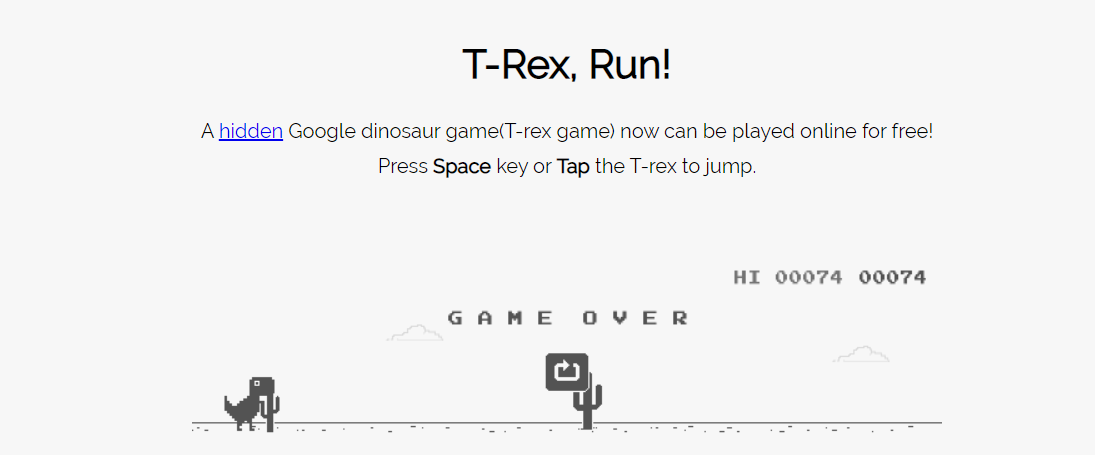
**EE 474 Lab 5: Human Computer Interface - Muscle Sensor**

**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).

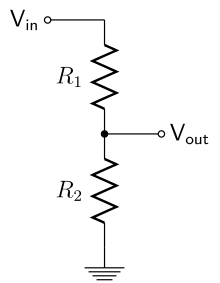
****

**Procedure:**

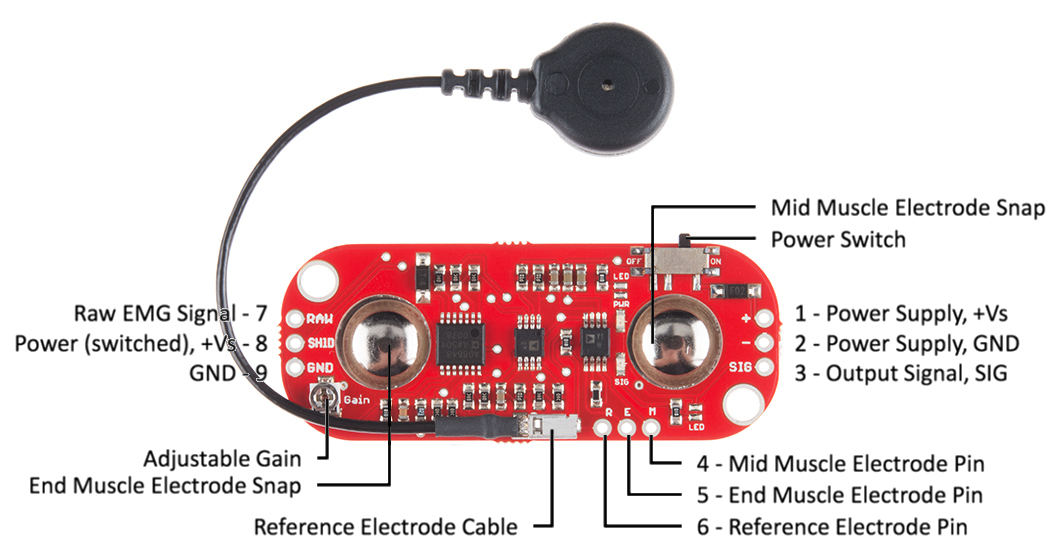
We broke this task into two parts, one for each of us. While Ricky worked on understanding the myoware sensor and getting its output to be read by the board’s ADC, Devyansh worked on interpreting the data sent into the computer through the UART so that it could control the computer.

**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 Vin = 3.3 and R2 was a 200 ohm potentiometer while R1 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 VDD. 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. 

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.

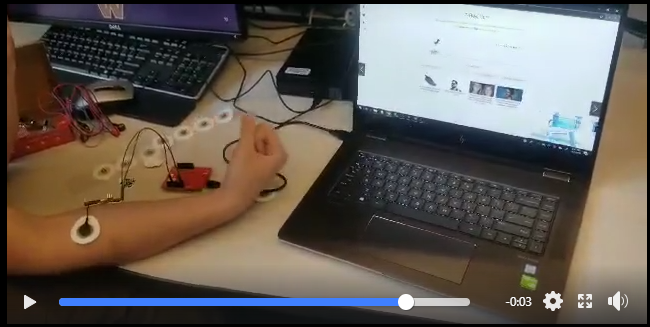
**Devyansh: Processing the data from UART and writing the Python script**

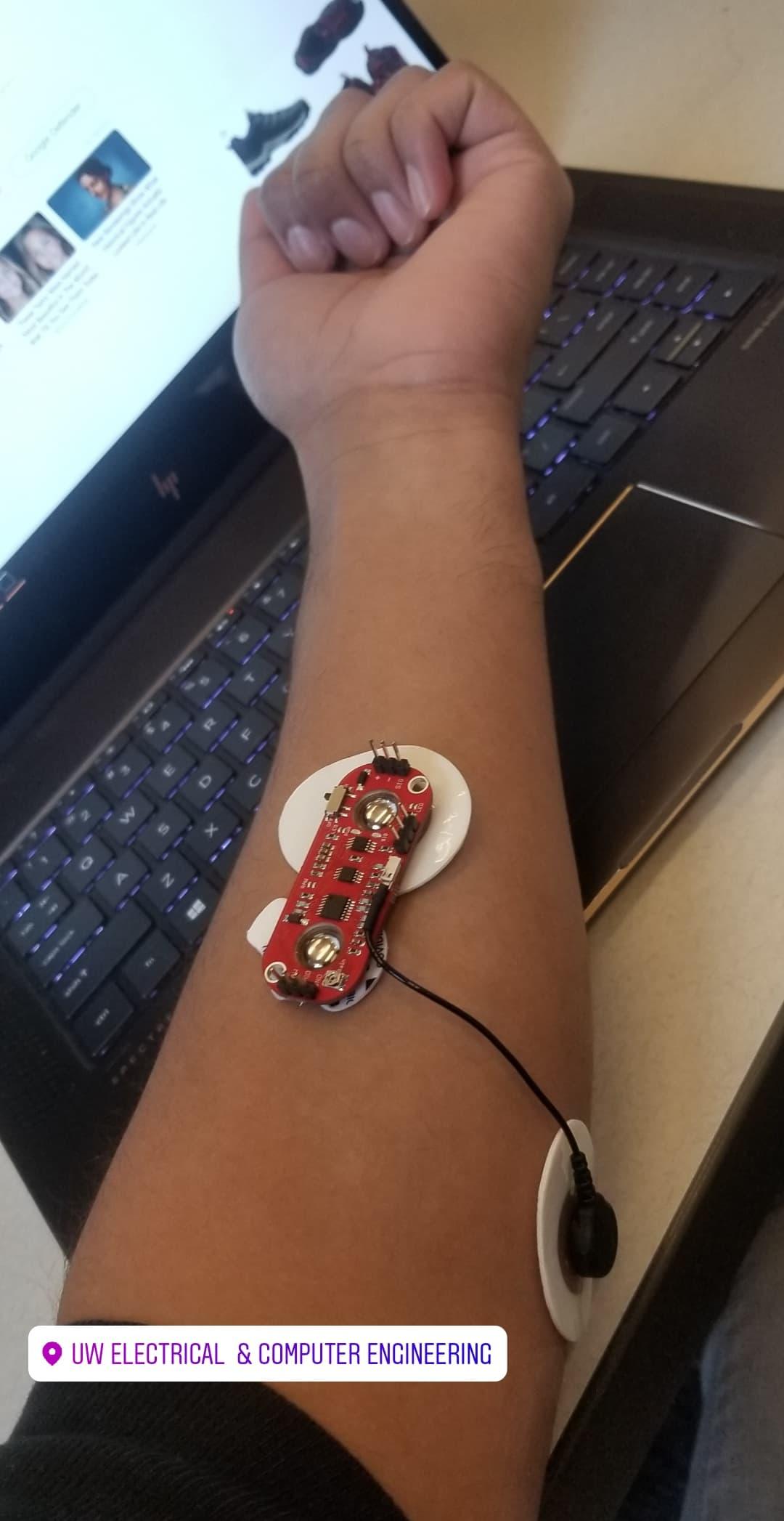
Devyansh was provided with data coming in from the sensor to PuTTY and then process it to play the game. To perform these tasks he first had to learn how to read in the serial data in python from the UART and by googling he found out about the library **PySerial** which could be used to read data from 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**

* #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
* }

* **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 functunality 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