation method can be found on pg 896
  #define UART_FBRD (*((volatile uint32_t *) 0x4000C028)) //Register for the fraction portion of the bard rate equation (pg 915) Number generation method can be found on pg 896
  #define UART_LCRH (*((volatile uint32_t *) 0x4000C02C)) //Arbiter of final configurations for the UART modules (pg 916)
  #define UART_TCC (*((volatile uint32_t *) 0x4000CFC8)) //Specifies the clock connected to the UART modules (pg 916)
  #define UART_DATA (*((volatile uint32_t *) 0x4000C000)) //Data module both for recieving and transmitting (pg 906)
  #define UART_FLAG (*((volatile uint32_t *) 0x4000C018)) //Information for the status of the data register, we will poll from bit 3 to see if it is busy or not (pg 911)


  //create shorthand for delay
  #define delay for(int j = 0; j < 1000; j++){};
  #define delay2 for(int j = 0; j < 1000000; j++){};

```

```
•
•   //Lab 2
•     //Task 1
•       void timer_start(void); //Sets a 1 Hz timer
•       void timer_off(void); //Turns off timer0A
•       int timer_read(void); //Outputs a 1 every second, 0 otherwise
•       void timed_change_colors(void); //Changes colors every second in a cycle
•       void timed_traffic_light(void); //Traffic light from Lab 1 but modified to
  incorporate timed changes
•
•       int timed_port_b4_input(void); //Modified button input to account for time
•       int timed_port_b5_input(void); //Modified button input to account for time
•
•       void timer0A_mask_start(void); //Enables timer0A to create interrupts
•
•       //Task 2
•       void interrupt_ctrl_change_colors(void); //Changes colors every second
•       void blue_blink(void); //blinks the blue onboard led every second
•       void port_f_digital_start(void); //enables port f pins as GPIO only
•
•       void pb0_input_start(void); //Makes pf0 an input
•       void pb4_input_start(void); //Makes pf4 an input
•
•       unsigned long port_f0_input(void); //reads data through pf0
•       unsigned long port_f4_input(void); //reads data through pf4
•
•       void port_f_interrupt_enable(void); //Enables port f's interrupt
•       void port_f_clear_interrupt(void); //Clears port f's interrupt
•
•       void port_b_interrupt_enable();//Enables port b's interrupt
•       void port_b_clear_interrupt(void);//Clears port b's interrupt
•       void interrupt_traffic_light(int *ptr); //traffic light program using interrupts
  which are triggered by off-board buttons. Takes an interger pointer
•
•
•
•   //Lab 1
•       //Task 1
•       void onboard_led_start(void); //Allows for control over the onboard LEDs via
  led_data or change_green or change_red. Useful for debugging
•
•       void change_colors(void); //Causes onboard led to cycle quickly through 7 colors.
•
•       void switch_control_set(void); //Allows the onboard LED to be controlled by the
  two onboard switches. SW1
•       void switch_control(void); //Sets SW1 to activate the red LED and SW2 to activate
  the green LED.
•
•
•       //Task 2
•       void port_b_digital_start(void); //Activates the digital GPIO pins 0-5 of port b.
   Deactivating all other functions
•
•       void pb0_output_start(void); //Sets pb0 to output
•       void pb1_output_start(void); //Sets pb1 to output
•       void pb2_output_start(void); //Sets pb2 to output
•
•       void pb0_on(void); //Sets pin 0 of port b
•       void pb1_on(void); //Sets pin 1 of port b
•       void pb2_on(void); //Sets pin 2 of port b
•       void pb0_off(void); //Clears pin 0 of port b
•       void pb1_off(void); //Clears pin 1 of port b
•       void pb2_off(void); //Clears pin 2 of port b
•
•       void pb4_input_start(void); //Makes pb4 an input
•       void pb5_input_start(void); //Makes pb5 an input
•
•       unsigned long port_b4_input(void); //reads data through pb4
•       unsigned long port_b5_input(void); //reads data through pb5
•
•       void traffic_light_start(void); //Sets up traffic light
•       void traffic_light(void); //Runs task 2
•
•
•       //Lab 3 Task 1
•       void reset_clk(void); //Resets the Clock Frequency to 16MHz
•       void set_clk_80MHz(void); //Changes the clock frequency to 80MHz
•
•       void UART_Init(void); //Initializes the UART module 0
•       void ADC_Init(void); //Initiates an ADC that is triggered by timer0A and reads
   from the temperture sensor
•
•       void UART_Print(double value); //Takes in a double and sends it through the UART
   module 0
•
•       void PLL_Init(void); //Generic function to work as a short hand for all the PLL
   functions
•       void PLL_Final(void); //Generic function to work as a short hand for the
   finalization of a new clock
•
•       void reset_clk(void); //Changes the system clock back to the 16MHz using the main
   oscillator
•       void set_clk_80MHz(void); //Changes the system clock to 80MHz using the main
   oscillator and the PLL
•       void set_clk_4MHz(void); //Changes the system clock to 4MHz using the main
   oscillator and the PLL
•       void set_clk_66MHz(void); //Changes the system clock to 66Mhz using the main
   oscillator and the PLL
•       void set_clk_50MHz(void); //Changes the system clock to 50MHz using the main
   oscillator and the PLL
```

```
•       void set_clk_20MHz(void); //Changes the system clock to 20MHz using the main
   oscillator and the PLL
•
•       void Timer_Init(uint32_t t); //Changes the timer to have a new load to count down
   from which is the value passed into the function
•
•       void ADC_interrupt_clear(void); //Clears the interrupt status of the ADC
•       void ADC_interrupt_enable(void); //Enables the ADC to generate interrupts
•
•
•       //Lab 5
•       void ADC_Analog_Init(void); //Initalizes the ADC to take in an analog input from
   pb4 every time the timer0 times out
•       void GPIO_b4_analog(void); //Initilaizes pb4 as an analog input into the ADC
•       void fast_timer_start(void); //Initializes a timer with a load of 1.
•
•
•
•
•
•  /* GPIO Settings
   -------------------------------------------------------------------------------------
   --
•      These functions pertain to GPIO functions*/
•
•  void GPIO_b4_analog(void)
•  {
•    clock_gate_control |= 0x2; //connects the clock to port b
•
•    delay
•    delay
•
•    port_b_digital_en &= ~0x10; //turns on digital functunality for port b4
•    port_b_control = port_b_control & ~0x0F0000; //disables port b alternate functions
   for pin 4
•    port_b_alternate &= ~0x10; //turns off alternate functions for port b4
•    port_b_analogue |= 0x10;  //enable analogue functions of port b4
•  }
•
•  /* UART Settings
   -------------------------------------------------------------------------------------
   --
•      These functions initiate different UART settings*/
•
•  void UART_Init(void)
•  {
•    UART_RCGC |= 0x1; //Connects Running clock
•    GPIO_RCGC |= 0x1; // Connects Port A
•
•
```

```
    GPIO_LOCK_A = 0x4C4F434B; //Unlocks Port A
    GPIO_COMMIT_A |= 0x3; //Allows alernate functions for port a to be set

    GPIO_Alternate_A |= 0x3; //Enables alternate functions for pa1 pa2
    GPIO_CTL &= ~0xFF; //Clears Register
    GPIO_CTL |= 0x11; //Sets MUX values
    GPIO_DIR_A |= 0x2; //Sets direction of PA1 to out
    GPIO_ENABLE_A |= 0x3; //Enables digital function of PA1 and PA0

    UART_CTL &= ~0x1; //Disables UART
    UART_LCRH &= ~0x10; //Flushes Fifo
    UART_IBRD = 0x68; //Integer value of baud rate for ClkDiv = 16
    UART_FBRD = 0xB; //Fraction value of baud rate for ClkDiv = 16
    UART_LCRH = 0x60; //Sets frame to 8 bits, disables parity, and configures a single
  stop bit
    UART_TCC = 0x5; //Sets Clock to PIOSC. Independent of the system clock

    UART_CTL |= 0x100; //Enables Transmission
    UART_CTL |= 0x1; //Enables UART

  }

  void UART_Print(double value)
  {
    char array[10];
    snprintf(array, 10, "%.3f\n\r", value); //Stores passed in value to the char array

    for(int i = 0; i < 10; i++)
    {
      while(UART_FLAG & 0x8) {} //checks to make sure that the last message sent before
  starting a new one
      UART_DATA = array[i]; //Transmits the characters one at a time
    }
  }



  /* Analog Digital Converter Settings
  ---------------------------------------------------------------------------------------
  --
      These functions initiate different ADC settings*/

  void ADC_Init(void)
  {
    ADC_RCGC |= 0x1; //Connects ADC to running clock
    delay
    ADC_SSEN &= ~0xF; //Disables all sample sequencers
    ADC_EVENT = 0x5000; //Sets timer to be trigger for SS 3 || if we need to manually
  start the SS we will set to 0 so we can use the ADC_Manual
    ADC_SSMUX3 &= ~0xF; //Not used here but set to be sure
  ADC_SSCTL3 = 0xE; //Selects the temperture sensor as the input

  timer0_control &= ~0x1; //Disables Timer for Reconfiguring
  timer0_control |= 0x20; //Enables the timer's trigger
  timer0_control |= 0x1; //Renables Timer

  ADC_SSEN |= 0x8; //Enable only sample sequencer 3
}

void ADC_Analog_Init(void)
{
  ADC_RCGC |= 0x1; //Connects ADC to running clock
  delay
  ADC_SSEN &= ~0xF; //Disables all sample sequencers
  ADC_PP &= ~0xF; //Clears Max Sample Rate
  ADC_PP |= 0x7; //Sets Max Sample Rate to 1 Msps
  ADC_SAMPLE_RATE = 0x7; //sets sample rate to 1 Msps
  ADC_EVENT = 0x5000; //Sets timer to be trigger for SS 3 || if we need to manually
start the SS we will set to 0 so we can use the ADC_Manual
  ADC_SSMUX3 = 0xA; //Select AIN10 as input
  ADC_SSCTL3 = 0x6; //Selects the temperture sensor as the input

  timer0_control &= ~0x1; //Disables Timer for Reconfiguring
  timer0_control |= 0x20; //Enables the timer's trigger
  timer0_control |= 0x1; //Renables Timer

  ADC_SSEN |= 0x8; //Enable only sample sequencer 3
}



/* Internal Clock Settings
---------------------------------------------------------------------------------------
--
    These functions set up new clock rates for the boards internal clock */

void PLL_Init(void)
{
  RCC &= ~0x400000; //Turns off SYSDIV
  RCC |= 0x800; //Sets Bypass
  RCC &= ~0x7C0; //Clears XTAL
  RCC |= 0x540; //Sets XTAL to 16MHz

  RCC2 |= 0x80000000; //Turns on RCC2

  RCC2 |= 0x800; //Sets Bypass RCC2
  RCC2 &= ~0x70; //Clears OSCSRC2

  RCC |= 0x400000; //Turns on SYSDIV
}
```

```
void PLL_Final(void)
{

  while((RIS | 0x40) == 0) {} //waits until clock says its ready through an interrupt

  RCC2 &= ~0x800; //Turns off bypass in RCC2

}


void reset_clk(void)
{
  PLL_Init();

  RCC = 0x070E0AC0; //Resets the Clock to Normal
  RCC2 &= ~0x80000000; //Turns off RCC2
}

void set_clk_80MHz(void)
{
  PLL_Init();

  RCC2 |= 0x80000000; //Turns on RCC2

  RCC2 |= 0x800; //Sets Bypass RCC2
  RCC2 &= ~0x70; //Clears OSCSRC2

  RCC2 |= 0x40000000; //Sets DIV400
  RCC2 &= ~0x2000; //Clears and PWRDN2
  RCC2 &= ~0x1FC00000; //Clears SYSDIV
  RCC2 |= 0x1000000; //Sets SYSDIV to divide by 4 + 1

  while((RIS & 0x40) == 0) {} //waits until clock says its ready through an interrupt

  RCC2 &= ~0x800; //Turns off bypass in RCC2

}

void set_clk_4MHz(void)
{
  PLL_Init();


  RCC2 |= 0x40000000; //Sets DIV400

  RCC2 &= ~0x2000; //Clears and PWRDN2

  RCC2 &= ~0x1FC00000; //Clears SYSDIV
  RCC2 |= 0x18C00000; //Sets SYSDIV to divide by 99 + 1
```

```

    PLL_Final();
  }


  void set_clk_66MHz(void)
  {
    PLL_Init();

    RCC2 |= 0x40000000; //Sets DIV400
    RCC2 &= ~0x1FC00000; //Clears SYSDIV
    RCC2 |= 0x1400000; //Sets SYSDIV to divide by 5 + 1
    RCC2 &= ~0x6000; //Clears USBPRWDN and PWRDN2
    RCC2 &= ~0x70; //Clears OSCSRC2
    RCC2 |= 0x80000000; //Sets USERRC2

    PLL_Final();
  }

  void set_clk_50MHz(void)
  {
    PLL_Init();

    RCC2 |= 0x40000000; //Sets DIV400
    RCC2 &= ~0x1FC00000; //Clears SYSDIV
    RCC2 |= 0x1C00000; //Sets SYSDIV to divide by 5 + 1
    RCC2 &= ~0x6000; //Clears USBPRWDN and PWRDN2
    RCC2 &= ~0x70; //Clears OSCSRC2
    RCC2 |= 0x80000000; //Sets USERRC2

    PLL_Final();
  }

  void set_clk_20MHz(void)
  {
    PLL_Init();

    RCC2 |= 0x40000000; //Sets DIV400
    RCC2 &= ~0x1FC00000; //Clears SYSDIV
    RCC2 |= 0x4A00000; //Sets SYSDIV to divide by 5 + 1
    RCC2 &= ~0x6000; //Clears USBPRWDN and PWRDN2
    RCC2 &= ~0x70; //Clears OSCSRC2
    RCC2 |= 0x80000000; //Sets USERRC2

    PLL_Final();
  }


```

```
/* Timers
-------------------------------------------------------------------------------------------
-----------------------------------

*/
void timer_start(void)
{
  clock_control_time |= 0x1; //connects clock to timer0
  delay
  timer0_control &= ~0x1; //turns of timer0
  delay
  timer0A_config &= 0x0;
  timer0A_mode |= 0x2;
  timer0A_mode &= ~0x1;
  timer0A_mode &= ~0x10;
  timer0A_load = 0x00F42400; //loads timer with 16M
  timer0_control |= 0x1; //enables timer
}

void timer_off(void)
{
  timer0_control &= ~0x1;
}

void Timer_Init(uint32_t t)
{
  timer0_control &= ~0x1; //Disables timer
  delay
  timer0A_load = t; //Loads timer with new load
  timer0_control |= 0x1; //Enables timer
}

void fast_timer_start(void)
{
  clock_control_time |= 0x1; // connects clock to timer0
  delay
  timer0_control &= ~0x1; //turns off timer0
  delay
  timer0A_config &= 0x0;
  timer0A_mode |= 0x2;
  timer0A_mode &= ~0x1;
  timer0A_mode &= ~0x10;
  timer0A_load = 0x001; //loads timer with 1
  timer0_control |= 0x1; //enables timer
}


/* Interrupts
-------------------------------------------------------------------------------------------
-----------------------------------
```

```

*/

void ADC_interrupt_clear(void)
{

  ADC_IC |= 0x8; //Clears interrupt

}


void ADC_interrupt_enable(void)
{
  ADC_IM |= 0x8; //Unmasks the ADC_interrupt
  NVIC_interrupt_enable |= 0x20000; //Enabels the interrupt on the NVIC
}

void port_b_clear_interrupt(void)
{
  port_b_interrupt_clear |= ~0x0; //clears the interrupt state of port b
}

void port_b_interrupt_enable(void)
{
  port_b_interrupt_mask = 0x0; //disables port b interrupts
  port_b_interrupt_sense &= ~0x30; //sets the interrupt to be edge-sensitive
  port_b_IBE &= ~0x30; //sets interrupts to trigger on a single edge
  port_b_interrupt_event |= 0x30; //sets interrupt to trigger on rising edge
  port_b_interrupt_clear |= 0x30; //clears the interrupt register
  port_b_raw_interrupt = 0x0; //Makes double sure the interrupt register is clear
  port_b_interrupt_mask |= 0x30; //Allows pins 4 and 5 to trigger interrupts
  NVIC_interrupt_enable |= 0x0000002; //Enables the interrupt for port b
}

void port_f_interrupt_enable(void)
{
  port_f_interrupt_mask = 0x0; //disables port f's interrupts
  port_f_interrupt_sense &= ~0x11; //configures port f's interrupts to be edge
triggered
  port_f_IBE &= ~0x11; //configures interrupts to be triggered by a single edge
  port_f_interrupt_event |= 0x11; //configures interrupt to trigger via the rising edge
  port_f_interrupt_clear |= 0x11; //clears port f's raw interrupt status
  port_f_raw_interrupt = 0x0; //makes doubly sure that port f's raw interrupt status is
clear
  port_f_interrupt_mask |= 0x11; //enables pin 0 of port f to trigger interrupts
  NVIC_interrupt_enable |= 0x40000000; //enables port f's interrupt in the NVIC array
}

void onboard_switches_interrupt_enable(void)
{
  port_f_interrupt_mask = 0x0; //disables port f's interrupts
  port_f_interrupt_sense &= ~0x11; //configures port f's interrupts to be edge triggered
  port_f_IBE &= ~0x11; //configures interrupts to be triggered by a single edge
  port_f_interrupt_event &= ~0x11; //configures interrupt to trigger via the rising edge
  port_f_interrupt_clear |= 0x11; //clears port f's raw interrupt status
  port_f_raw_interrupt = 0x0; //makes doubly sure that port f's raw interrupt status is clear
  port_f_interrupt_mask |= 0x11; //enables pin 0 of port f to trigger interrupts
  NVIC_interrupt_enable |= 0x40000000; //enables port f's interrupt in the NVIC array
}








void interrupt_traffic_light(int *x)
{
  //x is an integer pointer that is passed in so that this function can coordinate and communicate with the rest of the program
  if(timer_read() && (*x <= 7)) //if the passed in x is not deactivated (x > 7) then it will increment each second
  {
    *x = *x + 1;
  }
  if(*x == 0) //if the passed in x is 0 then the traffic light will switch to the stop state
  {
    pb0_on(); //red
    pb1_off(); //green
    pb2_off(); //yellow
  }
  if(*x == 4) //once the passed in x reaches 4 the traffic light will switch to the go state
  {
    pb0_off(); //red
    pb1_on(); //green
    pb2_off(); //yellow
    *x = -4; //resets the common counter to -4
  }
}

void port_f_clear_interrupt(void)
{
  port_f_interrupt_clear |= ~0x0; //clears all the bits in the raw interrupt register
}
```

```c


void port_f_digital_start(void)
{
  clock_gate_control = 0x10 | clock_gate_control; //activates port f

  delay
  delay

  portf_unlock = 0x4C4F434B;
  portf_commit |= 0x1;
  port_f_analogue = 0x0;  //disable analogue functions of port f
  port_f_control = port_f_control & ~0xFF000F; //disables port f alternate functions
pins 0 and 4
  port_f_alternate &= ~0x31; //turns off alternate functions for port f pins 0 and 5
  port_f_digital_en = 0x3F; //turns on digital functuality for port f pins 0 to 5
}

void pf0_input_start(void)
{
  port_f_direction &= ~0x1; //makes pf0 an input

}

unsigned long port_f0_input(void)
{

  return(port_f_data0);//returns 0 if unpressed != 0 if pressed
}

void pf4_input_start(void)
{
  port_f_direction = port_f_direction & ~0x10; //makes pf4 an input

}

unsigned long port_f4_input(void)
{
  return(port_f_data4);//returns 0 if unpressed != 0 if pressed
}

void blue_blink(void)
{
  if(timer_read()) //turns on led
  {
    led_data = BLUE;
    delay
  }
  led_data = 0x0;
}
```

```

void interrupt_ctrl_change_colors(void)
{
  static int i = 0; //counter variable kept consistent
  if(timer_read()) //changes the led with every pulse of the 1 Hz clock
  {
    if(i > 6)
    {
      led_data = 0x2;
      i = 0;
    }else{
      led_data += 2;
    }
    i++;
  }
  timer0_clear |= 0x1; //clears the timer's status
}

void timer0A_interrupt_start(void)
{
  timer0_interrupt_mask |= 0x1; //activates timer 0A interrupt
  NVIC_interrupt_enable |= 0x80000; //enables timer 0A interrupt in the NVIC table
}





int timer_read(void)
{
  if((timer0_interrupt_status) != 0)
  {
    timer0_clear |= 0x1;
    delay
    return 1;
  }else{
    return 0;
  }
}

void timed_change_colors(void)
{
  static int i = 0;
  if(timer_read())
  {
    if(i > 6)
    {
      led_data = 0x2;
      i = 0;
    }else{
        led_data += 2;
      }
      i++;
    }
  }

  void traffic_light_start(void)
  {
    port_b_digital_start();
    pb0_output_start();
    pb1_output_start();
    pb2_output_start();
    pb4_input_start();
    pb5_input_start();

    //starts off
    pb0_off(); //red led off
    pb1_off(); //green led off
    pb2_off(); //yellow led off
   }

  int timed_port_b4_input(void)
  {
    int i = 0;
    while(port_b_data4 != 0) //will increment i every second only while the button is
  pressed
    {
      if(timer_read())
      {
        i++;
      }
    }
    if(i > 1) //checks to see if the button was held down for two seconds
    {
      return(1);//returns 1 if pressed for two seconds
    }else
    {
      return 0; //returns 0 otherwise
    }
  }

  int timed_port_b5_input(void)
  {
    int i = 0;
    while(port_b_data5 != 0) //will increment i every second only while the button is
  pressed
    {
      if(timer_read())
      {
        i++;
      }
    }
    if(i > 1) //checks to see if the button was held down for two seconds
    {
      return(1);//returns 1 if pressed for two seconds
    }else
    {
      return 0;// 0 otherwise
    }
  }

  void timed_traffic_light(void)
  {
    int state = 0; //FSM state variable
    int ped_pressed = 1; //for counting and timing. 0 is it's "inactive" value

    while(1)
    {
      switch(state)
        {
          case 0 : //go state
            pb1_on();
            pb0_off();
            pb2_off();
            break;

          case 1 : //off
            pb1_off();
            pb2_off();
            pb0_off();
            break;

          case 2 : //warn state
            pb2_on();
            pb1_off();
            pb0_off();
            break;

          case 3 : //pause on yellow
            break;

          case 4 : //stop state
            pb0_on();
            pb1_off();
            pb2_off();
            break;

          case 5 : //pause on red
            break;
         }

    if(timed_port_b4_input() != 0x0)
    {
      if(state % 2 == 0)//turns off traffic light
      {
        ped_pressed = 0; //resets ped_pressed to inactive
        state = 1; //switches state to the off state
       }else if(state % 2 != 0) //resets traffic light to stop
      {
        state = 4;
        ped_pressed = 1;
       }
    }

    if((timed_port_b5_input() != 0x0) && (state == 0))//pedestrain button pushed in go
state
      {
        state = 2;
        ped_pressed = 1;
       }

     if((ped_pressed > 0) && timer_read()) //if ped_pressed is activated and the
traffic light is unpaused it will start to count
      {
        ped_pressed++;
      }

     if((ped_pressed > 5) && (state % 2 == 0)) //if the counter reachers 400000 and the
traffic light is unpaused it will move to the next unpaused state
      {
        if(state == 4) //resets to go state
      {
        state = 0;
        ped_pressed = 1; //resets counter
     } else if(state == 0) //if in go, go to stop
     {
       state = 4;
       ped_pressed = 1; //resets counter
     } else if(state == 2) //if in warn, go to stop
     {
       state += 2;
       ped_pressed = 1;
     }
   }
  }
}

void onboard_led_start(void)
{
 clock_gate_control |= 0x20; // enable Port F GPIO
 led_dir |= 0xE; // set PF1 to PF3 as output
 led_digital |= 0xE; // enable digital pin PF1 to PF3
 led_data &= ~0xE; //set all leds to off
}

void change_colors(void)
{
 int j = 0; //for the sake of delay loops
 led_data = RED; //initially sets led to red

 //cycles through
 for (j = 0; j < 1000000; j++) {}
    led_data = PURPLE;
 for (j = 0; j < 1000000; j++) {}
    led_data = BLUE;
 for (j = 0; j < 1000000; j++) {}
    led_data = TURQUOISE;
 for (j = 0; j < 1000000; j++) {}
    led_data = WHITE;
 for (j = 0; j < 1000000; j++) {}
    led_data = GREEN;
 for (j = 0; j < 1000000; j++) {}
    led_data = YELLOW;
 for (j = 0; j < 1000000; j++) {}
}

void switch_control_set(void)
{
 clock_gate_control |= 0x20; // enable Port F GPIO

 portf_unlock = 0x4C4F434B; //unlock code for port f's commit register
 portf_commit |= 0x11; //allows setting of pull up resistors for port f pins 0 and 4
 portf_pull_up = 0x11; //attaches pull up resistors to pins PF0 and PF5


 led_dir = 0x0E; // set pins 1 to 3 to outputs and 4 and 0 to inputs
 led_digital |= 0x1F; // enable digital logic in Port F
}

void switch_control(void)
{
  if(switch2 == 0){
    change_green = GREEN; //lights up the green led but allows the rest of the ports data to be unchanged
  }else{
    change_green = 0x0; //turns off green
  }
  if(switch1 == 0){
    change_red = RED; //lights up the green led but allows the rest of the ports data to be unchanged
    }else{
      change_red = 0x0; //turns off red
    }
   }

void port_b_digital_start(void)
{
  clock_gate_control = 0x2 | clock_gate_control; //activates port b

  delay
  delay

  port_b_analogue = 0x0;  //disable analogue functions of port b
  port_b_control = port_b_control & ~0xFFFFFF; //disables port b alternate functions
pins 0 to 5
  port_b_alternate &= ~0x3F; //turns off alternate functions for port b pins 0 to 5
  port_b_digital_en = 0x3F; //turns on digital functunality for port b pins 0 to 5
}

void pb0_output_start(void)
{
  port_b_direction |= 0x1; //changes pb0 to output
}

void pb0_on(void)
{
  port_b_data0 = 0x1; //sets pb0
}

void pb0_off(void)
{
  port_b_data0 = 0x0; //clears pb0
}

void pb1_output_start(void)
{
  port_b_direction |= 0x2; //changes pb1 to output
}

void pb1_on(void)
{
  port_b_data1 = 0x2; //sets pb1
}

void pb1_off(void)
{
  port_b_data1 = 0x0; //clears pb1
}

void pb2_output_start(void)
```

```
{
  port_b_direction |= 0x4; //changes pb2 to output
}

void pb2_on(void)
{
  port_b_data2 = 0x4; //sets pb2
}

void pb2_off(void)
{
  port_b_data2 = 0x0; //clears pb2
}

void pb4_input_start(void)
{
  port_b_direction = port_b_direction & ~0x10; //makes pb4 an input

}

unsigned long port_b4_input(void)
{
  return(port_b_data4);//returns 0 if unpressed != 0 if pressed
}

void pb5_input_start(void)
{
  port_b_direction = port_b_direction & ~0x20; //makes pb5 an input

}

unsigned long port_b5_input(void)
{
  return(port_b_data5);//returns 0 if unpressed != 0 if pressed
}


void traffic_light(void)
{
  int state = 0; //FSM state variable
  int ped_pressed = 1; //for counting and timing. 0 is it's "inactive" value

  while(1)
  {
    switch(state)
      {
        case 0 : //go state
          pb1_on();
          pb0_off();
          pb2_off();
```

```
     break;

    case 1 : //off
     pb1_off();
     pb2_off();
     pb0_off();
     break;

    case 2 : //warn state
     pb2_on();
     pb1_off();
     pb0_off();
     break;

    case 3 : //pause on yellow
     break;

    case 4 : //stop state
     pb0_on();
     pb1_off();
     pb2_off();
     break;

    case 5 : //pause on red
     break;
   }

  if((port_b4_input() != 0x0) && (state % 2 == 0))//turns off traffic light
   {
    ped_pressed = 1; //resets ped_pressed to inactive
    state = 1; //switches state to the off state
    delay
   } else if((port_b4_input() != 0x0) && (state % 2 != 0)) //resets traffic light
   {
    state = 0;
    delay
   }

  if((port_b5_input() != 0x0) && (state == 0))//pedestrain button pushed in go state
   {
    ped_pressed++; //activated
    state = 2;
    delay
   }

   if((ped_pressed > 0) && (state % 2 == 0)) //if ped_pressed is activated and the
traffic light is unpaused it will start to count
   {
    ped_pressed++;
   }
```

```

     if((ped_pressed > 400000) && (state % 2 == 0)) //if the counter reachers 400000 and the traffic light is unpaused it will move to the next unpaused state
      {
        if(state == 4) //resets to go state
      {
        state = 0;
        ped_pressed = 1; //resets counter
     } else if(state == 0) //if in go, go to stop
     {
       state = 4;
       ped_pressed = 1; //resets counter
     } else if(state == 2) //if in warn, go to stop
     {
       state += 2;
       ped_pressed = 1;
     }
   }
  }
}

#endif
```

humanMachineInterface.py

```
import serial                                   #Used to read in data from the UART
from serial import Serial
import pyautogui                                # Used to map keyboard to python

serialPort = serial.Serial(port = "COM6", baudrate=9600, bytesize=8, timeout=None,
stopbits=serial.STOPBITS_ONE, xonxoff = True) # These input values made sure we were
connected with the rright device
serialString = serialPort.readline()                    # Read the complete value each time

pyautogui.PAUSE = 0.01                                              # Make sure there is no
delay between consecutive values when the autogui library is used
jumpFlag = 0.0                                                              # Flag to make sure
that jump command is only sent once and then sent again only after relaxing the muscle

while(1):

   # Wait until there is data waiting in the serial buffer
   #if(serialPort.in_waiting > 0):

      # Read data out of the buffer until a carraige return / new line is found
      serialString = serialPort.readline().decode('ascii')
                                    # Decode the bytes data into string, so that I can print the values
      convString = "" + serialString[4] + serialString[5] + serialString[6] + serialString[7]
# Used this technique to convert the data into processable form of float
      print(float(convString))

      if(float(convString) > 1.0): # 3.11 is highest
         if jumpFlag == 0.0:                            # If muscle was relaxed before
                  pyautogui.press("space")
                  print("Above threshold")
                  jumpFlag = 1.0                                # Set threshold to prevent multiple
jumps
      else:
      #  pyautogui.press("m")
         jumpFlag = 0.0
```