Bal(l)ancing Act: PID controlled balancing of a ball using resistive touch sensing and dual-servo actuation

Caitlin Huang, Hugo So
MicroPs (E155) Fall 2021

***Abstract*—This document describes the design, implementation, and control of a ball-balancing panel. This project aims to create an integrated system using an STM32F401RE microcontroller to implement sensing and PID control. The system includes a resistive touch panel for sensing, and two SG90 micro servos for actuation. The mechanical components of the system are mostly 3D-printed. The project was successfully able to keep the ball actively balanced near the center of the panel, although the ball does not tend to settle at the setpoint.**

I. Introduction

The system is designed to integrate several subsystems into a control loop: the dynamics of the ball dictate its motion on the sensing panel, which generates data that indicates the location of the ball. The location data is fed to a PID controller that calculates a target angle for the servos. The current position of the ball and angle of the panel are accounted for by driving the servos to set the desired panel angle. The panel pose then dictates the dynamics of the ball, forming a closed control loop (Fig. 1).

For sensing the location of the ball, it is placed on a resistive touch panel, which scans between the x- and y-axes to inputs. The analog voltage read from the panel is then passed through a 12-bit analog-digital converter (ADC) on the STM32F401RE microcontroller unit (MCU), which converts the voltage to a digital signal that the MCU understands. A windowed average of two samples of the ADC output is computed to discard noisy inputs and converted back to an analog voltage value. The analog voltage is then converted to a position using a calibration curve and fed into a Proportional-Integral-Derivative (PID) controller. The PID controller then computes the error between the detected position and the desired setpoint, and uses the PID gains to calculate an appropriate control effort for each servo. Different PID gains are required for each axis due to different x and y dimensions of the touch panel. Note that the PID calculation for each axis is decoupled. The control effort was tuned to achieve desired servo angles via pulse-width modulation (PWM).

Apart from simply presenting a relatively simple but difficult-to-implement controls problem, this project is an analog for several controls problems with practical applications. The closed-loop control of actuators using external sensor data is similar to the balancing capabilities used in camera gimbals or Segways and hoverboards. The control problem of using the motion of a joint to actuate an effector is reminiscent of robotic arms used in industrial automation and mobile robots such as robotic quadrupeds.

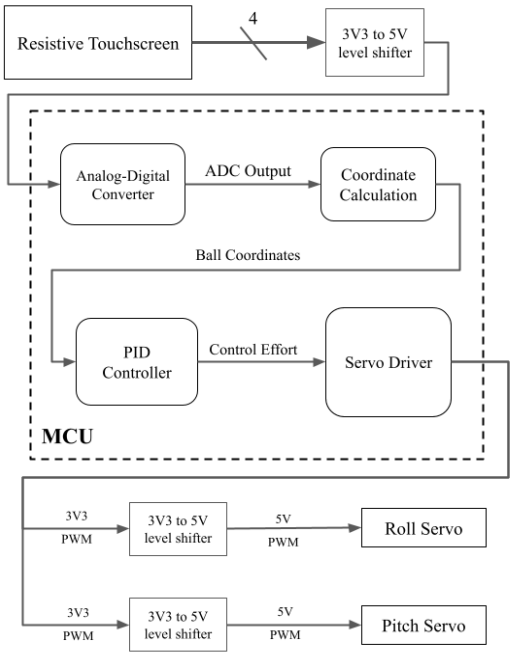


Fig. 1. System block diagram.

II. Physical Subsystem

Fig. 2. Photograph of completed system.

The structure and actuating components of the system are mostly composed of 3D printed parts and COTS fasteners (Fig. 3). The system is mounted on an aluminum baseplate, upon which a square base is mounted. The top of the base tower interfaces with a universal joint that allows the tilting panel to pitch and roll, but not yaw. The touch panel is mounted in a frame with attachment points to the universal joint to form the tilting panel. The top panel also has two attachment points so that linkages can be attached to actuate the panel.

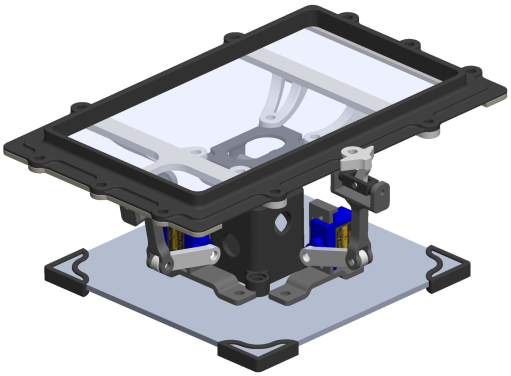


Fig. 3. CAD model of final mechanical subsystem.

The system was designed in CAD for 3D printing, and to verify fit and motion under actuation (Fig. 3). This was especially useful for estimating an initial servo angle vs. panel pose calibration curve, which allowed for informed design of the linkages and servo programming. The servos are mounted to a pair of motor mounts, also fastened to the baseplate. Driver linkages and free secondary linkages were used to connect the servos to the top panel, with actuating joints at multiple points to account for lateral movement in the linkages as the panel tilts. Washers and foam blocks were added to the system to decrease play in the linkage joints and jitter from the servo mounts. A bill of materials is included in Appendix B.

III. Sensor Implementation

A. Custom Sensor Design FPGA Design

The initial custom sensor design consisted of three primary layers; a top layer consisting of rows of aluminum foil adhered to a sheet of clear polyvinyl chloride (PVC), a middle spacer layer laser-cut from a sheet of low density polyethylene (LDPE), and a bottom layer of columns of aluminum foil adhered to a sheet of paper. These layers are backed with an aluminum plate, and the edges are clamped in place with a 3D-printed frame (Fig. 3). Wires are run to the aluminum foil through holes in the frame, which also clamps the wires to the foil, forming a mechanical connection. The intent of the system is that the top and bottom layers remain separated by the spacer sheet until the top sheet is depressed, such as by a ball rolling across the surface.

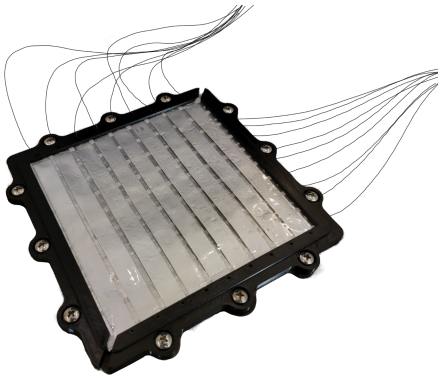


Fig. 4. Photograph of completed custom sensor assembly.

To get the position from the sensor to the microcontroller, the FPGA waited for inputs from the sensor and as master, sent an eight bit data output logic (x and y positions) over `SPI_MOSI` one bit at a time. This triggers an interrupt on the microcontroller to read the position. Interrupts were used so that the microcontroller did not need to continuously be polling for a position. The sensor was tested and verified in ModelSim. The waveforms are included in Appendix E. The clock and inputs were forced in simulation. When there is no contact, the sensor scans through the columns for row inputs, and when an input is detected, the sensor goes into a state to send data over to the microcontroller. Interrupts were enabled on the microcontroller and tested by turning on an LED when the data received matches the position sent over SPI from the microcontroller [3]. SystemVerilog HDL is included in Appendix D.

B. Resistive Touch Panel

The positions from the custom sensor ultimately were not precise enough mostly due to false positives, so a resistive panel was used instead. The touch panel has two layers separated with dot spacers. It senses something on the panel when the two resistive layers meet and create a resistive divider from the power to ground pins. Depending on where the force was detected, the resistances in the resistive divider would change to give a different output voltage. The output voltage from this divider is then read off to be converted to a position.

The touch panel was first tested to see which pins were connected to the same panel. To read a voltage, two of the connected pins should be power and ground, one pin is left tristated, and the remaining pin reads the output voltage. These pin functions change depending on the axis to read from [1]. How this was accounted for and implemented to read a digital position will be discussed further in Section V, Subsection A. Four 0.1µF bypass capacitors were added to smooth out some of the noisy readings.

IV. Servo Implementation

The system is actuated using a pair of SG90 9g micro servos. The servos drive a set of two-bar linkages with articulated joints that act on the top panel 3 inches away from the centerpoint. Each servo was calibrated using two samples (0 and 90 degrees) to get an equation that relates PWM duty cycle to servo angle. The servo angles were also related to the angle of the touch panel, first by using the CAD model and then by using an inclinometer on the touch panel. This allowed servo angles to be sent to the servos based on a desired panel angle, which is easier to relate to the ball position within the PID controller. The servo control equations are hard coded into the `setServo` function of the servo driver. Calibration curves are included in Appendix E.

A. Servo Selection

The original design for the system was actuated using a dual cable drive, which uses two servos to essentially puppet the top platform, which is always kept in tension. This design presented both benefits and challenges. One benefit to this

design is the stability of the movement given that the tension of the cables is tuned well. Unlike the current system, which experiences bouncing and significant backlash, especially at the ends of the panel further away from the servos, a cable driven system keeps both sides of the panel in tension, keeping it from bouncing. In the current system, which uses linkages, the universal joint in the center acts as a fulcrum for the panel, which is a long lever actuated at one end by the servo-driven linkage. This means that at the opposite end of the platform, the ball frequently bounces, reducing the pressure applied to the platform and yielding false negatives. This leads to strong oscillations when the ball is at that end of the platform. While a windowed average of the control effort lessened this issue, it persisted in the final design.

Even so, the linkage-driven design was selected for its repeatability and relative ease of control. The relationship between the servo angle and platform angle is almost exactly linear, except when both servos are at a relatively large angle. One important consideration was the range of motion offered by each alternative and the servos required to drive each. The linkage system can be driven with standard 180° servos, which allows for somewhat precise control of the servo position.

In contrast, in order to achieve more than about 10° of motion using the cable-driven system, continuous rotation servos would have been required in order to drive a spool the necessary angular distance. Because it is only possible to directly control the speed of continuous rotation servos and not the position, the implementation of precise setpoints and granular PID control would have been difficult .

B. Servo Driver

Two timers were used to drive the servos. Timer 5 was configured to PWM mode and sent PWM signals to two channels, one for each servo. The PWM frequency remained a constant 50Hz because that is the frequency required by the servos, and the duty cycle varied based on the desired servo angle. Timer 2 was used as a delay between PWM signals. To drive the servos, the `setServo` function takes in two angles and converts them to PWM signals for Timer 5. The two angle inputs can be the outputs of the `setPlatform` function that calculates servo angles based on the desired panel angle. The original delay using Timer 2 called after each time `setServo` was called would interfere with the PWM signal, changing the frequency or changing the `setServo` angle mid pulse. One way to mitigate that issue was to wait for the timer to finish counting up to a period before setting a new servo angle so that the new angle would not interfere with the current one.

V. Microcontroller Design

A. Analog-Digital Converter and Coordinate Calculation

The four wire touch panel pins had alternating GPIO `pinModes` to read the x and y ADC. In `getCoordinates`, the function loops through two for loops where GPIO `pinMode` and output is set initially before reading the analog voltage pin. The raw readings from the ADC are noisy so the values were filtered by comparing the most recent ADC reading to the previous one and an average was taken from every twenty samples. Then, using equations from calibration curves, the ADC values were converted to an x position for PID control.

B. PID Control

The PID controller first calls the coordinate calculation function, which gets the position measurement from the ADC and subtracts the desired position from the measured position in each axis to obtain a control error. Note that calculating the error in the `getCoordinates` function ensures that there is no delay between the ADC reading and the PID error. This error is passed into a discrete form of the PID controller equation (equation 1).

$$u[n] = K_p e[n] + K_i \sum_{i=0}^{n} e[i]dt + K_d(e[n] - e[n-1]) \quad (1)$$

The gains for PID control are specified as global variables in the file header, and unique P, I, and D gains are used for the x- and y- axes independently. In both cases, the integral gain is minimally used, and the proportional gain mostly dominates the response. The derivative gain is used more for the x-axis, which is larger and has a longer lever arm due to the geometry of the touchscreen. To implement PID control, an integral of the error was calculated by summing all previous errors in a pair of `integral_x/y` variables, and a derivative of the error is calculated by taking the difference between the current error and previous error. The error is then stored in the previous error variable to use on the next iteration of the PID control loop.

Using equation 1, the controller outputs a control effort for each servo. The control effort, which represents a pose for the touch panel, is converted to a servo angle using the calibration curves hard-coded into the `setServo` function. The servo driver then sets the position of the servo, as discussed in Section IV, Subsection B.

VI. Results

The project was successful in generating a system that sensed the ball's location on the touchscreen and responded by actuating the panel using a pair of servos. Independently, the parts of the system work, that is, moving the ball around on the platform causes a digital input to the MCU that allows it to continuously output coordinates to the terminal. The PID is then able to generate a setpoint for the pose of the panel in each axis. Given panel angle setpoints, the servo driver is able to calculate an appropriate servo angle and corresponding PWM signal to send to the servo to achieve that setpoint.

However, there are several points in the system where it breaks down in function. Because the ball does not always make solid contact with the touchscreen due to bouncing and force vectoring at the corners, the touch panel often reads a default value, which causes violent oscillations in the servo motion. The connection between the driving linkage and the

servo is not durable and begins to slip after testing for a few hours, causing actuation to be incomplete and inconsistent. A combination of these factors made PID difficult to tune because it was difficult to get repeatable inputs to the system and therefore difficult to consistently produce the same outputs. The resultant system is able to move the ball around the center of the platform, without being able to get the ball to settle near the center. The system performs slightly better when the position of the ball is constrained to a region near the center of the platform; at the edges, the servo and linkages sometimes fail to apply enough force to bring the ball back without bouncing it. This causes oscillations and a very noisy position input, which keep the ball stuck at the edge of the platform. The next section will discuss improvements and future work to address key issues present within the system.

VII. Future Work/Improvements

One simple way to address the issue of inconsistent detection of ball location is to simply use a heavier ball that is harder to bounce off the platform. This would ensure that a consistent ball location is detected, yielding consistent PID control effort outputs. The team found indication that this approach would yield better results, as the PID outputs obtained from pressing the ball into the platform are much more stable and consistent than those obtained by the ball rolling around.

A better filter on the input may also yield better controller results. This may range anywhere from a better low-pass filtering method to remove noise to more advanced Kalman filtering techniques such as an EKF.

Mechanical shortcomings in the system can be addressed by using metal linkages and motor mounts with less flex, more rigid servo spline-linkage interfaces, and joints with less slop to allow for more repeatable control. Stronger servos would also allow the system to move more repeatably without struggling and oscillating, especially when the ball is on the opposite end of the panel. A better touchscreen or sensor may also give the PID controller better error values.

A more extensive change to the system may include implementing control on the FPGA, which may perform the PID control calculations faster than the MCU does. This may address the lag that was present in the system at some points, which can be seen most noticeably when the ball overshoots the centerpoint and doesn't respond fast enough to return the ball toward the center. One or a combination of these changes is recommended for a second version of this system, and will likely yield more desirable results.

References

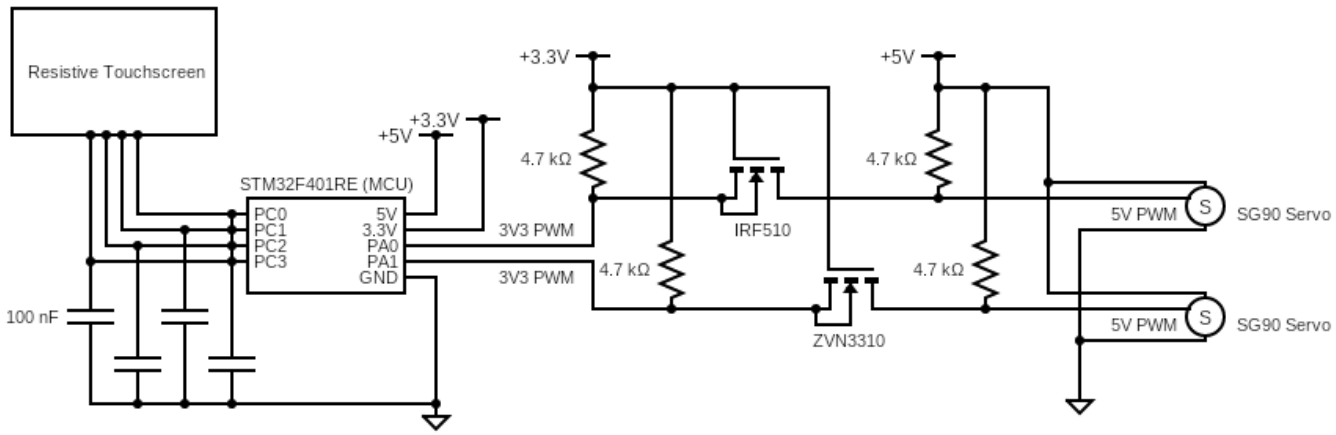
[1] Hantouch, USA, "How it works: 4-Wire Analog-Resistive Touch Screens," Sparkfun. https://www.sparkfun.com/datasheets/LCD/HOW%20DOES%20IT%20WORK.pdf

[2] "STM32F401xE Datasheet," STMicroelectronics, Rev. 3, January 2015. http://pages.hmc.edu/brake/class/e155/fa21/assets/doc/STM32F401RE_Datasheet.pdf

[3] H. Limm and A. Moody, "E155 Final Project: Magic See-Saw," E155 Project Reports, Fall 2019. http://pages.hmc.edu/harris/class/e155/projects19/Limm_Moody.pdf

[4] "RM0368 Reference Manual: STM32F401xD/E advanced Arm-based 32-bit MCUs," STMicroelectronics, Rev. 5, December 2018. http://pages.hmc.edu/brake/class/e155/fa21/assets/doc/STM32F401RE_Reference_Manual_RM0368.pdf

Appendix A. Circuit Schematic



Appendix B. Bill of Materials

Component	Vendor	Part Number	Cost	Quantity	Total
Resistive Touchscreen	Ebay	N/A	$9.99	1	$9.99
SG90 9g Micro Servo	Digi-Key	SER0006	$3.62	2	$7.24
1/8" x 6" 6061 Al	McMaster-Carr	8975K921	$6.64	1	$6.64
Stainless Steel Ball	McMaster-Carr	9529K27	$3.87	1	$3.87
Assorted Hardware	HMC Stockroom	N/A	<$5.00	N/A	$5.00
				Total	**$32.74**

Appendix C. C Code

main.c

```
// main.c

#include "STM32F401RE_FLASH.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_SPI.h"
#include "STM32F401RE_TIM.h"
#include "STM32F401RE_GPIO.h"
#include "STM32F401RE_ADC.h"
#include "main.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "STM32F401RE_USART.h"

#define USART_ID USART2_ID
#define cpol 0
#define cpha 0
#define X_PWM 0 // TIM5_ch1
#define Y_PWM 1 // TIM5_CH2
#define freq 50
#define clk_freq 84000000
#define xDes 0
#define yDes 0

#define kpx .8
#define kix 0
#define kdx 7.8

#define kpy 0.5
#define kiy 0
#define kdy 0.5

#define anglex_min -20
#define anglex_max 20

#define angley_min -9
#define angley_max 4

//float kp[] = {1.95, 2.31};
volatile double e_x = 0;
volatile double e_x1 = 0;
volatile double e_x2 = 0;
volatile double e_y = 0;
volatile double e_y1 = 0;
volatile double e_y2 = 0;
volatile double du = 0;
volatile double u = 0;
volatile double dv = 0;
volatile double v = 0;
volatile float xSum, ySum, xAvg, yAvg,  xPos, yPos, u_prev, v_prev, integral_x, derivative_x,
integral_y, derivative_y;

volatile int xADC, yADC, x,y;



void configureGPIO(){
   // enable clock to gpio
   RCC->AHB1ENR.GPIOBEN = 1;
```

```
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOCEN = 1;

    ////////////////////////////////////////////////////////
    // SPI Pins
    ////////////////////////////////////////////////////////

    // set spi pins to alt functions
    pinMode(GPIOB, SPI1_MOSI, GPIO_ALT);
    pinMode(GPIOB, SPI1_SCK, GPIO_ALT);

    // set cs pin to be read from FPGA
    pinMode(GPIOB, SPI1_CS, GPIO_INPUT);

    GPIOB->OSPEEDR = (0b11 << 2*SPI1_SCK); // set Pb3,6,7 to fastest clock
    // configure alternate functions to AF5 selection (0101)

    GPIOB->AFRL |= (0b0101 << 4*SPI1_MOSI);
    GPIOB->AFRL |= (0b0101 << 4*SPI1_SCK);

    ////////////////////////////////////////////////////////
    // PWM Pins
    ////////////////////////////////////////////////////////
    pinMode(GPIOA, X_PWM, GPIO_ALT); // set pinA0 as output
    pinMode(GPIOA, Y_PWM, GPIO_ALT); // set pinA0 as output

    GPIOA->AFRL |= (0b0010 << 4*X_PWM); //alternate function for TIM5 AF02 PA0
    GPIOA -> AFRL |= (0b0010 << 4*Y_PWM); //alternate function for TIM5 AF02 PA1

    digitalWrite(GPIOA, X_PWM, 0); // initialize at 0
    digitalWrite(GPIOA, Y_PWM, 0);

    ////////////////////////////////////////////////////////
    // ADC Pins
    ////////////////////////////////////////////////////////

    // set PC2 and PC3 as analog
    // pinMode(GPIOC, 2, GPIO_ANALOG); // adc X position
    // pinMode(GPIOC, 3, GPIO_ANALOG); // adc Y position
    // pinMode(GPIOC, )

    pinMode(GPIOB, 0, GPIO_OUTPUT); // set pinB0 as output

}

void SPI1_IRQHandler(){
    uint8_t data = spiRx8();
    if (data == 0b11100000){
        digitalWrite(GPIOB, 0, 1);
    }
}

volatile uint16_t xADC_prev, yADC_prev;


void getCoordinates(){
    //reading y
    for(int i = 1; i< 20; ++i){

        pinMode(GPIOC, 1, GPIO_OUTPUT);
        digitalWrite(GPIOC, 1, GPIO_LOW);

        pinMode(GPIOC, 3, GPIO_OUTPUT);
```

```
        digitalWrite(GPIOC, 3, GPIO_HIGH);


        yADC = ADC_read(2,0);
        if (abs(yADC - yADC_prev) < 600){ // discard noisy values
            ySum = yADC+ySum;
        }
        yAvg = ((ySum*3.3)/(4096))/i; // convert ADC sum to voltage
        yADC_prev = yADC; // store previous ADC


    } yPos = -19*yAvg+16.9; //convert voltage to position y
    e_y= yDes-yPos; // calculate error of that position y

    //reading x
    for(int i = 1; i< 20; ++i){

        pinMode(GPIOC, 3, GPIO_INPUT); // c3 is tristated
        pinMode(GPIOC, 0, GPIO_OUTPUT);
        digitalWrite(GPIOC, 0, GPIO_LOW);

        pinMode(GPIOC, 2, GPIO_OUTPUT);
        digitalWrite(GPIOC, 2, GPIO_HIGH);

        xADC_prev = xADC; // set previous val
        xADC = ADC_read(1,1);
        if (abs(xADC - xADC_prev) < 400){
            xSum = xADC+xSum;
        }
        xAvg = ((xSum*3.3)/(4096))/i; // convert ADC sum to voltage

    }xPos = 24.9*xAvg-20.7; // convert voltage to position x
    e_x = xPos-xDes; // calculate error of that position x
}

void pid(){
    getCoordinates(); // get the coordinates
    integral_x = integral_x + e_x;
    derivative_x = e_x - e_x1;
    // control effort for servo x
    u = (kpx * e_x) + (kix * integral_x) + (kdx * derivative_x);
    // Update previous value
    e_x1 = e_x;

    integral_y = integral_y + e_y;
    derivative_y = e_y - e_y1;
    // control effort for servo y
    v = (kpx * e_y) + (kix * integral_y) + (kdx * derivative_y);
    // Update previous value
    e_y1 = e_y;
}

////////////////////////////////////////////////////////
// +x +y: up and right
// +x -y: down and right
// -x +y: up and left
// +x +y: down and left
////////////////////////////////////////////////////////
void setServo(float anglex, float angley){
    if (anglex > anglex_max){
        anglex = anglex_max;
    } else if (anglex< anglex_min){
        anglex = anglex_min;
```

```
    }

    float pwmx, pwmy;
    pwmx = -0.05*anglex +6.5; //3.3 to 13  6.5 is 0 -> 2 is 90 (pwm) = -0.05(angle) +6.5 , 4.25
for 45
    pwmy = 0.047*angley +3.5; // 3.5 is 0 --> 7.7 is 90 (pwm) = 0.047(angle) + 3.5, 5.6 for 445
    pwm(TIM5, 50, pwmx, pwmy);
}

float setPlatform(float anglex, float angley, int i){
    float servo_angle[2];
    servo_angle[0] = 2.86*anglex + 12.9; // x
    servo_angle[1] = 1.99*angley + 14.4; // y
    return servo_angle[i];
}

int main(void) {
    configureFlash();
    configureClock(); // Set system clock to 84 MHz
    configureGPIO();
    configureTIM2(); // initialize tim 2 for delay
    pwmInit(TIM5); // initialize tim 5 for pwm

    spiInit(cpol, cpha);

    ADC_init();

    USART_TypeDef * USART = initUSART(USART_ID);  // usart for debugging purposes :)

    // setServo(setPlatform(10, 10, 0), setPlatform(0, 0, 1));
    // delay_pwm(1000);
    // setServo(setPlatform(0, 0, 0), setPlatform(0, 0, 1));
    // delay_pwm(1000);
    //pwm(TIM5, 50, 4, 10);


    setServo(0,0);

    while(1){
        uint8_t msg[96];
        // PB0 is safety pin so can unplug from 3.3V to stop servos at any time
        // manually toggle when ball is ready on plate
        if(digitalRead(GPIOB, 0)){
            pid();
            // u = u_prev + 0.5*(u-u_prev);
            // v = v_prev + 0.5*(v-v_prev);
            while (TIM5->CNT != TIM5->ARR.ARR);
            //if(e_x > 0){u=u+u*0.8;}
            setServo(u, v);

        } else { // set the plaform to zero
            setServo(setPlatform(0, 0, 0), setPlatform(0, 0, 1));
            delay_pwm(1000);
            v = 0;
            u = 0;
        }

        uint8_t i = 0;
        do {
          sendChar(USART, msg[i]);
          i += 1;
        } while (msg[i]);
```

```
        sprintf(msg, "%d %d %d %d\n\r", (int) (u*100), (int) (v), (int) (xPos*100), (int)
(e_x*100));
    }

}
```

TIMER.C

```
// ch_tim.c
// Source code for TIM functions

#include "STM32F401RE_TIM.h"
#include "STM32F401RE_RCC.h"

void pwmInit(TIM_TypeDef * TIM){
    RCC->APB1ENR.TIM5EN = 1;

    //control register CR
    TIM->CR1.CMS    = 0b00; //edge-aligned mode
    TIM->CR1.DIR    = 0; //upcounter
    TIM->CR1.CKD = 0b00; // input clock divided by 1
    TIM->CR1.CEN = 0; // disable counter

    //register ccmrx
    TIM->CCMR1.OC1M  = 0b110; //pwm mode 1
    TIM->CCMR1.OC2M = 0b110; //pwm mode 2
    TIM->CCMR1.OC1PE = 1; //enable preload register
    TIM->CCMR1.OC2PE = 1; //enable output 2 preload
    //smcr register
    TIM->SMCR.SMS  = 0b000; //slave mode disabled
    // ccer register
    TIM->CCER.CC1P    = 0; // ch 1 clock active high
    TIM -> CCER.CC2P = 0; //ch 2 clk active high
}


void pwm(TIM_TypeDef * TIM, uint32_t freq, float dc1, float dc2){
    // counter clk freq CK_CNT = fck_psc/(psc[15:0]+1)
    uint32_t period = 84e6/freq;
    float dutycycle1 = period*dc1/100;
    float dutycycle2 = period*dc2/100;
    TIM->CR1.UDIS       = 1; //disable update events
    TIM->CNT            = 0; // reset counter
    TIM->ARR.ARR        = period;
    TIM->CCR1.CCR1      = dutycycle1;
    TIM->CCR2.CCR2      = dutycycle2;
    //before starting counter, event generation register egr
    //TIM->EGR.UG         = 1; //initialize all registers

    TIM->CR1.UDIS       = 0; //enable update events

    TIM->CCER.CC1E      = 1; // on ch1
    TIM->CCER.CC2E       = 1;  //on ch2
    TIM->CR1.CEN        = 1; //enable counter
}

void pwm_end(TIM_TypeDef * TIM) {
    TIM->CR1.UDIS       = 1; //disable update events
    TIM-> CR1.CEN = 0; // disable counter
    TIM-> CCER.CC1E = 0; // disable output
    TIM->CCER.CC2E       = 0;  //disable output
}
```

```
void configureTIM2(){
   RCC->APB1ENR.TIM2EN = 1; // clock to tim9
   //control register CR
   TIM2->CR1.CMS   = 0b00; //edge-aligned mode
   TIM2->CR1.DIR   = 0; //upcounter
   TIM2->CR1.CKD = 0b00; // input clock divided by 1
   TIM2->CR1.URS = 1; //update when over or underflow
   TIM2->CR1.CEN = 0; // disable counter
   TIM2->SMCR.SMS  = 0b000; //slave mode disabled
}

void delay_pwm(uint32_t delayms){
   TIM2->CR1.UDIS = 1; // disable updates
   TIM2-> CNT = 0; //reset count
   TIM2->ARR.ARR = (84e3*delayms);
   TIM2->CR1.OPM = 1; //mode: one pulse
   TIM2->CR1.UDIS = 0; // renable updates
   TIM2-> CR1.CEN = 1; //enable counter
   while(!(TIM2->SR)&1);  //registers are updated
   TIM2-> SR = 0; // clear-> no updates
   TIM2->CR1.CEN = 0; // disable counter
}
```

ADC.c

```
#include "STM32F401RE_ADC.h"


void ADC_init(){
   RCC->APB2ENR.ADC1EN = 1; //enable clk to adc1
   ADC1->CCR.ADCPRE = 0b01; // adcclk = pclk2/4
   ADC1->CR2.ADON = 1; //power on adc

   // x position PC1 ADC1_IN11
   // y position PC2 ADC1_IN12
   ADC1->SQR1.L = 0b01; // 2 conversions
   ADC1->SQR3.SQ1 = 11; //first conversion: ADC1_IN12
   ADC1->SQR3.SQ2 = 12; //second conversion: ADC1_IN11

   ADC1->CR1.SCAN = 1; // enable scan mode
   ADC1->CR2.EOCS = 1; // set at end of each regular conversion
}

uint16_t ADC_read(int pin, uint16_t i){
   pinMode(GPIOC, pin, GPIO_ANALOG);
   ADC1->CR2.CONT = 0; //single conversion mode
   ADC1->CR2.SWSTART = 1; //start converison on regular channels
   uint32_t adc[2];

   while(!(ADC1->SR.EOC)); //wait for end of conversion
   adc[0] = ADC1->DR.DATA;

   while(!(ADC1->SR.EOC)); //wait or end of conversion
   adc[1] = ADC1->DR.DATA;
   return adc[i];
}
```

ADC.h

```
// STM32F401RE_ADC.h
// Header for ADC functions
```

```c
#ifndef STM32F4_ADC_H
#define STM32F4_ADC_H

#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"
#include <stdint.h>

////////////////////////////////////////////////////////////////////////////////
// Definitions
////////////////////////////////////////////////////////////////////////////////

#define __IO volatile

// Base addresses
#define ADC1_BASE (0x40012000UL) // base address of ADC1


////////////////////////////////////////////////////////////////////////////////
// Bitfield structs
////////////////////////////////////////////////////////////////////////////////
typedef struct {
   volatile uint32_t AWDCH   : 5;
   volatile uint32_t EOCIE   : 1;
   volatile uint32_t AWDIE   : 1;
   volatile uint32_t JEOCIE  : 1;
   volatile uint32_t SCAN    : 1;
   volatile uint32_t AWDSGL  : 1;
   volatile uint32_t JAUTO   : 1;
   volatile uint32_t DISCEN  : 1;
   volatile uint32_t JDISCEN : 1;
   volatile uint32_t DISCNUM : 3;
   volatile uint32_t         : 6;
   volatile uint32_t JAWDEN  : 1;
   volatile uint32_t AWDEN   : 1;
   volatile uint32_t RES     : 2;
   volatile uint32_t OVRIE   : 1;
   volatile uint32_t         : 5;
} ADC_CR1_bits;

typedef struct {
   volatile uint32_t AWD   : 1;
   volatile uint32_t EOC   : 1;
   volatile uint32_t JEOC  : 1;
   volatile uint32_t JSTRT   : 1;
   volatile uint32_t STRT  : 1;
   volatile uint32_t OVR   : 1;
   volatile uint32_t         : 26;
} ADC_SR_bits;

typedef struct {
   volatile uint32_t SQ13 : 5;
   volatile uint32_t SQ14 : 5;
   volatile uint32_t SQ15 : 5;
   volatile uint32_t SQ16 : 5;
   volatile uint32_t L    : 4;
   volatile uint32_t      : 8;
} SQR1_bits;

typedef struct {
   volatile uint32_t SQ1 : 5;
   volatile uint32_t SQ2 : 5;
   volatile uint32_t SQ3 : 5;
   volatile uint32_t SQ4 : 5;
```

```
   volatile uint32_t SQ5 : 5;
   volatile uint32_t SQ6 : 5;
   volatile uint32_t     : 2;
} SQR3_bits;

typedef struct {
   volatile uint32_t ADON         : 1;
   volatile uint32_t CONT         : 1;
   volatile uint32_t              : 6;
   volatile uint32_t DMA          : 1;
   volatile uint32_t DDS          : 1;
   volatile uint32_t EOCS         : 1;
   volatile uint32_t ALIGN        : 1;
   volatile uint32_t              : 4;
   volatile uint32_t JEXTSEL      : 4;
   volatile uint32_t JEXTEN       : 2;
   volatile uint32_t JSWSTART     : 1;
   volatile uint32_t              : 1;
   volatile uint32_t EXTSEL       : 4;
   volatile uint32_t EXTEN        : 2;
   volatile uint32_t SWSTART      : 1;
   volatile uint32_t              : 1;
} ADC_CR2_bits;

typedef struct {
   volatile uint32_t              : 16;
   volatile uint32_t ADCPRE       : 2;
   volatile uint32_t              : 4;
   volatile uint32_t VBATE        : 1;
   volatile uint32_t TSVREFE      : 1;
   volatile uint32_t              : 8;
} CCR_bits;

typedef struct {
   volatile uint32_t DATA :16;
   volatile uint32_t      :16;
} ADC_DR_bits;

typedef struct {
   __IO ADC_SR_bits SR;
   __IO ADC_CR1_bits CR1;
   __IO ADC_CR2_bits CR2;
   __IO uint32_t SMPR1;
   __IO uint32_t SMPR2;
   __IO uint32_t JOFR1;
   __IO uint32_t JOFR2;
   __IO uint32_t JOFR3;
   __IO uint32_t JOFR4;
   __IO uint32_t HTR;
   __IO uint32_t LTR;
   __IO SQR1_bits SQR1;
   __IO uint32_t SQR2;
   __IO SQR3_bits SQR3;
   __IO uint32_t JSQR;
   __IO uint32_t JDR1;
   __IO uint32_t JDR2;
   __IO uint32_t JDR3;
   __IO uint32_t JDR4;
   __IO ADC_DR_bits DR;
   __IO CCR_bits CCR;
} ADC_TypeDef;

#define ADC1 ((ADC_TypeDef *) ADC1_BASE)
```

```
////////////////////////////////////////////////////////////////////////////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
void ADC_init();
uint16_t ADC_read(int pin, uint16_t i);
#endif
```

Appendix D. Verilog Code

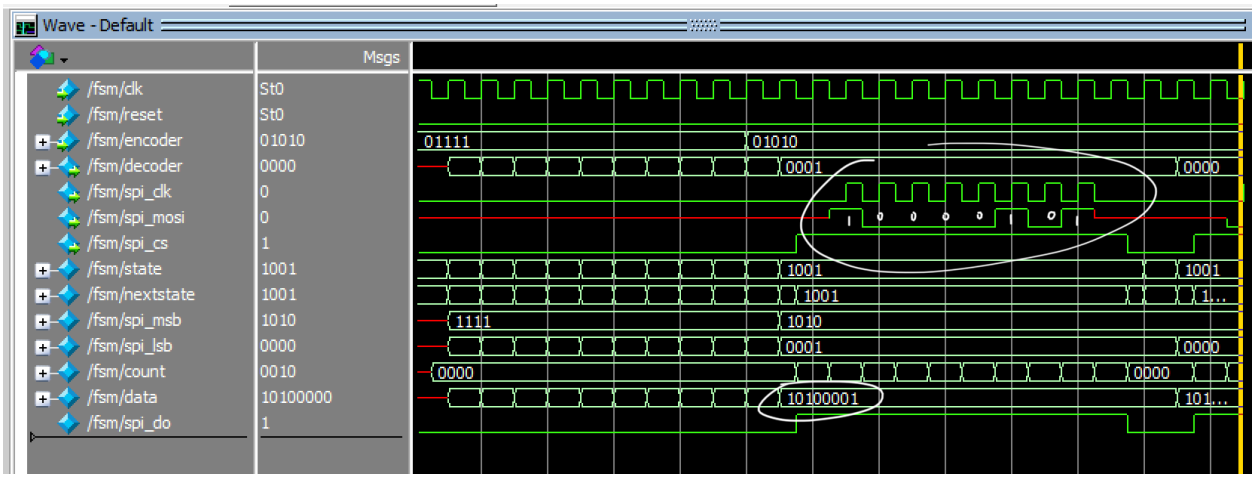
```
module sensor ( input logic clk, reset,
                input logic [4:0] encoder, //EO, A3, A2, A1, A0
                output logic [3:0] decoder,
                output logic spi_clk, spi_mosi, spi_cs // J12, J13, G12
                );
        logic slo_clk;
        logic [13:0] counter; //10k

    always_ff @(posedge clk)
        if(counter == 14'b10000) begin //slow down to 1.2kHz
                counter <= 0;
        slo_clk = ~slo_clk; end
        else counter <= counter+1;


    fsm fsminst(slo_clk, reset, encoder, decoder, spi_clk, spi_mosi, spi_cs);

endmodule

module fsm( input logic clk, reset,
            input logic [4:0] encoder, //EO, A3, A2, A1, A0
            output logic [3:0] decoder,
            output logic spi_clk, spi_mosi, spi_cs
            );
    logic [3:0] state, nextstate, spi_msb, spi_lsb, count;
    logic [7:0] data;
    logic spi_do;

    parameter S0 = 4'd0;
    parameter S1 = 4'd1;
    parameter S2 = 4'd2;
    parameter S3 = 4'd3;
    parameter S4 = 4'd4;
    parameter S5 = 4'd5;
    parameter S6 = 4'd6;
    parameter S7 = 4'd7;
    parameter S8 = 4'd8;
    parameter S9 = 4'd9;



    always_ff @(posedge clk)
```

```
        if(reset) state <= S0;
        else state <= nextstate;

//    always_ff @(posedge clk)
//          if(state == S9 || nextstate == S9) spi_do <= 1;
//          else spi_do <= 0;


    always_comb begin : scanning
        case(state)
            S0: if(encoder == 5'b01111) nextstate = S1;
                else begin                  nextstate = S9;
                end

            S1: if(encoder == 5'b01111) nextstate = S2;
                else begin                  nextstate = S9;
                end
            S2: if(encoder == 5'b01111) nextstate = S3;
                else begin                  nextstate = S9;
                end
            S3: if(encoder == 5'b01111) nextstate = S4;
                else begin                  nextstate = S9;
                end
            S4: if(encoder == 5'b01111) nextstate = S5;
                else begin                  nextstate = S9;
                end
            S5: if(encoder == 5'b01111) nextstate = S6;
                else begin                  nextstate = S9;
                end
            S6: if(encoder == 5'b01111) nextstate = S7;
                else begin                  nextstate = S9;
                end
            S7: if(encoder == 5'b01111) nextstate = S8;
                else begin                  nextstate = S9;
                end
            S8: if(encoder == 5'b01111) nextstate = S0;
                else begin                  nextstate = S9;
                end
            S9: if(~spi_do)                 nextstate = S0;
                else begin                  nextstate = S9;
                end

            default: nextstate = S0;
        endcase
```

```
    end
     //output logic

     assign data = {spi_msb, spi_lsb};


    always_ff @(posedge clk)
        if(state != S9) begin
            spi_msb <= encoder[3:0];
            spi_lsb <= state;
                decoder <= state;
        end

   /// always_ff @(posedge clk)
      //  if(~spi_do  || count != 4'd8 && count != 4'd0)
        //    spi_clk_pos = 1;
       // else
         //   count = 0;

    always_ff @(negedge clk)
        if(spi_do && count <= 4'd9 && count >= 4'd1)
        begin
            spi_mosi = data[count-1];
            count = count + 4'd1;
        end
          else if ((state == S9) && count == 4'd0) count = 4'd1;
        else if(count == 4'd11)
        begin
            count = 4'd0;
        end
          else count = 4'd0;

    assign spi_clk = (count >= 4'd2 && count < 4'd10 && (state == S9)) ? clk: 1'b0;
    assign spi_do = (count <= 4'd10 && count > 4'd0) && (state == S9);
    assign spi_cs = spi_do;


endmodule
```

Appendix E. Misc.

ModelSim Waveforms



Panel Angle vs. Servo Angle Calibration Curves