Wilson Ives and Evan Hassman
Prof. Brake and Prof. Harris
E155 Final Project Report
December 13, 2019

Abstract

For our Microprocessor Systems: Design & Application final project we decided to construct a reaction wheel inverted pendulum. In order to create this system, we needed to implement and utilize additional hardware aside from the provided ATSAM Microcontroller and Cyclone IV E FPGA. Our goals for this project were to have it

1. Powered from a 12V supply
2. Balance indefinitely in an inverted position with no user input
3. Start from a stationary position of at least 5 degrees off vertical and recover to equilibrium

Not only were we able to achieve these goals but we were also able to introduce additional functionality. We were able to have the pendulum stabilize down, as well as swing up from the downwards position to a stable inverted position.

Introduction

Motivation

For our final project we always wanted to create a control system that utilized digital hardware in order to control a mechanical system. We had many initial ideas, but we felt that this project would be the most achievable as it was the easiest to model and had the fewest moving parts compared to our other ideas. That being said, we did know that this project would be quite the undertaking.

Block Diagram

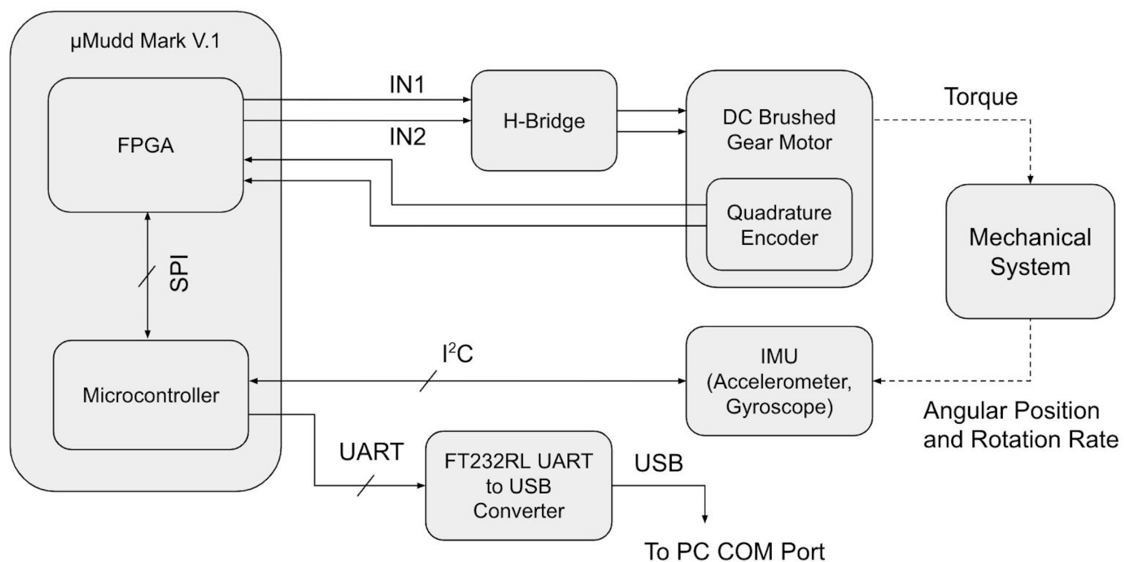


Figure 1: Complete System Block Diagram

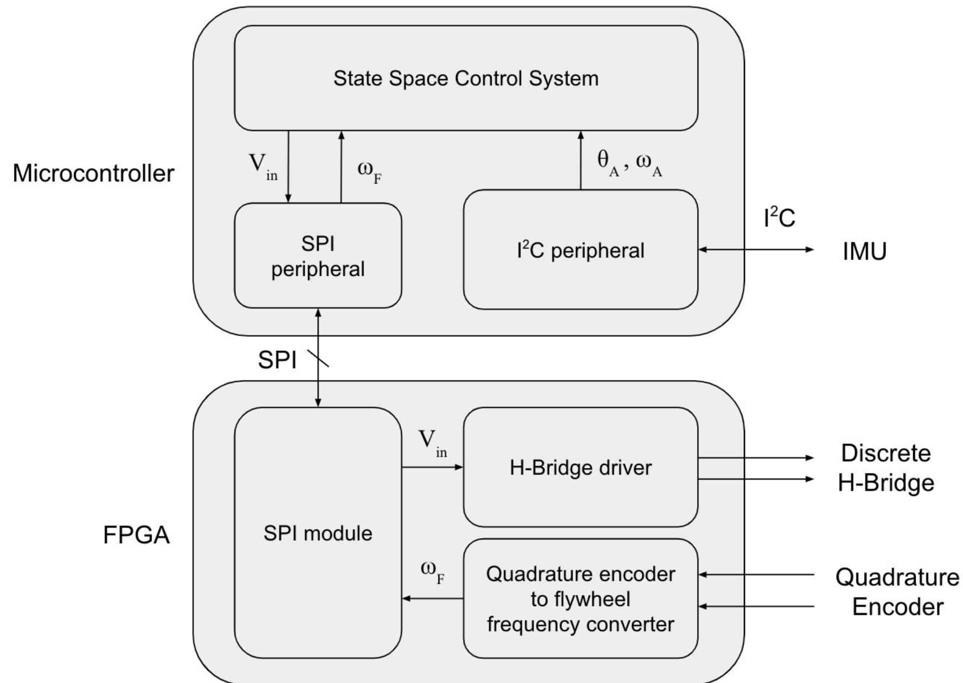


Figure 2: μMudd Mark V.1 Block Diagram

Overview

We first made a simplified model of the reaction wheel pendulum as shown in figures 3 and 4. Three system parameters can be used to fully describe the system state.

- Pendulum arm angular velocity (ω_A)
- Pendulum arm angle (θ_A)
- Flywheel angular velocity (ω_F)

Before constructing the physical system we did extensive MATLAB modeling (see Appendix XXX) to ensure the system was indeed controllable for our choice of motor and general system dimensions. We then optimized the remaining system dimensions to maximize the system's controllability. The techniques used for this optimization are beyond the scope of this class.

Next we designed multiple controllers based on linearized versions of our nonlinear system using a linear quadratic regulator in MATLAB. Finally we designed a swing-up controller that adds energy to the system until the system reaches a region of state space from which the linear controller can bring the system to equilibrium.

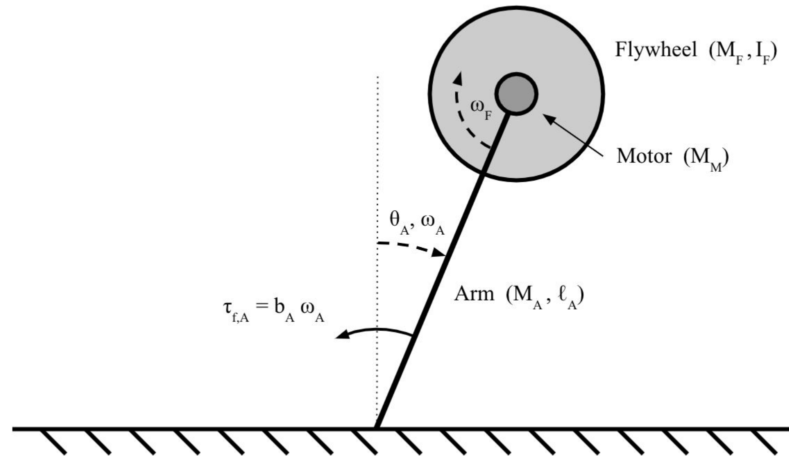


Figure 3: Pendulum Arm

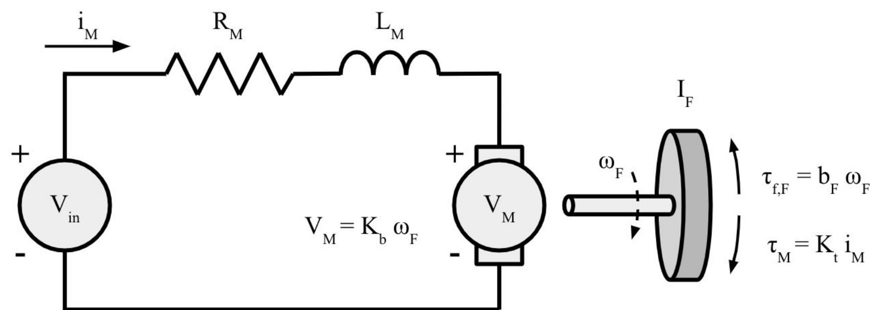


Figure 4: Motor Model

New Hardware

We used a lot of new hardware for this project.

DC Brushed Gear Motor

This motor was used to actuate the flywheel and the integrated encoder was used to measure the flywheel velocity. We selected this motor in conjunction with our flywheel design in order to maximize the system controllability. Unfortunately the weakness of the motor gearbox limited the max torque the motor could output below what the motor itself was capable of providing.

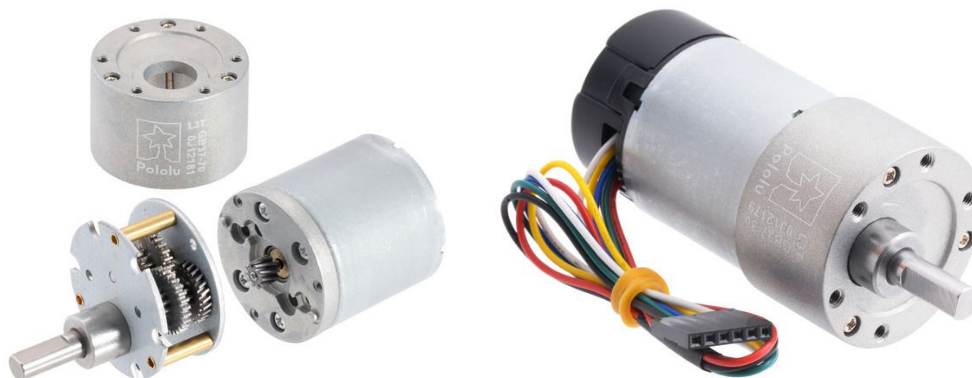


Figure 5: Pololu DC Brushed Gear Motor

- Pololu part number: 4752
- 12V drive
- 30:1 gear ratio
- Quadrature encoder (1920 cts per revolution)

6 Degree of Freedom Inertial Measurement Unit

We used this IMU to measure the other two state variables of the system: the pendulum arm angle and pendulum arm angular velocity. The pendulum arm angular velocity can be determined directly from the gyroscope, while the pendulum arm angle can be calculated from the acceleration due to gravity. To ensure that the only acceleration measured by the accelerometer was from gravity we mounted the imu directly on the axis of rotation so the centripetal and radial terms were zero (as seen in the picture below to the right).

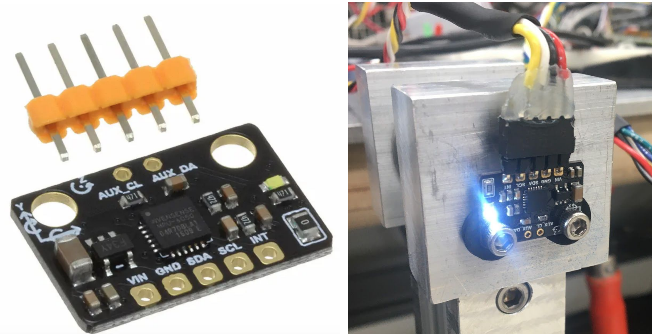


Figure 6: SEN0142 IMU breakout (left), IMU mounted on pendulum arm (right)

- DFRobot part number SEN0142 (Implements the MPU6050 IMU)
- +/-2g full-scale acceleration output (16bit)
- +/-1000 degrees per second full-scale gyroscope output (16bit)
- Up to 400kHz I^{2}C interface

H-bridge

We used a number of h-bridges during the course of this project. Originally we used a discrete h-bridge of our own design for which we could control all four MOSFET gates independently. Controlling such an h-bridge is slightly more complicated because extra care needs to be taken to ensure power is not shorted to ground by turning on both mosfets on one side of the H. We used the FPGA to implement this h-bridge driver. Later in the project we switched to using an integrated h-bridge module which implemented the TLE5206 as shown in figure 7. The h-bridge chip was implemented on a breakout board that was designed for a previous project.

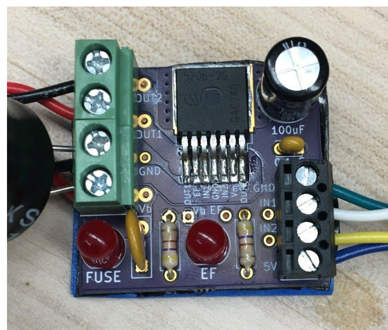


Figure 7: TLE5206 breakout board

- 5A cts output current, 6A peak
- Overcurrent, overvoltage, undervoltage, and short circuit protection (I picked these things for the E80 boards and they are practically indestructible)

UART to USB Converter

This UART for USB converter was incredibly useful for debugging for this project. It allowed us to output serial commands to a COM port on the PC. We also made us of Arduino's serial plotter to plot various values in real time as we manipulated the system.

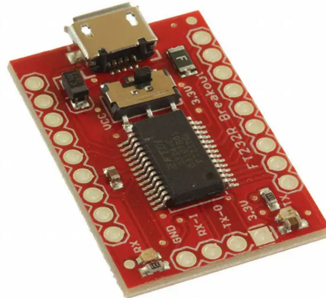


Figure 8: Sparkfun BOB-12731 (Implements the FT232RL)

- Up to 3Mbaud serial transfer rate
- USB 2.0 full speed compatible

Power Supply

The power supply used for this project is a converted server PSU that is capable of outputting 25A at 12V which is plenty for our application. One hitch we encountered was that, like most switch-mode power supplies, our supply was not capable of sinking current. Thus to prevent large voltage spikes when braking the motor we added a constant 6A load to the supply output to we never required the power supply to sink current at any point during the pendulum's operation. A picture of the supply and the power resistors used to dissipate the constant 6A (72W) load is shown below.

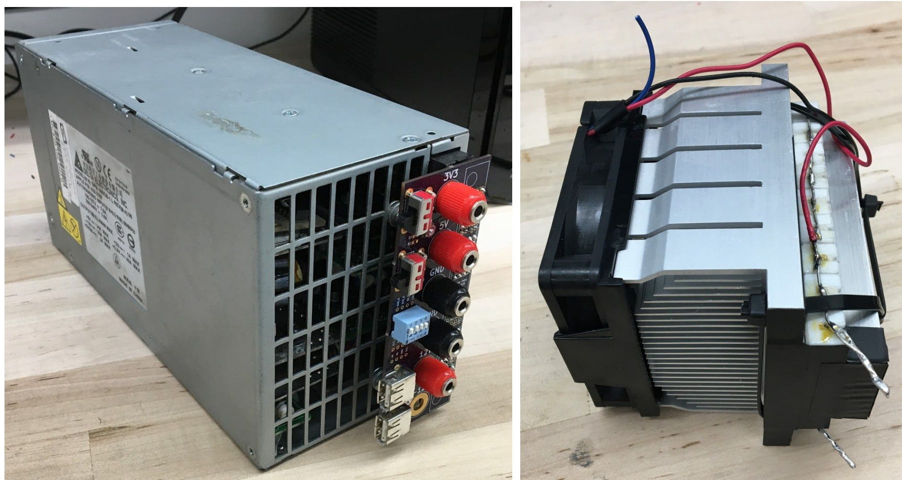
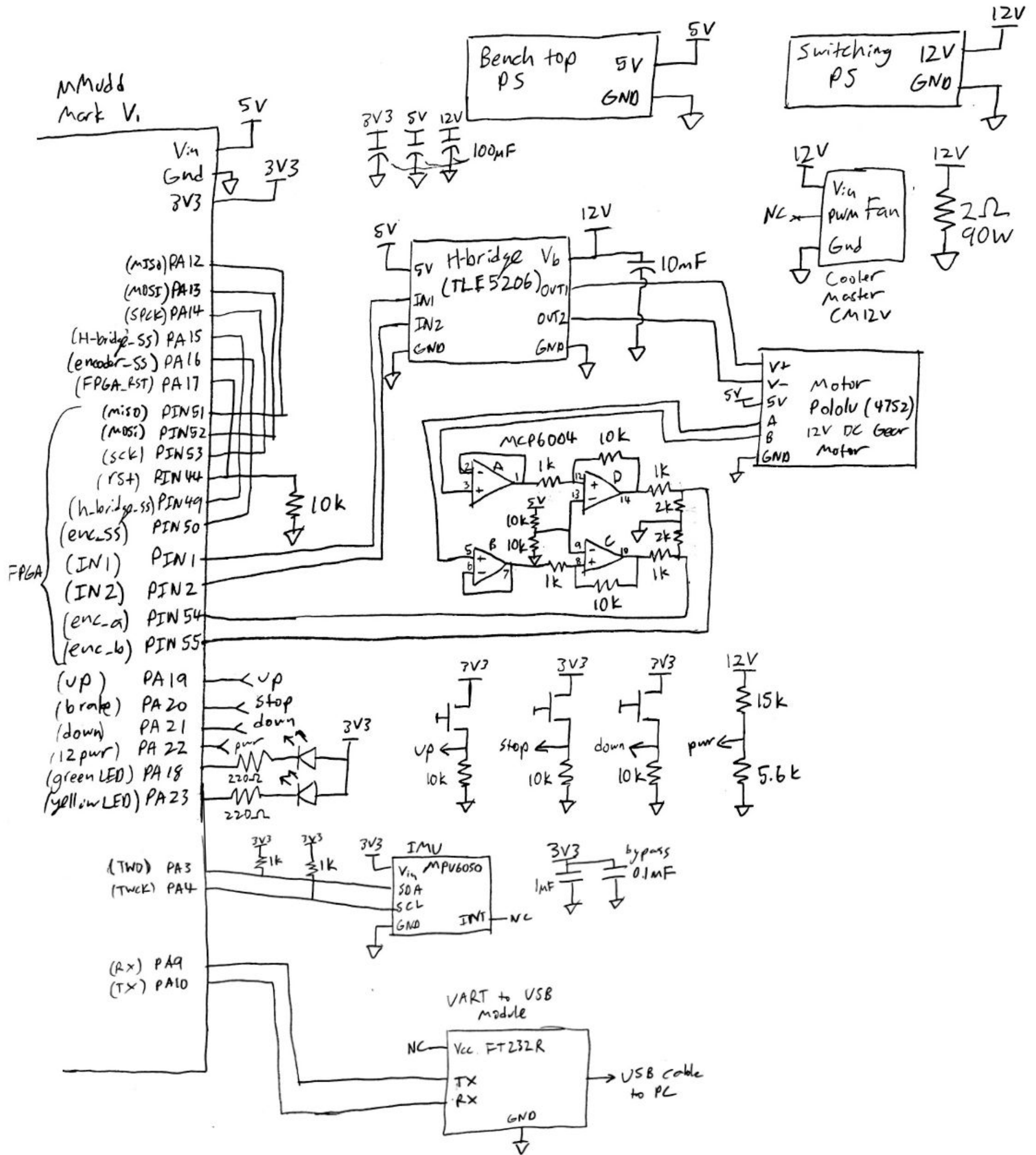


Figure 9: Server PSU (left), 2 Ohm, 90W resistive load attached to CPU heatsink with fan (right)

Schematics



Microcontroller Design

Our Microcontroller was really the brains of our system.

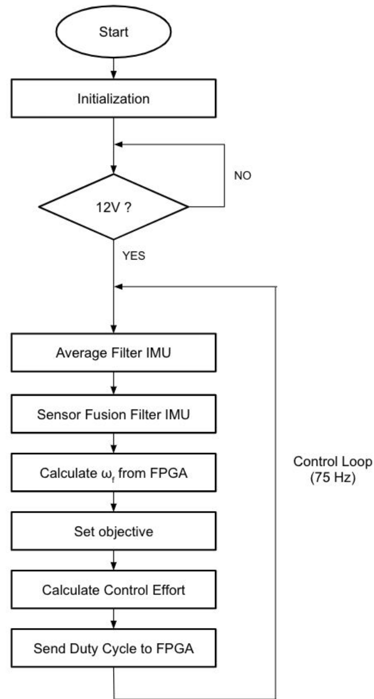


Figure 10: Microcontroller Flow-chart

Initialization

This block includes the initialization of the chip peripherals (Programmable Input/Output, Two Wire Interface, Serial Peripheral Interface, Timer Counter, Universal Asynchronous Receiver/Transmitter) as well as the initialization of many PIO pins, and the configuration our IMU registers, for our desired use, using .

12V Power Loop

This loop was created in order to protect our H-bridge and surrounding hardware. We did not want to attempt to drive our h-bridge when it was not powered, as we believe this might have broken previous motor driver modules. Thus, we wait in this loop until a pin goes high, using a voltage divider, indicating that 12V has been turned on.

Average Filter

Due to the large amounts of noise we had in our IMU data we need to filter it out. We averaged the IMU data with NUM_AVERAGES amount of samples. Currently, NUM_AVERAGES is 8. This filtering helped a little but large deviances still skewed the calculated value. Within this calculation we also added in a calibration offset that we found experimentally.

Sensor Fusion Filter

Using the Arduino serial plotter, as well as MATLAB plotting, we were able to see that the gyroscope's angular velocity data was much less prone to noise compared the acceleration data. Since our sampling is running on a fixed frequency we could integrate our gyroscoping data, ω_A, to determine the change in θ_A since the last sample. We then added a portion of this change to the previous θ_A as well

as a portion of the θ_A calculated from the acceleration vectors. This filter was extremely effective at ultimately allowed for the pendulum to be balanced vertically.

Calculate ω_F

The FPGA was doing the bulk of the work in determining the angular velocity of the flywheel using the motors quadrature encoder and it's programmed hardware. However, for increased precision we bussed over the two integer values that the FPGA found via SPI to then used floating point division to calculate ω_f, along with calibration to convert to units of rad/s.

Set Objective

Our system controller consisted of four objectives, as seen in the FSM below. Our system set it's objective based upon user input as well as a boolean value denoting whether or not the system could be vertically stabilized, calculated based on our state.

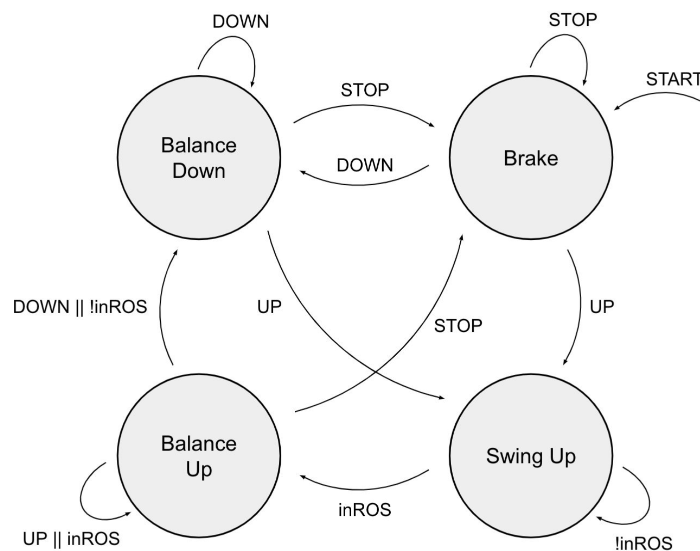


Figure 11: Objective Finite State Machine

Calculate Control Effort

Depending upon the objective of our system we needed to use the appropriate equation to calculate our motor control effort. The control effort is dependent upon the state of the system. The equations calculating these control efforts were found using MATLAB simulation and are outside the scope of this class.

Send Duty Cycle to FPGA

Using the previously calculated floating point control effort, we found the corresponding PWM duty cycle as a sign magnitude 12-bit value. This 12-bit value was then bussed to the FPGA using the SPI peripheral. It was then used to drive the h-bridge and our motor.

FPGA Design

The functionality implemented on our FPGA can be split into three main modules: an SPI slave interface, an h-bridge driver, and a quadrature decoder. In the following section each module will be described.

SPI Slave

The SPI slave module was based on the SPI module example from the E85 textbook. The implied hardware is shown in figure 11 below. We actually implemented two copies of the hardware below. One spi slave only received the most recently calculated control effort from the microcontroller and returned nothing (at least no meaningful information). The second spi slave only returned the most recently calculated values from the quadrature decoder module.

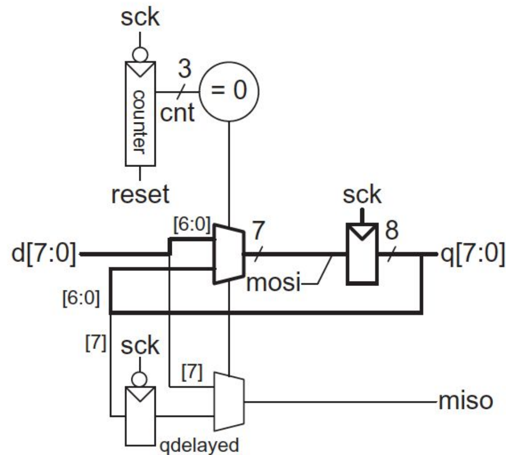


Figure 11: SPI slave as implemented in verilog

H-bridge Driver

The h-bridge driver module receives a 12 bit sign magnitude control effort from the corresponding spi slave. The most significant bit of the 12 bit control effort determines the direction of motor torque. The other 11 bits control the PWM duty cycle. In this way the h-bridge driver can precisely control the output average voltage seen by the motor. This module had initially been designed to drive the gates of 4 independent mosfets but late in the project we switched to using an integrated module which only required a direction and PWM duty cycle control. This functionality could be easily implemented directly on the microcontroller but since our spi interface was already set up and working, we decided to keep this functionality on the FPGA.

Quadrature Decoder

This module was the meat of the functionality on the FPGA. The module implemented a very large FSM and worked by counting signed encoder state changes in a 16 bit value called *increment*. The *increment* value is updated throughout a fixed sample period that determines the update rate of the spi slave that is sampled by the microcontroller. The module also keeps track of the number of clock cycles that occur between the first and last encoder state change during the sample in a value called *counts*. Both these values are sent to a holding register in the quadrature decoder spi slave module. Once received by the microcontroller, both values are cast to floats and used to calculate the angular speed of the flywheel. This functionality is also depicted in figure 12.

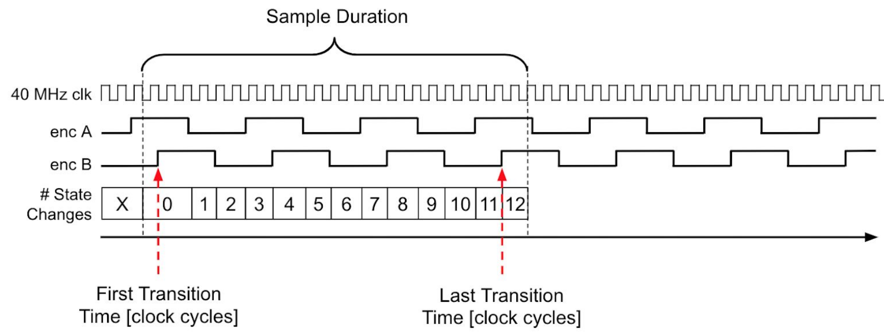


Figure 12: quadrature decoder module functionality

Results

We were not only able to get the pendulum to recover from a +/- 5 degree deviation from vertical but also get the pendulum to swing up from rest. The three plots below show experimental data from a single trial in which we first instructed the pendulum to swing up automatically (figure 13). We then proceeded to hit the pendulum so it deviated from vertical and recovered back to equilibrium (figure 14). As can be seen the pendulum was able to recover from deviations larger than 5 degrees as promised. Finally, we pushed the pendulum out of the region it could recover from. At this point the pendulum switched objectives to balancing downwards (figure 15).

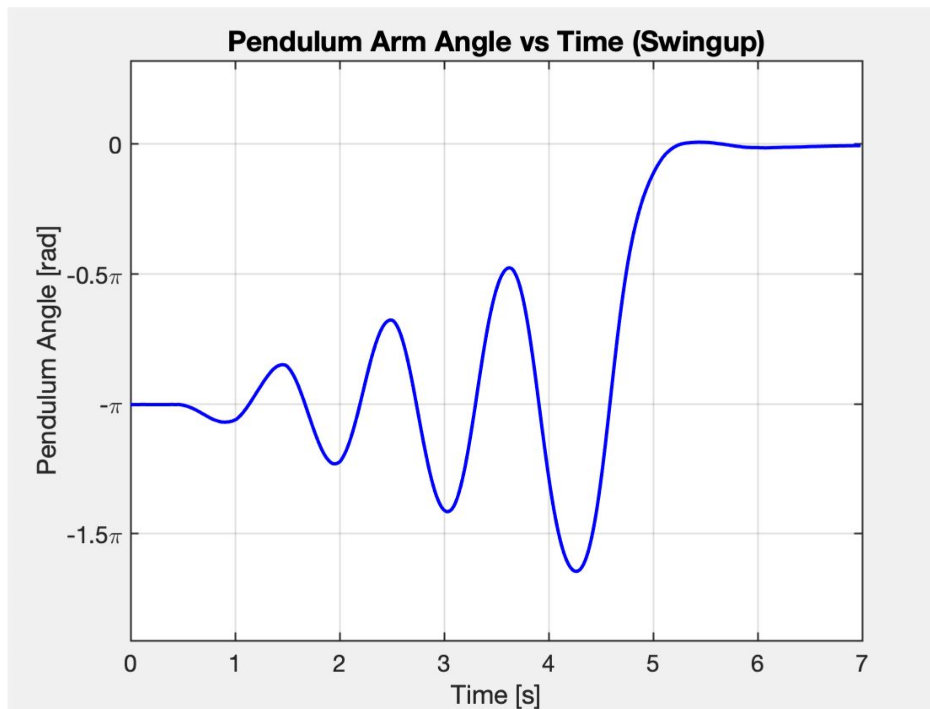


Figure 13: Experimental pendulum swing-up

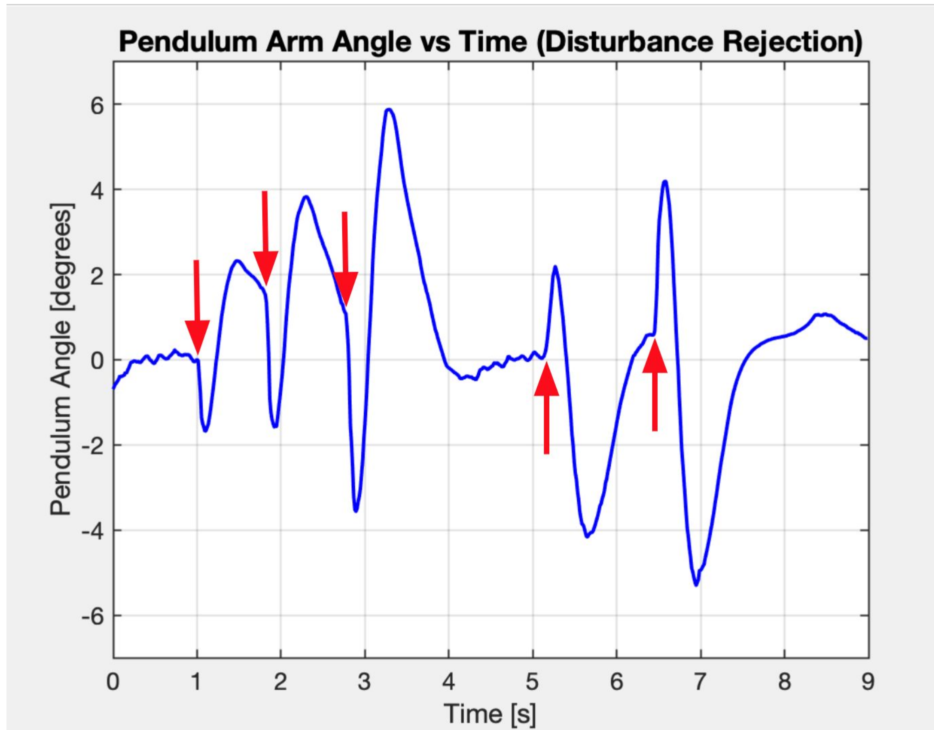


Figure 14: Experimental inverted balancing with disturbance rejection. The red arrows indicate when and in what direction external disturbances were applied during the trial.

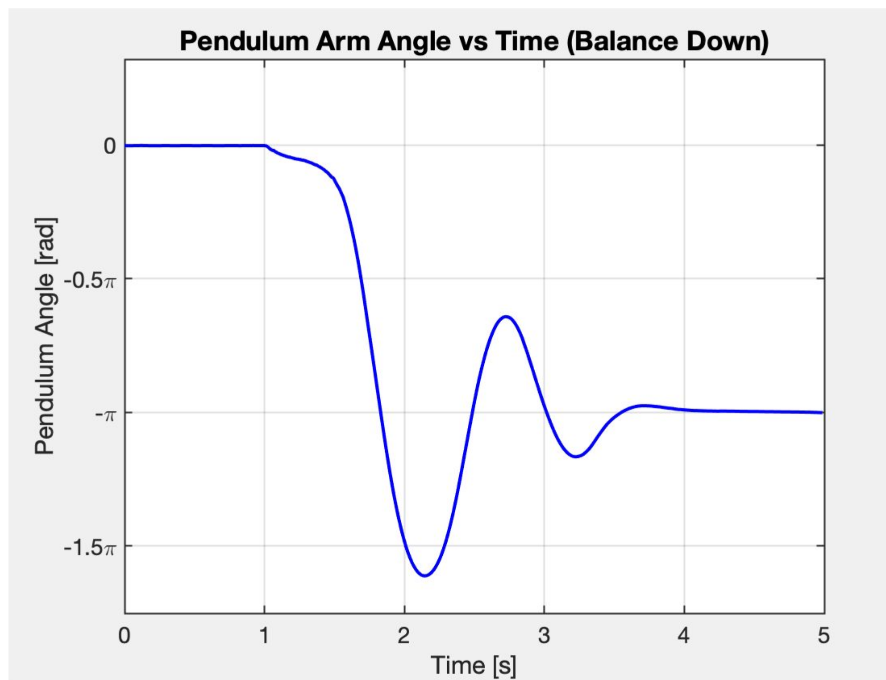


Figure 15: Experimental downwards balance. As can be seen the pendulum balances downwards much more quickly than it swings up.

References

1] Belascuen, G. and Aguilar, N. (2018). Design, Modeling and Control of a Reaction Wheel Balanced Inverted Pendulum. *2018 IEEE Biennial Congress of Argentina (ARGENCON)*.
[2] F. Jepsen, A. Soborg, A. Pedersen and Z. Yang, "Development and control of an inverted pendulum driven by a reaction wheel", *2009 International Conference on Mechatronics and Automation*, 2009. Available: 10.1109/icma.2009.5246460 [Accessed 30 October 2019].
[3] Harris and Harris, E85 textbook
[4] Pololu Motor datasheet: https://www.pololu.com/file/0J1706/pololu-37d-metal-gearmotors.pdf
[5] Sparkfun UART to USB guide: https://www.sparkfun.com/products/12731

Bill of Materials

Item	Purpose	Vendor	Vendor PN	Cost [$]
IMU	state estimation	Digikey	1738-1070-ND	$10.20
DC gear motor	drive flywheel	Pololu	4752	$39.95
bearings (x2)	secure armature to axle	McMaster Carr	6383K234	$19.64
uart to USB	Serial Debugging	Digikey	1568-1195-ND	$15.95
H-bridge	drive motor	previous project	N/A	N/A
12V server PSU	power system	HMC electronics recycling bin	N/A	N/A
Scrap Aluminum	machining all the parts on the mechanical system	makerspace	N/A	N/A
Total				**$75.54**

Appendix Table of Contents

1. Software
    a. Final.c
    b. Controller.h
    c. MPU6050.h
    d. SAM4S4B_twi.h
    e. FPGA_spi.h
    f. Serial.h
2. Verilog
3. Matlab Code
    a. Flywheel_pendulum_swingup.m
    b. Region_of_attraction.m
    c. State_to_video.m
    d. calculate _parameters.m
    e. csv_reader.m
4. Machining

Appendix 1: Software

Appendix 1a: Final.c

```
/*
File:   final.c
Author: Evan Hassman
        Wilson Ives
Email:  ehassman@g.hmc.edu
        wives@g.hmc.edu
Date:   November 2019

Description:
                             Main C code to run on ATSAM4S4B to control inverted pendulum
*/

#include "SAM4S4B/SAM4S4B.h"
#include "MPU6050/MPU6050.h"
#include "Serial/Serial.h"
#include "FPGA/FPGA_spi.h"
#include "math.h"
#include "stdbool.h"
#include "Serial/Serial.h"
#include "Controller/Controller.h"

// IMU Calibration
#define G_Z_OFFSET 0.016
#define A_X_OFFSET -0.026
#define A_Y_OFFSET -0.012
#define THETA_A_OFFSET -.0203

#define V12_PWR_PIN       PIO_PA22
#define TIME_PIN          PIO_PA27
#define GREEN_LED_PIN     PIO_PA18
#define YELLOW_LED_PIN    PIO_PA23

// Data Storage
#define SAMPLE_FREQ 75.0 // sample frequency in Hz
#define NUM_SAMPLES 2000 // number of samples to store in buffer
#define NUM_POINTS  4 // number of values to store every sample cycle

float data_buff[NUM_SAMPLES][NUM_POINTS];  // holds state and control effort for every time
step


/////////////////////////////////////////////////////////////////
// Helper Function
/////////////////////////////////////////////////////////////////

/** Print 2D array in CSV format
 *
 * Function uses serial print functions to print out data_buff
 * in a CSV format to be used in MATLAB, Excel, or other
 * post processing applications
 */
void serialPrintCSV() {
    serialPrintln("-------------------------------------");
    serialPrintln("--------Beginning of CSV file--------");
```

```
    serialPrintln("--------------------------------------");

    for (int i = 0; i < NUM_SAMPLES; i++) {
      for (int j = 0; j < NUM_POINTS; j++) {
        serialPrintFloat(data_buff[i][j], 5);
        if (j != (NUM_POINTS-1))
          serialPrint(",");
      }
      serialPrintln("");
    }
    serialPrintln("--------------------------------------");
    serialPrintln("-----------End of CSV file-----------");
    serialPrintln("--------------------------------------");
}

/////////////////////////////////////////////////////////////////
// Main
/////////////////////////////////////////////////////////////////

int main (void) {
    ///////////////////////////////////////
    // Initialization
    ///////////////////////////////////////
    samInit();
              pioInit();
              tc0DelayInit();
    uartInit(SERIAL_PARITY,SERIAL_BAUD_RATE_DIV);  // in order to print to serial

    // IMU initialization and TWI, I2C, peripheral initialization
              twiInit(MPU6050_DEFAULT_ADDRESS);  // must be initialized before IMU
    imuInit();

    // FPGA SPI pin initialization and SPI peripheral initialization
              // "clock divide" = master clock frequency / desired baud rate
              // the phase for the SPI clock is 1 and the polarity is 0
    spiInit(MCK_FREQ/1000000, 0, 1); // clock speed can be way higher
    FPGA_spiInit();

    controllerInit();  // initalizae PIO pins for objective selecting

    // PIO Initialization
    pioPinMode(V12_PWR_PIN, PIO_INPUT);
    pioPinMode(GREEN_LED_PIN, PIO_OUTPUT);
    pioPinMode(YELLOW_LED_PIN, PIO_OUTPUT);

    // Variable Definitions
    float imuData[3];   // [Ax, Ay, Gz] Acceleration in g's, angular velocity in
degrees/second
    float sums[3];      // [ax_sum, ay_sum, az_sum] used in running average of imu data
    float state[3];     // [w_a, theta_a, w_f]
    uint16_t u = 0;     // Control Effort, sign-magnitude duty cycle
    float u_float = 0;  // Control Effort, float [V]
    float theta_a;      // Pendulum Angle as calculated from only accelerometer samples
    bool first_sample = true; // true only on the first sample

    // T_sample_count is the number of clock cycles in a given sample period
    uint32_t T_sample_count = (uint32_t)((1/SAMPLE_FREQ) * TC_CLK3_SPEED);
```

```
    int objective = 0; // 0: brake, -1: balance down, 1: swing up, 2: balance up

    // indicate the controller is waiting for 12V supply to power on
    pioDigitalWrite(GREEN_LED_PIN, PIO_HIGH); // green LED off
    pioDigitalWrite(YELLOW_LED_PIN, PIO_LOW); // yellow LED on

    // Protection of H-Bridge
    sendControlEffort(&u);
    while(pioDigitalRead(V12_PWR_PIN) != PIO_HIGH) {};

    // indicate 12V supply is powered on
    pioDigitalWrite(YELLOW_LED_PIN, PIO_HIGH); // yellow LED off
    pioDigitalWrite(GREEN_LED_PIN, PIO_LOW); // green LED on

    /////////////////////////////////////////
    // Balance control loop
    /////////////////////////////////////////
    for (int i = 0; i < NUM_SAMPLES; i++) {
      // timestamp the sample to ensure a steady sample period
      TC0->TC_CH[0].TC_CCR.SWTRG = 1; // Reset counter
      TC0->TC_CH[0].TC_RC = T_sample_count; // Set compare value

      averageFilter(sums, imuData);

      // Calculate w_a
      state[0] = -PI/180*(sums[2]/NUM_AVERAGES) + G_Z_OFFSET;

      // Calculate theta_a using only accelerometer samples
      theta_a = -atan2((sums[0]/NUM_AVERAGES)+A_X_OFFSET, (sums[1]/NUM_AVERAGES)+A_Y_OFFSET) +
THETA_A_OFFSET;

      if (first_sample) {  // first sample only use the accelerometer measurement to set
theta_a
        state[1] = theta_a;
        first_sample = 0;
      }
      else {
        sensorFusionFilter(state, &theta_a, SAMPLE_FREQ);  // Otherwise use sensor fusion to
set theta_a
      }

      receiveQuadratureEncoderData(&state[2]);
      setObjective(&objective, state);
      control(&objective, state, &u_float, &u);

      //serialPrintFloat(state[1]*180.0/PI, 5);
      //serialPrintln("");

      // serialPrintStateControlPlotter(state, &u_float);

      data_buff[i][0] = state[0];
      data_buff[i][1] = state[1];
      data_buff[i][2] = state[2];
      data_buff[i][3] = u_float;

      while(!(TC0->TC_CH[0].TC_SR.CPCS)); // Wait until an RC Compare has occurred
```

```
    }
    serialPrintCSV();
    while(1);

    while(1) {
      // timestamp the sample to ensure a steady sample period
      TC0->TC_CH[0].TC_CCR.SWTRG = 1; // Reset counter
      TC0->TC_CH[0].TC_RC = T_sample_count; // Set compare value

      averageFilter(sums, imuData);

      // Calculate w_a
      state[0] = -PI/180*(sums[2]/NUM_AVERAGES) + G_Z_OFFSET;

      // Calculate theta_a using only accelerometer samples
      theta_a = -atan2((sums[0]/NUM_AVERAGES)+A_X_OFFSET, (sums[1]/NUM_AVERAGES)+A_Y_OFFSET);

      if (first_sample) {  // first sample only use the accelerometer measurement to set
theta_a
        state[1] = theta_a;
        first_sample = 0;
      }
      else {
        sensorFusionFilter(state, &theta_a, SAMPLE_FREQ);  // Otherwise use sensor fusion to
set theta_a
      }

      receiveQuadratureEncoderData(&state[2]);
      setObjective(&objective, state);
      control(&objective, state, &u_float, &u);

      //serialPrintFloat(state[1]*180.0/PI, 5);
      //serialPrintln("");

      serialPrintStateControlPlotter(state, &u_float);

      while(!(TC0->TC_CH[0].TC_SR.CPCS)); // Wait until an RC Compare has occurred
    }

}
```

Appendix 1b: Controller.h

```
/*
File:   Controller.h
Author: Evan Hassman
        Wilson Ives
Email:  ehassman@g.hmc.edu
        wives@g.hmc.edu
Date:   November 2019

Description:

*/

#ifndef CONTROLLER_H
#define CONTROLLER_H
```

```c
#include "../SAM4S4B/SAM4S4B_pio.h"
#include "stdbool.h"
#include "math.h"

// Motor Characteristics
#define I_MAX_BALANCE_UP 6
#define I_MAX_SWING_UP 2.2
#define I_MAX_BALANCE_DOWN 4
#define V_SAT 12
#define K_B 0.3472
#define R_M 2.2

// Physical System Characteristics
#define I_F 0.00475   // Inertia of the flywheel
#define I_0 0.0657    // Inertia of arm
#define TAU_G 2.468   // Torque due to gravity on pendulum arm

// Determined via MATLAB simulation
// Stabilize Upwards
#define K1_UP 17.667
#define K2_UP 106.0433
#define K3_UP 0.7085

// Stabilize Downwards
#define K1_DOWN 6.9494
#define K2_DOWN 40.1930
#define K3_DOWN 0.1132

// ROS variables
#define A -11.7
#define SLOPE 0.0286
#define WIDTH_SU 0.6 //swing up ROS width
#define WIDTH_BU 0.9 //balance up ROS width

// Filtering
#define NUM_AVERAGES 8
#define SENSOR_FUSION 0.98

#define BALANCE_UP_PIN    PIO_PA19
#define MOTOR_OFF_PIN     PIO_PA20
#define BALANCE_DOWN_PIN  PIO_PA21

#define PI 3.1415926535897932384626433

/** Initialize Control Pins
 *
 */
void controllerInit() {
    pioPinMode(MOTOR_OFF_PIN, PIO_INPUT);  // slows motor to stop, when high
    pioPinMode(BALANCE_DOWN_PIN, PIO_INPUT);
    pioPinMode(BALANCE_UP_PIN, PIO_INPUT);
}

/** Set saturation control limits
 *
 * Limits control effort u_float [v] to be bounded by voltage
```

```
 * saturation limit as well as current saturation limit, upper
 * and lower, as calculated based on I_MAX...
 */
void saturationLimit(float* u_float, float upper, float lower) {
    // Voltage Saturation Limit
    if (*u_float > V_SAT)
      *u_float = V_SAT;
    else if (*u_float < -V_SAT)
      *u_float = -V_SAT;

    // Current Saturation limit
    if (*u_float > upper)
      *u_float = upper;
    else if (*u_float < lower)
      *u_float = lower;
}

/* Determine if state is within region of stability
 * Determines whether the system can be controlled to
 * equilibrium in the upwards position using the
 * linear controller
 */
bool controllable(int* objective, float* state) {
    if (*objective == 2)
      return ((state[1] > -0.349) & (state[1] < 0.349)) || // +/- 20 degrees
             ((state[0] < A*sin(0.5*state[1]) - SLOPE*state[2] + WIDTH_BU) &
              (state[0] > A*sin(0.5*state[1]) - SLOPE*state[2] - WIDTH_BU));
    else
      return (state[0] < A*sin(0.5*state[1]) - SLOPE*state[2] + WIDTH_SU) &
             (state[0] > A*sin(0.5*state[1]) - SLOPE*state[2] - WIDTH_SU) &
             (state[1] > -1) & (state[1] < 1);
}

/* Calculates the energy of the pendulum arm based on the state [J]
 */
void arm_energy(float* state,float* E_a) {
  *E_a = 0.5*I_0*state[0]*state[0] + // kinetic energy
         TAU_G*(1+cos(state[1]));    // potential energy
}

/* Calculates the energy of the flywheel based on the state [J]
 */
void flywheel_energy(float* state, float* E_f) {
  *E_f = 0.5*I_F*(state[0]+state[2])*(state[0]+state[2]);
}

/* Determine if the system has sufficient energy to reach equilibrium
 */
bool sufficientEnergy(int* objective, float* state) {
  float E_f = 0; // flywheel energy
  float E_a = 0; // pendulum arm energy
  arm_energy(state, &E_a);
  flywheel_energy(state, &E_f);

  if (*objective == 1)
    if(state[0]>0)
      return (E_a > (2*TAU_G - E_f));
```

```
    else
      return (E_a > (2*TAU_G + E_f));
  else
    return 0;
}

/** Sets the value of objective based on button inputs
 */
void setObjective(int* objective, float* state) {
    bool inROS = controllable(objective, state);

    if (pioDigitalRead(MOTOR_OFF_PIN))
      *objective = 0;
    else if (pioDigitalRead(BALANCE_DOWN_PIN) || (*objective == 2 && !inROS))
      *objective = -1;
    else if (pioDigitalRead(BALANCE_UP_PIN) && !inROS)
      *objective = 1;
    else if ((*objective == 1 || pioDigitalRead(BALANCE_UP_PIN)) && inROS)
      *objective = 2;
}

/* Set Duty Cycle
 * Converts control effort u_float [V] to
 * sign magnitude 12-bit duty cycle
 */
void setDutyCycle(uint16_t* u, float* u_float) {
    // Convert float to 12-bit sign-magnitude duty cycle
    uint16_t magnitudebits;
    uint8_t signbit;
    if (*u_float < 0) {
      magnitudebits = (int16_t) ((*u_float) * -2047.0/12);
      signbit = 1;
    }
    else {
      magnitudebits = (int16_t) ((*u_float) * 2047.0/12);
      signbit = 0;
    }
    *u = (signbit << 11) | magnitudebits;
}

/* Calculate Control effort
 * Calculate control effort u[V] when trying to balance
 * upwards. Control limit is also bounded by voltage
 * and current saturation limits
 */
void getControlEffortBalanceUp(float* u_float, float* state) {
    *u_float = K1_UP*(state[0]) + K2_UP*(state[1]) + K3_UP*(state[2]);

    // Saturation Limit
    float upper = (state[2])*K_B + I_MAX_BALANCE_UP*R_M;
    float lower = (state[2])*K_B - I_MAX_BALANCE_UP*R_M;
    saturationLimit(u_float, upper, lower);
}

/* Calculate Control effort
 * Calculate control effort u[V] when trying to balance
 * downwards. Control limit is also bounded by voltage
```

```
 * and current saturation limits
 */
void getControlEffortBalanceDown(float* u_float, float* state) {
    if(state[1] > 0)
      *u_float = K1_DOWN*(state[0]) + K2_DOWN*(state[1]-PI) + K3_DOWN*(state[2]);
    else
      *u_float = K1_DOWN*(state[0]) + K2_DOWN*(state[1]+PI) + K3_DOWN*(state[2]);

    // Saturation Limit
    float upper = (state[2])*K_B + I_MAX_BALANCE_UP*R_M;
    float lower = (state[2])*K_B - I_MAX_BALANCE_UP*R_M;
    saturationLimit(u_float, upper, lower);
}

/* Calculate Control effort
 * Calculate control effort u[V] when trying to swing up
 * pendulum. Control limit is also bounded by voltage
 * and current saturation limits
 */
void getControlEffortSwingUp(float* u_float, float* state) {
    if (state[0] > 0.2)
      *u_float = -12;
    else
      *u_float = 12;

    // Saturation Limit
    float upper = (state[2])*K_B + I_MAX_SWING_UP*R_M;
    float lower = (state[2])*K_B - I_MAX_SWING_UP*R_M;
    saturationLimit(u_float, upper, lower);
}

/* Calculate Control effort
 * sets control effort u[V] to 0.
 * Control limit is also bounded by voltage
 * and current saturation limits
 */
void getControlEffortBrake(float* u_float, float* state) {
    *u_float = 0;
    float upper = (state[2])*K_B + I_MAX_SWING_UP*R_M;
    float lower = (state[2])*K_B - I_MAX_SWING_UP*R_M;
    saturationLimit(u_float, upper, lower);
}

/* Calculate Control effort
 * sets control effort u[V] to set motor current to zero
 */
void getControlEffortCoast(float* u_float, float* state) {
    *u_float = 0;
    float upper = (state[2])*K_B;
    float lower = (state[2])*K_B;
    saturationLimit(u_float, upper, lower);
}

/* Constrain Angle Sensor Fusion
 * constrains angle between -pi and pi
 * following sensor fusion algorithm
 */
```

```
void constrainAngle(float* theta){
    while(*theta > PI)
      *theta = *theta - 2*PI;
    while(*theta < -PI)
      *theta = *theta + 2*PI;
}

/* Apply Sensor Fusion
 * Calculate theta_a using integration of w_a and
 * sensor fusion factor
 */
void sensorFusionFilter(float* state, float* theta, float frequency) {
    state[1] = state[1] + state[0]*(1/frequency)*SENSOR_FUSION;

    if(state[1] > ((*theta) + PI))
      state[1] += ((*theta)-state[1]+2*PI)*(1-SENSOR_FUSION);
    else if (state[1] < ((*theta) - PI))
      state[1] += ((*theta)-state[1]-2*PI)*(1-SENSOR_FUSION);
    else
      state[1] += ((*theta)-state[1])*(1-SENSOR_FUSION);

    constrainAngle(&state[1]);
}

void averageFilter(float* sums, float* imuData) {
    // Reset Average Filter
    sums[0] = 0;
    sums[1] = 0;
    sums[2] = 0;

    // Sample from the IMU NUM_AVERAGES times
    for (int i = 0; i < NUM_AVERAGES; i++) {
      imuFastSample(imuData);
      sums[0] += imuData[0];
      sums[1] += imuData[1];
      sums[2] += imuData[2];
    }
}

/* Control system based on objective
 */
void control(int* objective, float* state, float* u_float, uint16_t* u) {
    // Calculate control effort [V] based on state and the current stabilization scheme
    if (*objective == 0)
      getControlEffortBrake(u_float, state);
    else if (*objective == -1)
      getControlEffortBalanceDown(u_float, state);
    else if (*objective == 1)
      getControlEffortSwingUp(u_float, state);
    else if (*objective == 2)
      getControlEffortBalanceUp(u_float, state);
    else  // should not happen
      getControlEffortBrake(u_float, state);

    // Convert float control effort to sign magnitude duty cycle and send via SPI
    setDutyCycle(u, u_float);
    sendControlEffort(u);
```

```
}
#endif
```

Appendix 1c: MPU6050.h

```
/*
File:   MPU6050.c
Author: Evan Hassman
Email:  ehassman@g.hmc.edu
Date:   November 2019

Description:
    Contains register address locations, definitions,
    and functions for the MPU6050 IMU
*/


////////////////////////////////////////////
// Original Notes
////////////////////////////////////////////

// I2Cdev library collection - MPU6050 I2C device class
// Based on InvenSense MPU-6050 register map document rev. 2.0, 5/19/2011 (RM-MPU-6000A-00)
// 10/3/2011 by Jeff Rowberg <jeff@rowberg.net>
// Updates should (hopefully) always be available at https://github.com/jrowberg/i2cdevlib
//
// Changelog:
//     ... - ongoing debug release

// NOTE: THIS IS ONLY A PARIAL RELEASE. THIS DEVICE CLASS IS CURRENTLY UNDERGOING ACTIVE
// DEVELOPMENT AND IS STILL MISSING SOME IMPORTANT FEATURES. PLEASE KEEP THIS IN MIND IF
// YOU DECIDE TO USE THIS PARTICULAR CODE FOR ANYTHING.

/* ============================================
I2Cdev device library code is placed under the MIT license
Copyright (c) 2012 Jeff Rowberg

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
=============================================== */

#ifndef MPU6050_H
#define MPU6050_H

#include "../SAM4S4B/SAM4S4B_twi.h"
#include "../Serial/Serial.h"
```

```c
#include "stdio.h"
#include "math.h"


//////////////////////////////////////////////
// Register Memory Definitions
//////////////////////////////////////////////

#define MPU6050_ADDRESS_AD0_LOW         0x68 // address pin low (GND), default for InvenSense
evaluation board
#define MPU6050_ADDRESS_AD0_HIGH        0x69 // address pin high (VCC)
#define MPU6050_DEFAULT_ADDRESS         MPU6050_ADDRESS_AD0_LOW

#define MPU6050_RA_XG_OFFS_TC           0x00 //[7] PWR_MODE, [6:1] XG_OFFS_TC, [0] OTP_BNK_VLD
#define MPU6050_RA_YG_OFFS_TC           0x01 //[7] PWR_MODE, [6:1] YG_OFFS_TC, [0] OTP_BNK_VLD
#define MPU6050_RA_ZG_OFFS_TC           0x02 //[7] PWR_MODE, [6:1] ZG_OFFS_TC, [0] OTP_BNK_VLD
#define MPU6050_RA_X_FINE_GAIN          0x03 //[7:0] X_FINE_GAIN
#define MPU6050_RA_Y_FINE_GAIN          0x04 //[7:0] Y_FINE_GAIN
#define MPU6050_RA_Z_FINE_GAIN          0x05 //[7:0] Z_FINE_GAIN
#define MPU6050_RA_XA_OFFS_H            0x06 //[15:0] XA_OFFS
#define MPU6050_RA_XA_OFFS_L_TC         0x07
#define MPU6050_RA_YA_OFFS_H            0x08 //[15:0] YA_OFFS
#define MPU6050_RA_YA_OFFS_L_TC         0x09
#define MPU6050_RA_ZA_OFFS_H            0x0A //[15:0] ZA_OFFS
#define MPU6050_RA_ZA_OFFS_L_TC         0x0B
#define MPU6050_RA_XG_OFFS_USRH         0x13 //[15:0] XG_OFFS_USR
#define MPU6050_RA_XG_OFFS_USRL         0x14
#define MPU6050_RA_YG_OFFS_USRH         0x15 //[15:0] YG_OFFS_USR
#define MPU6050_RA_YG_OFFS_USRL         0x16
#define MPU6050_RA_ZG_OFFS_USRH         0x17 //[15:0] ZG_OFFS_USR
#define MPU6050_RA_ZG_OFFS_USRL         0x18
#define MPU6050_RA_SMPLRT_DIV           0x19
#define MPU6050_RA_CONFIG               0x1A
#define MPU6050_RA_GYRO_CONFIG          0x1B
#define MPU6050_RA_ACCEL_CONFIG         0x1C
#define MPU6050_RA_FF_THR               0x1D
#define MPU6050_RA_FF_DUR               0x1E
#define MPU6050_RA_MOT_THR              0x1F
#define MPU6050_RA_MOT_DUR              0x20
#define MPU6050_RA_ZRMOT_THR            0x21
#define MPU6050_RA_ZRMOT_DUR            0x22
#define MPU6050_RA_FIFO_EN              0x23
#define MPU6050_RA_I2C_MST_CTRL         0x24
#define MPU6050_RA_I2C_SLV0_ADDR        0x25
#define MPU6050_RA_I2C_SLV0_REG         0x26
#define MPU6050_RA_I2C_SLV0_CTRL        0x27
#define MPU6050_RA_I2C_SLV1_ADDR        0x28
#define MPU6050_RA_I2C_SLV1_REG         0x29
#define MPU6050_RA_I2C_SLV1_CTRL        0x2A
#define MPU6050_RA_I2C_SLV2_ADDR        0x2B
#define MPU6050_RA_I2C_SLV2_REG         0x2C
#define MPU6050_RA_I2C_SLV2_CTRL        0x2D
#define MPU6050_RA_I2C_SLV3_ADDR        0x2E
#define MPU6050_RA_I2C_SLV3_REG         0x2F
#define MPU6050_RA_I2C_SLV3_CTRL        0x30
#define MPU6050_RA_I2C_SLV4_ADDR        0x31
#define MPU6050_RA_I2C_SLV4_REG         0x32
```

```
#define MPU6050_RA_I2C_SLV4_DO          0x33
#define MPU6050_RA_I2C_SLV4_CTRL        0x34
#define MPU6050_RA_I2C_SLV4_DI          0x35
#define MPU6050_RA_I2C_MST_STATUS       0x36
#define MPU6050_RA_INT_PIN_CFG          0x37
#define MPU6050_RA_INT_ENABLE           0x38
#define MPU6050_RA_DMP_INT_STATUS       0x39
#define MPU6050_RA_INT_STATUS           0x3A
#define MPU6050_RA_ACCEL_XOUT_H         0x3B
#define MPU6050_RA_ACCEL_XOUT_L         0x3C
#define MPU6050_RA_ACCEL_YOUT_H         0x3D
#define MPU6050_RA_ACCEL_YOUT_L         0x3E
#define MPU6050_RA_ACCEL_ZOUT_H         0x3F
#define MPU6050_RA_ACCEL_ZOUT_L         0x40
#define MPU6050_RA_TEMP_OUT_H           0x41
#define MPU6050_RA_TEMP_OUT_L           0x42
#define MPU6050_RA_GYRO_XOUT_H          0x43
#define MPU6050_RA_GYRO_XOUT_L          0x44
#define MPU6050_RA_GYRO_YOUT_H          0x45
#define MPU6050_RA_GYRO_YOUT_L          0x46
#define MPU6050_RA_GYRO_ZOUT_H          0x47
#define MPU6050_RA_GYRO_ZOUT_L          0x48
#define MPU6050_RA_EXT_SENS_DATA_00     0x49
#define MPU6050_RA_EXT_SENS_DATA_01     0x4A
#define MPU6050_RA_EXT_SENS_DATA_02     0x4B
#define MPU6050_RA_EXT_SENS_DATA_03     0x4C
#define MPU6050_RA_EXT_SENS_DATA_04     0x4D
#define MPU6050_RA_EXT_SENS_DATA_05     0x4E
#define MPU6050_RA_EXT_SENS_DATA_06     0x4F
#define MPU6050_RA_EXT_SENS_DATA_07     0x50
#define MPU6050_RA_EXT_SENS_DATA_08     0x51
#define MPU6050_RA_EXT_SENS_DATA_09     0x52
#define MPU6050_RA_EXT_SENS_DATA_10     0x53
#define MPU6050_RA_EXT_SENS_DATA_11     0x54
#define MPU6050_RA_EXT_SENS_DATA_12     0x55
#define MPU6050_RA_EXT_SENS_DATA_13     0x56
#define MPU6050_RA_EXT_SENS_DATA_14     0x57
#define MPU6050_RA_EXT_SENS_DATA_15     0x58
#define MPU6050_RA_EXT_SENS_DATA_16     0x59
#define MPU6050_RA_EXT_SENS_DATA_17     0x5A
#define MPU6050_RA_EXT_SENS_DATA_18     0x5B
#define MPU6050_RA_EXT_SENS_DATA_19     0x5C
#define MPU6050_RA_EXT_SENS_DATA_20     0x5D
#define MPU6050_RA_EXT_SENS_DATA_21     0x5E
#define MPU6050_RA_EXT_SENS_DATA_22     0x5F
#define MPU6050_RA_EXT_SENS_DATA_23     0x60
#define MPU6050_RA_MOT_DETECT_STATUS    0x61
#define MPU6050_RA_I2C_SLV0_DO          0x63
#define MPU6050_RA_I2C_SLV1_DO          0x64
#define MPU6050_RA_I2C_SLV2_DO          0x65
#define MPU6050_RA_I2C_SLV3_DO          0x66
#define MPU6050_RA_I2C_MST_DELAY_CTRL   0x67
#define MPU6050_RA_SIGNAL_PATH_RESET    0x68
#define MPU6050_RA_MOT_DETECT_CTRL      0x69
#define MPU6050_RA_USER_CTRL            0x6A
#define MPU6050_RA_PWR_MGMT_1           0x6B
#define MPU6050_RA_PWR_MGMT_2           0x6C
```

```
#define MPU6050_RA_BANK_SEL             0x6D
#define MPU6050_RA_MEM_START_ADDR       0x6E
#define MPU6050_RA_MEM_R_W              0x6F
#define MPU6050_RA_DMP_CFG_1            0x70
#define MPU6050_RA_DMP_CFG_2            0x71
#define MPU6050_RA_FIFO_COUNTH          0x72
#define MPU6050_RA_FIFO_COUNTL          0x73
#define MPU6050_RA_FIFO_R_W             0x74
#define MPU6050_RA_WHO_AM_I             0x75


////////////////////////////////////////////
// MPU6050 IMU User Functions
////////////////////////////////////////////

void imuSetUserCtrl(unsigned char byte) {
    twiWriteByte(MPU6050_RA_USER_CTRL, byte);
}

/** Set full-scale gyroscope range.
 *
 * FS_SEL
 * +/- 250*(n+1) degrees/second
 */
void imuSetFullScaleGyroRange(unsigned char range) {
    twiWriteByte(MPU6050_RA_GYRO_CONFIG, range);
}

/** Set full-scale accelerometer range
 *
 * AFS_SEL
 * +/- 2^(n+1) g
 */
void imuSetFullScaleAccelRange(unsigned char range) {
    twiWriteByte(MPU6050_RA_ACCEL_CONFIG, range);
}

/** Set CONFIG register
 *
 * EXT_SYNC_SET
 * 0: Input Disabled
 *
 * DLPF_CFG
 * 1: Accel[184Hz bandwidth, 2.0ms delay], Gyro[188Hz bandwidth, 1.9ms delay, 1kHz Fs]
 */
void imuSetCONFIG(unsigned char byte) {
    twiWriteByte(MPU6050_RA_CONFIG, byte);
}

/** Set PWR_MGT_1 Register
 *
 * Disable Sleep and Cycle mode
 *
 * An internal 8MHz oscillator, gyroscope based clock, or external sources can
 * be selected as the MPU-60X0 clock source. When the internal 8 MHz oscillator
 * or an external source is chosen as the clock source, the MPU-60X0 can operate
 * in low power modes with the gyroscopes disabled.
```

```
 *
 * Upon power up, the MPU-60X0 clock source defaults to the internal oscillator.
 * However, it is highly recommended that the device be configured to use one of
 * the gyroscopes (or an external clock source) as the clock reference for
 * improved stability. The clock source can be selected according to the following table:
 *
 * CLK_SEL | Clock Source
 * --------+--------------------------------------
 * 0       | Internal oscillator
 * 1       | PLL with X Gyro reference
 * 2       | PLL with Y Gyro reference
 * 3       | PLL with Z Gyro reference
 * 4       | PLL with external 32.768kHz reference
 * 5       | PLL with external 19.2MHz reference
 * 6       | Reserved
 * 7       | Stops the clock and keeps the timing generator in reset
 */
void imuSetPWR_MGT_1(unsigned char byte) {
    twiWriteByte(MPU6050_RA_PWR_MGMT_1, byte);
}

/** Power on and prepare for general usage.
 *
 * This will activate the device and take it out of sleep mode (which must be done
 * after start-up). This function also sets both the accelerometer and the gyroscope
 * to their most sensitive settings, namely +/- 2g and +/- 2000 degrees/sec, and sets
 * the clock source to use the X Gyro for reference, which is slightly better than
 * the default internal clock source.
 */
void imuInit() {
    imuSetUserCtrl(0x00);
    imuSetPWR_MGT_1(0x03);
    imuSetCONFIG(0x01);
    imuSetFullScaleGyroRange(0x10);
    imuSetFullScaleAccelRange(0x00);
}

/** Get raw 6-axis motion sensor readings (accel/gyro).
 *
 * Retrieves all currently available motion sensor values.
 * Return array of full-scale float values
 * ax, ay, az
 * gx, gy, gz
 */
void imuGetMotion(float* imu) {
            unsigned char data[14];
            twiReadBytes(MPU6050_RA_ACCEL_XOUT_H, data, 14);

            int16_t axi = (((uint16_t)data[0]) << 8)  | ((uint16_t)data[1]);
            imu[0] = 2 * ((float)axi / ((float)pow(2, 15)));

            int16_t ayi = (((uint16_t)data[2]) << 8)  | ((uint16_t)data[3]);
            imu[1] = 2 * ((float)ayi / ((float)pow(2, 15)));

            int16_t azi = (((uint16_t)data[4]) << 8)  | ((uint16_t)data[5]);
            imu[2] = 2 * ((float)azi / ((float)pow(2, 15)));
```

```
                int16_t gxi = (((uint16_t)data[8]) << 8)  | ((uint16_t)data[9]);
                imu[3] = 1000 * ((float)gxi / ((float)pow(2, 15)));

                int16_t gyi = (((uint16_t)data[10]) << 8)  | ((uint16_t)data[11]);
                imu[4] = 1000 * ((float)gyi / ((float)pow(2, 15)));

                int16_t gzi = (((uint16_t)data[12]) << 8)  | ((uint16_t)data[13]);
                imu[5] = 1000 * ((float)gzi / ((float)pow(2, 15)));
}

/** Sample the IMU for A_x, A_y, G_z
 *
 * Sample the IMU using two separate TWI, I2C, transaction
 * to mazimize speed of entire transaction
 * Return array of full-scale float values
 */
void imuFastSample(float* imu) {
                unsigned char data[4];
                twiReadBytes(MPU6050_RA_ACCEL_XOUT_H, data, 4);

                int16_t axi = (((uint16_t)data[0]) << 8)  | ((uint16_t)data[1]);
                imu[0] = 2 * ((float)axi / ((float)pow(2, 15)));

                int16_t ayi = (((uint16_t)data[2]) << 8)  | ((uint16_t)data[3]);
                imu[1] = 2 * ((float)ayi / ((float)pow(2, 15)));

    twiReadBytes(MPU6050_RA_GYRO_ZOUT_H, data, 2);

                int16_t gzi = (((uint16_t)data[0]) << 8)  | ((uint16_t)data[1]);
                imu[2] = 1000 * ((float)gzi / ((float)pow(2, 15)));
}

#endif
```

Appendix 1d: SAM4S4B_twi.h

```
/*
File:   SAM4S4B_twi.h
Author: Evan Hassman
Email:  ehassman@g.hmc.edu
Date:   November 2019

Description:
    Contains base address locations, register structs, definitions, and functions for the TWI
    (Two-wire Interface) peripheral of the SAM4S4B microcontroller, specifically I2C.

    Initializes to Master Mode
*/

#ifndef SAM4S4B_TWI_H
#define SAM4S4B_TWI_H

#include <stdint.h>
#include "SAM4S4B_pio.h"
#include "SAM4S4B_pmc.h"

//////////////////////////////////////////////////////////////////////////////////////////////
//////
// TWI Base Address Definitions
//////////////////////////////////////////////////////////////////////////////////////////////
//////

#define TWI0_BASE (0x40018000U) // TWI0 Base Address


//////////////////////////////////////////////////////////////////////////////////////////////
//////
// TWI Registers
//////////////////////////////////////////////////////////////////////////////////////////////
//////

// Bit field struct for the TWI_CR register
typedef struct {
    volatile uint32_t START     : 1;
    volatile uint32_t STOP      : 1;
    volatile uint32_t MSEN      : 1;
    volatile uint32_t MSDIS     : 1;
    volatile uint32_t SVEN      : 1;
    volatile uint32_t SVDIS     : 1;
    volatile uint32_t QUICK     : 1;
    volatile uint32_t SWRST     : 1;
    volatile uint32_t           : 24;
} TWI_CR_bits;

// Bit field struct for the TWI_MMR register
typedef struct {
    volatile uint32_t           : 8;
    volatile uint32_t IADRSZ    : 2;
    volatile uint32_t           : 2;
    volatile uint32_t MREAD     : 1;
    volatile uint32_t           : 3;
    volatile uint32_t DADR      : 7;
```

```
    volatile uint32_t           : 9;
} TWI_MMR_bits;

// Bit field struct for the TWI_SMR register
typedef struct {
    volatile uint32_t           : 16;
    volatile uint32_t SADR      : 7;
    volatile uint32_t           : 9;
} TWI_SMR_bits;

// Bit field struct for the TWI_IADR
typedef struct {
    volatile uint32_t IADR      : 24;
    volatile uint32_t           : 8;
} TWI_IADR_bits;

// Bit field struct for the TWI_CWGR register
typedef struct {
    volatile uint32_t CLDIV     : 8;
    volatile uint32_t CHDIV     : 8;
    volatile uint32_t CKDIV     : 3;
    volatile uint32_t           : 13;
} TWI_CWGR_bits;

// Bit field struct for the TWI_SR register
typedef struct {
    volatile uint32_t TXCOMP    : 1;
    volatile uint32_t RXRDY     : 1;
    volatile uint32_t TXRDY     : 1;
    volatile uint32_t SVREAD    : 1;
    volatile uint32_t SVACC     : 1;
    volatile uint32_t GACC      : 1;
    volatile uint32_t OVRE      : 1;
    volatile uint32_t           : 1;
    volatile uint32_t NACK      : 1;
    volatile uint32_t ARBLST    : 1;
    volatile uint32_t SCLWS     : 1;
    volatile uint32_t EOSACC    : 1;
    volatile uint32_t ENDRX     : 1;
    volatile uint32_t ENDTX     : 1;
    volatile uint32_t RXBUFF    : 1;
    volatile uint32_t TXBUFE    : 1;
    volatile uint32_t           : 16;
} TWI_SR_bits;

// Bit field struct for the TWI_RHR
typedef struct {
    volatile uint32_t RXDATA    : 8;
    volatile uint32_t           : 24;
} TWI_RHR_bits;

// Bit field struct for the TWI_THR
typedef struct {
    volatile uint32_t TXDATA    : 8;
    volatile uint32_t           : 24;
} TWI_THR_bits;
```

```
// Peripheral struct for the SPI peripheral
typedef struct {
    volatile TWI_CR_bits    TWI_CR;         // (TWI offset: 0x00) Control Register
    volatile TWI_MMR_bits   TWI_MMR;        // (TWI offset: 0x04) Master Mode Register
    volatile TWI_SMR_bits   TWI_SMR;        // (TWI offset: 0x08) Slave Mode Register
    volatile TWI_IADR_bits  TWI_IADR;       // (TWI offset: 0x0C) Internal Address Register
    volatile TWI_CWGR_bits  TWI_CWGR;       // (TWI offest: 0x10) Clock Waveform Generator
Register
    volatile uint32_t       Reserved1[3];
    volatile TWI_SR_bits    TWI_SR;         // (TWI offest: 0x20) Status Register
    volatile uint32_t       TWI_IER;        // (TWI offset: 0x24) Interrupt Enable Register
    volatile uint32_t       TWI_IDR;        // (TWI offset: 0x28) Interrupt Disable Register
    volatile uint32_t       TWI_IMR;        // (TWI offset: 0x2C) Interrupt Mask Register
    volatile TWI_RHR_bits   TWI_RHR;        // (TWI offset: 0x30) Receive Holding Register
    volatile TWI_THR_bits   TWI_THR;        // (TWI offset: 0x34) Transmit Holding Register
    volatile uint32_t       Reserved2[5];
    volatile uint32_t       Reserved3[10];  // (TWI offset: 0x100-0x128) Reserved for PDC
registers
} Twi;

// Pointer to an Spi-sized chunk of memory at the SPI peripheral
#define TWI ((Twi*) TWI0_BASE)

// The specific PIO pins and peripheral function which TWI uses, set in twiInit()
#define TWI_TWD0_PIN      PIO_PA3
#define TWI_TWCK0_PIN               PIO_PA4
#define TWI_FUNC                    PIO_PERIPH_A


///////////////////////////////////////////////////////////////////////////////////////////////
//////
// TWI User Functions
///////////////////////////////////////////////////////////////////////////////////////////////
//////

/** Enables the TWI peripheral and intializes its slave device address.
 *      -- devAddr: (0x00 to 0x7F) 7 bit address to write to
 * Refer to the datasheet for more low-level details.
 */
void twiInit(uint8_t devAddr) {
    pmcEnablePeriph(PMC_ID_TWI0);
    pioInit();

    pioPinMode(TWI_TWCK0_PIN, TWI_FUNC);  // TWCK0, SCL
    pioPinMode(TWI_TWD0_PIN, TWI_FUNC);  // TWD0, SDA

    TWI->TWI_CR.MSEN = 1;  // Master Mode Enable
    TWI->TWI_CR.MSDIS = 0;  // Master Mode Disable
    TWI->TWI_CR.SVDIS = 1;  // Slave Mode Disable

    TWI->TWI_MMR.IADRSZ = 1;  // 1 byte internal slave device address
    TWI->TWI_MMR.DADR = devAddr;

    // t_high/low = ((C(H/L)DIV * 2^CKDIV) + 4) * t_periphClk
               // 100kHz
    TWI->TWI_CWGR.CHDIV = 96;
```

```
    TWI->TWI_CWGR.CLDIV = 96;
    TWI->TWI_CWGR.CKDIV = 0;  //increase both SCL high and low periods
}

/** Transmits a single byte via I2C
 *      -- data: 8 bit value to be sent
 * Note that during transmition the devAddr and MREAD are sent before
 */
void twiWriteByte(unsigned char regAddr, unsigned char data) {
    TWI->TWI_IADR.IADR = regAddr;  // set slave register internal address
    TWI->TWI_MMR.MREAD = 0;  // Master Write Direction

    // Send Data
    TWI->TWI_THR.TXDATA = data;
    while (!TWI->TWI_SR.TXRDY) {};  // wait for transmition register to empty

    TWI->TWI_CR.STOP = 1;
    while (!TWI->TWI_SR.TXCOMP) {};  // wait for transaction to complete
}

/** Transmit multiple bytes via I2C
 *      -- data is a pointer to the first element of a char array
 * Note that during transmition the devAddr and MREAD are sent before
 * Additionally, an acknowledge is required between each byte
 */
void twiWriteBytes(unsigned char regAddr, unsigned char* data) {
    TWI->TWI_IADR.IADR = regAddr;  // set slave register internal address
               TWI->TWI_MMR.MREAD = 0;  // Master Write Direction

    // Send Data
               for (uint16_t i = 0; i < sizeof(data); ++i) {
                       TWI->TWI_THR.TXDATA = data[i];
                       while (!TWI->TWI_SR.TXRDY) {};  // wait for transmition register to
empty
               }

               TWI->TWI_CR.STOP = 1;
               while (!TWI->TWI_SR.TXCOMP) {};  // wait for transaction to complete
}

/** Read in a single byte via I2C
 *      -- returns byte received
 * Note that for a single byte the STOP flag is set with the START flag
 */
unsigned char* twiReadByte(unsigned char regAddr) {
    TWI->TWI_IADR.IADR = regAddr;  // set slave register internal address
               TWI->TWI_MMR.MREAD = 1;  // Master Read Direction
               TWI->TWI_CR.START = 1;
               TWI->TWI_CR.STOP = 1;

    // Wait for transmition to finish
               while(!TWI->TWI_SR.RXRDY) {};

    unsigned char* data;
               *data = TWI->TWI_RHR.RXDATA;
       return data;
}
```

```
/** Read in multiple bytes via I2C
 *                    -- data: char array to be filled with data
 *      -- amount: (>1) the number of bytes to be received, size of data
 *      -- return bytes in array
 * Note that the STOP flag must be set following the next-to-last data received
 */
void twiReadBytes(unsigned char regAddr, unsigned char* data, uint16_t amount) {
    TWI->TWI_IADR.IADR = regAddr;  // set slave register internal address
    TWI->TWI_MMR.MREAD = 1;  // Master Read Direction
              TWI->TWI_CR.START = 1;

    for(uint16_t i = 0; i < amount; ++i) {
                      while(!TWI->TWI_SR.RXRDY) {};
      data[i] = (char)TWI->TWI_RHR.RXDATA;

        if (i == amount-2)
            TWI->TWI_CR.STOP = 1;
    }
}

#endif
```

Appendix 1e: FPGA_spi.h

```
/*
File:   FPGA_spi.h
Author: Evan Hassman
        Wilson Ives
Email:  ehassman@g.hmc.edu
        wives@g.hmc.edu
Date:   November 2019

Description:

*/

#ifndef FPGA_SPI_H
#define FPGA_SPI_H

#include "../SAM4S4B/SAM4S4B_spi.h"
#include "../SAM4S4B/SAM4S4B_pio.h"
#include "../SAM4S4B/SAM4S4B_tc.h"

#define H_BRIDGE_SS_PIN             PIO_PA15
#define ENCODER_SS_PIN        PIO_PA16
#define FPGA_RESET_PIN        PIO_PA17

#define PI 3.14159265358979323846264 33


////////////////////////////////////////////////////////////////
// SPI Communication with FPGA
////////////////////////////////////////////////////////////////

/** Initialize SPI communication w/ FPGA
 *
 * Initialize and set both slave select pins low
 * Initialize and flash on FPGA reset pin
 */
void FPGA_spiInit() {
              pioPinMode(H_BRIDGE_SS_PIN, PIO_OUTPUT);
    pioDigitalWrite(H_BRIDGE_SS_PIN, PIO_LOW);
              pioPinMode(ENCODER_SS_PIN, PIO_OUTPUT);
    pioDigitalWrite(H_BRIDGE_SS_PIN, PIO_LOW);

    // FPGA Reset
    pioPinMode(FPGA_RESET_PIN, PIO_OUTPUT);
    pioDigitalWrite(FPGA_RESET_PIN, PIO_HIGH);
    pioDigitalWrite(FPGA_RESET_PIN, PIO_LOW);
}

/** Transmit Control Effort via SPI
 *
 * Send the sign-magnitude 12-bit control effort
 * duty cycle via SPI to the FPGA
 */
void sendControlEffort (uint16_t* u) {
    uint8_t MSB = (uint8_t)(*u>>8);
    uint8_t LSB = (uint8_t)((*u<<8)>>8);
              pioDigitalWrite(H_BRIDGE_SS_PIN, PIO_HIGH);
```

```
              spiSendReceive(MSB);
    spiSendReceive(LSB);
              pioDigitalWrite(H_BRIDGE_SS_PIN, PIO_LOW);
}

/** Calculate W_f using SPI transaction
 *
 * Receive number of encoder state changes and number
 * of clock cycles during quadrature encoder sample
 * from FPGA via SPI. Use values to calculate W_f
 */
void receiveQuadratureEncoderData (float* wf) {
    // slave select high
              pioDigitalWrite(ENCODER_SS_PIN, PIO_HIGH);

              int16_t encoderStateChanges = spiSendReceive16(0x4444);
              uint32_t numClockCycles = spiSendReceive32U(0x44444444);

    // slave select low
    pioDigitalWrite(ENCODER_SS_PIN, PIO_LOW);

    if (numClockCycles == 0)
      *wf = 0;
    else
      *wf = 2*PI*40000000/1920.0*((float)encoderStateChanges)/((float)numClockCycles);

}

#endif
```

Appendix 1f: Serial.h

```
/*
File:   Serial.h
Author: Evan Hassman
        Wilson Ives
Email:  ehassman@g.hmc.edu
        wives@g.hmc.edu
Date:   November 2019

Description:
                             Communicate via UART to FT232R chip to then communicate
                             via serial to PC COM Port
                             Used for debugging
*/

#ifndef SERIAL_H
#define SERIAL_H

#include "../SAM4S4B/SAM4S4B_uart.h"
#include "stdio.h"
#include <string.h>
#include <stdlib.h>

/////////////////////////////////////////////////////////////////
// UART to Serial Communication
/////////////////////////////////////////////////////////////////

#define SERIAL_PARITY 0
#define SERIAL_BAUD_RATE_DIV 21 // from lab 6 demo 21.7 would yield 115200 baud for 40MHz
clock

/** Print array of chars
 *
 * Transmit array of chars over UART to be serial printed
 */
void serialPrint(char* str) {
              char* ptr = str;
              while (*ptr) uartTx(*ptr++);
}

/** Print array of chars with newline
 *
 * Transmit array of chars followed by newline character
 * over UART to be printed
 */
void serialPrintln(char* str) {
              char* ptr = str;
              while (*ptr) uartTx(*ptr++);
              uartTx('\n'); // newline
              uartTx('\r'); // return
}

/** Print single char in HEX
 *
 * Convert char value to HEX char array
 * to be printed
 */
```

```
void serialPrintByteHex(uint8_t byte) {
              char hex_str[3];
              sprintf(hex_str, "%02X", (unsigned int)byte);
              serialPrint(hex_str);
}

/** Print string of chars in HEX
 *
 * Convert chars in string to HEX char array
 * to be printed
 */
void serialPrintBytesHex(char* str) {
              char* ptr = str;
              while (*ptr) serialPrintByteHex(*ptr++);
}

/** Print a uint32_t in HEX
 *
 * Convert unsigned 32-bit int to HEX char array
 * to be printed
 */
void serialPrintU32Hex(uint32_t value) {
    char hex_str[4];
    sprintf(hex_str, "%08X", value);
              serialPrint(hex_str);
}

/** Print a short in HEX
 *
 * Convert short to HEX char array to be printed
 */
void serialPrintShortHex(short data) {
              char lo = data & 0xFF;
              char hi = data >> 8;
              serialPrintByteHex(hi);
              serialPrintByteHex(lo);
}

/** Print a float
 *
 * Convert float to char array to be printed
 */
void serialPrintFloat(float num, int digits) {
    char str[digits];
    sprintf(str, "%f", num);
    serialPrint(str);
}

/** Print state of system
 *
 * Print W_a, Theta_a, and W_f as floats in easy to read
 * group within serial monitor
 */
void serialPrintState(float* state) {
    serialPrintln("------------------");
    serialPrint("W_a:             ");
    serialPrintFloat(state[0], 5);
```

```
    serialPrintln("");
    serialPrint("Theta_a:         ");
    serialPrintFloat(state[1], 5);
    serialPrintln("");
    serialPrint("W_f:             ");
    serialPrintFloat(state[2], 5);
    serialPrintln("");
}

/** Print state of system and control effort
 *
 * Print W_a, Theta_a, W_f, and control effort as floats
 * in easy to read group within serial monitor
 */
void serialPrintStateControl(float* state, float* u) {
    serialPrintState(state);
    serialPrint("Control Effort:  ");
    serialPrintFloat(*u, 5);
    serialPrintln("");
}

/** Print state of system
 *
 * Print W_a, Theta_a, and W_f to be interpretted by
 * Arduino serial plotter
 */
void serialPrintStatePlotter(float* state) {
    serialPrintFloat(state[0], 5);
    serialPrint(", ");
    serialPrintFloat(state[1], 5);
    serialPrint(", ");
    serialPrintFloat(state[2], 5);
    serialPrintln("");
}

/** Print state of system and control effort
 *
 * Print W_a, Theta_a, W_f, and control effort to be
 * interpretted by Arduino serial plotter
 */
void serialPrintStateControlPlotter(float* state, float* u) {
    serialPrintFloat(state[0], 5);
    serialPrint(",");
    serialPrintFloat(state[1], 5);
    serialPrint(",");
    serialPrintFloat(state[2], 5);
    serialPrint(",");
    serialPrintFloat(*u, 5);
    serialPrintln("");
}
#endif
```

Appendix 2: Verilog

```
////////////////////////////////////////////
// final_project.sv
// HMC E155 11 December 2019
// ehassman@g.hmc.edu, wives@g.hmc.edu
////////////////////////////////////////////

// wrapper module for spi_slave, h_bridge_driver, and quadrature_decoder modules
module final_project(   input  logic clk, rst,
                        input  logic sck,
                        input  logic mosi,
                        output logic miso,
                        input  logic encoder_ss, h_bridge_ss,
                                                 input  logic enc_a, enc_b,
                        output logic IN1,IN2);
    logic [15:0] u;
    logic [47:0] enc;
        logic a,b;

        synchronizer2 s2(clk,rst,{enc_a,enc_b},{a,b}); // synchronizer for encoder inputs

    // instantiate spi slave module
        spi_slave ss(clk,rst,sck,mosi,miso,h_bridge_ss,encoder_ss,u,enc);
    // instantiate h_bridge_driver module
    h_bridge_driver hbd(clk,rst,h_bridge_ss,u,IN1,IN2);
    // instantiate quadrature_decoder module
    quadrature_decoder qd(clk,rst,a,b,enc);

endmodule


// wrapper module for quadrature decoder spi slave and h-bridge driver spi slave
module spi_slave(input logic clk, rst,
                                           input logic sck,
                                           input logic mosi,
                                           output logic miso,
                                           input logic h_bridge_ss, encoder_ss,
                                           output logic [15:0] u,
                                           input logic [47:0] enc);
       logic [15:0] u_recieved, u_return;
       logic [47:0] enc_recieved, enc_return;
       logic h_bridge_miso, enc_miso;

       assign u_return = 16'b1; // this value doesnt matter
       assign miso = (h_bridge_ss) ? h_bridge_miso : enc_miso;

       // spi slave modules
       h_bridge_spi_slave hbss(sck,mosi,h_bridge_miso,rst,h_bridge_ss,u_return,u_recieved);
       enc_spi_slave ess(sck,mosi,enc_miso,rst,encoder_ss,enc_return,enc_recieved);
       // registers
       reg_16 tr1(clk,rst,!h_bridge_ss,u_recieved,u);
       reg_48 tr2(clk,rst,!encoder_ss,enc,enc_return);

endmodule


// spi slave for h-bridge driver
```

```
// recieves 16 bit control effort from spi master returns meaningless data
module h_bridge_spi_slave(input logic sck, // From master
                                                             input logic mosi, // From
master
                                                             output logic miso, // To
master
                                                             input logic rst, // System
reset
                                                             input logic h_bridge_ss, //
SS pin
                                                             input logic [15:0] u_return,
// Data to send
                                                             output logic [15:0]
u_recieved); // Data received

       logic [3:0] cnt; // this has to correspond with the number of bits being sent and
received
       logic qdelayed;

       // 4-bit counter tracks when all 16 bits are transmitted
       always_ff @(negedge sck, posedge rst)
              if (rst)
                     cnt = 4'b0;
              else if(h_bridge_ss)
                     cnt = cnt + 4'b1;

       // Loadable shift register
       // Loads d at the start, shifts mosi into bottom on each step
       always_ff @(posedge sck)
              if(h_bridge_ss)
                     u_recieved <= (cnt == 4'b0) ? {u_return[14:0], mosi} :
{u_recieved[14:0], mosi};
       // Align miso to falling edge of sck
       // Load d at the start
       always_ff @(negedge sck)
              if(h_bridge_ss)
                     qdelayed = u_recieved[15];

       assign miso = (cnt == 4'b0) ? u_return[15] : qdelayed;

endmodule

// spi slave for quadrature decoder
// recieves any 48 bit value from spi master and returns 48 bits from quadrature decoder
module enc_spi_slave(input logic sck, // From master
                                                  input logic mosi, // From master
                                                  output logic miso, // To master
                                                  input logic rst, // System reset
                                                  input logic enc_ss, // SS pin
                                                  input logic [47:0] enc_return, // Data to
send
                                                  output logic [47:0] enc_recieved); // Data
received

       logic [5:0] cnt; // this has to correspond with the number of bits being sent and
received
       logic qdelayed;
```

```
        // 4-bit counter tracks when all 32 bits are transmitted
        always_ff @(negedge sck, posedge rst)
                if (rst)
                        cnt = 6'b0;
                else if(enc_ss)
                        if(cnt == 47)
                                cnt = 6'b0;
                        else
                                cnt = cnt + 6'b1;

        // Loadable shift register
        // Loads d at the start, shifts mosi into bottom on each step
        always_ff @(posedge sck)
                if(enc_ss)
                        enc_recieved <= (cnt == 6'b0) ? {enc_return[46:0], mosi} :
{enc_recieved[46:0], mosi};
        // Align miso to falling edge of sck
        // Load d at the start
        always_ff @(negedge sck)
                if(enc_ss)
                        qdelayed = enc_recieved[47];

        assign miso = (cnt == 6'b0) ? enc_return[47] : qdelayed;

endmodule


// This module drives an H-bridge IC
// The two outputs IN1 and IN2 control the direction of the power applied to the load
// | IN1 | IN2 | Result
// | ----|-----|--------------
// |  0  |  0  | Brake to GND
// |  0  |  1  | Drive CCW
// |  1  |  0  | Drive CW
// |  1  |  1  | DBrake to VCC
module h_bridge_driver( input logic clk, rst,
                                                          input logic h_bridge_ss,
                                                          input logic [11:0] u,
                        output logic IN1,IN2);

         logic [14:0] counter_state;
    logic [11:0] current_u, next_current_u; // stores the sampled value of u so u doesn't
change in the middle of a PWM cycle

    // Next State CL
    always_comb
        begin
        // next counter state and next current u state
            if ((counter_state == 15'b111111111111111)&(!h_bridge_ss)) // this will occur at
40MHz/(2^12) = 9.766kHz
                                    next_current_u = u; // sample input u and store in fsm
state register at the beginning of each PWM cycle
            else
                next_current_u = current_u; // no change
        end
```

```
    // Register
    always_ff @(posedge clk)
        if(rst)
            begin
               counter_state   <= 15'b000000000000000;
               current_u       <= 12'b000000000000;
            end
        else
                      begin
                counter_state   <= counter_state + 15'b1;
                current_u       <= next_current_u;
            end

    // Output CL
    always_comb
              begin
                     IN1 = current_u[11] & (counter_state[14:4] < current_u[10:0]);
                     IN2 = (~current_u[11]) & (counter_state[14:4] < current_u[10:0]);
              end
endmodule


// This module decodes two quadrature encoder inputs and returns a 48 bit number enc.
// The 2 MSBs of enc is the number of encoder state changes that have occured during the last
sample period.
// The next 4 bytes are the number of clock cycles that have occured between the first and
last encoder
// state change during the sample period
module quadrature_decoder(  input  logic clk, rst,
                            input  logic enc_a, enc_b,
                            output logic [47:0] enc);

    // next_enc holds the next output of the module formatted as described in the module
declaration
    logic [47:0] next_enc;

    // encoder state keeps track of which of the four possible states the quadrature encode is
in
    typedef enum {s0,s1,s2,s3} encoder_state_type;
    encoder_state_type encoder_state, next_encoder_state;

    // increment holds number of signed state changes since the last calculation
    // max of increments 10560 per second or 211 per 20ms (50Hz sample rate)
    logic [15:0] increment, next_increment;

    // counter holds the number of clock cycles since the last calculation
    //      @ 50Hz sample rate 800,000 clock cycles will occur. 2^20 is 1,048,576
    logic [31:0] counter, next_counter;

    // first_transition_time is a timestamp for the first change of encoder_state after
    logic [31:0] first_transition_time, next_first_transition_time;

    // last_transition_time is a timestamp for the last change of encoder_state
    logic [31:0] last_transition_time, next_last_transition_time;

    // when first_transition is 1 the encoder hasn't changed state since the counter was reset
    logic first_transition, next_first_transition;
```

```
    logic new_sample; // goes high for a single clock cycle at 50Hz to calculate w_f
    // new sample triggers the output enc to be updated and the fsm state to be reset

    pulse_gen pg(clk, rst, new_sample);

    // next state CL
    always_comb
        begin
            // next encoder state
            case({enc_a,enc_b})
                2'b00: next_encoder_state = s0;
                2'b01: next_encoder_state = s1;
                2'b11: next_encoder_state = s2;
                2'b10: next_encoder_state = s3;
            endcase

            // reset fsm state except for enc when new_sample is started
            if(new_sample)
                begin
                    next_counter                = 32'b0;
                    next_increment              = 16'b0;
                    next_first_transition       = 1'b1;
                    next_first_transition_time  = 32'b0;
                    next_last_transition_time   = 32'b0;
                    next_enc = {increment,(last_transition_time-first_transition_time)};
                end

            // continue incrementing the fsm state as usual
            else
                begin
                    // next enc
                    next_enc = enc; // no change until new_sample is asserted

                    // next counter
                    next_counter = counter + 32'b1;

                    // next increment
                    if(~first_transition) // only count increments after the first state
transition
                        case({encoder_state,next_encoder_state})
                            {s0,s1}: next_increment = increment + 16'b1;
                            {s0,s3}: next_increment = increment - 16'b1;
                            {s1,s2}: next_increment = increment + 16'b1;
                            {s1,s0}: next_increment = increment - 16'b1;
                            {s2,s3}: next_increment = increment + 16'b1;
                            {s2,s1}: next_increment = increment - 16'b1;
                            {s3,s0}: next_increment = increment + 16'b1;
                            {s3,s2}: next_increment = increment - 16'b1;
                            default: next_increment = increment; // this contains cases if
states are skipped for some reason
                        endcase
                    else // if first transition is high, increment remains zero
                        next_increment = 16'b0;

                    // next first_transition, first_transition_time, and last_transition_time
                    if(encoder_state != next_encoder_state) // a state transition has occured
```

```
                        if(first_transition) // this state change is the first of this sample
                            begin
                                // update both timestamps
                                next_first_transition_time = counter;
                                next_last_transition_time = counter;
                                // any state transitions that follow are not the first
transition anymore
                                next_first_transition = 1'b0;
                            end
                        else // this state change is not the first of this sample
                            begin
                                next_first_transition_time = first_transition_time; // don't
change
                                next_last_transition_time = counter; // update last transition
time
                                next_first_transition = first_transition; // will be zero
already
                            end
                    else // if no state transition has occured don't change anything
                        begin
                            next_first_transition = first_transition;
                            next_first_transition_time = first_transition_time;
                            next_last_transition_time = last_transition_time;
                        end
                end
        end

    // Register
    always_ff @(posedge clk)
        if(rst) // reset fsm state
            begin
                                     counter                     <= 32'b0;
                increment                          <= 16'b0;
                first_transition                   <= 1'b1;
                first_transition_time              <= 32'b0;
                last_transition_time               <= 32'b0;
                encoder_state                      <= s0; // this shouldn't really be reset to
a known value
                enc                                <= 48'b0;
            end
        else // normal operation
                  begin
                counter                 <= next_counter;
                encoder_state           <= next_encoder_state;
                increment               <= next_increment;
                first_transition        <= next_first_transition;
                first_transition_time   <= next_first_transition_time;
                last_transition_time    <= next_last_transition_time;
                                     enc                     <= next_enc;

            end

endmodule


// Generates a single clock cycle pulse every 20ms (50Hz)
// to begin a new sample of the quadrature decoder
```

```
module pulse_gen(input  logic clk, rst,
                 output logic pulse);
       // This module takes a 40MHz clock signal (clk) as input
    // and outputs single clock cycle pulse every 400,000 clock
    // cycles (50Hz)
    logic [23:0] counter, reset_val;
    assign reset_val = 799999;

    always_comb
        if(counter==reset_val)
            pulse = 1;
        else
            pulse = 0;

    always_ff @(posedge clk, posedge rst)
           if (rst)  counter <= 24'b0;
        else
            if(counter == reset_val)
                counter <= 24'b0;
            else
                counter <= counter + 24'b1;
endmodule

// 16 bit register used in spi slave module for control effort
module reg_16(input logic clk,rst,en,
                                    input logic [15:0] in,
                                    output logic [15:0] out);

       always_ff @(posedge clk, posedge rst)
              if(rst)
                     out <= 16'b0;
              else if(en)
                     out <= in;
endmodule

// 48 bit register used in spi slave module for decoder output
module reg_48(input logic clk,rst,en,
                                    input logic [47:0] in,
                                    output logic [47:0] out);

       always_ff @(posedge clk, posedge rst)
              if(rst)
                     out <= 48'b0;
              else if(en)
                     out <= in;
endmodule

// 2 bit synchronizer for encoder inputs
module synchronizer2(input logic clk,rst,
                                                   input logic [1:0] in,
                                                   output logic [1:0] out);

       always_ff @(posedge clk, posedge rst)
              if(rst)
                     out <= 2'b0;
              else
                     out <= in;
```

```
endmodule
```

Appendix 3: Matlab Code

Appendix 3a: Flywheel_pendulum_swingup.m

```
clear all
clc

Kt = 0.2497;
Kb = 0.3472;
R_m = 2.2;
L_arm = 0.25;
% the values below were computed using calculate_parameters.m
I_f = 0.00475;
I_0 = 0.0657;
Tau_g = 2.468;


% Model linearized about theta_a = 0
A_top = ...
    [0, Tau_g/I_0,  Kt*Kb/(R_m*I_0);
     1, 0,          0;
     0, -Tau_g/I_0, -Kb*Kt*(I_0+I_f)/(I_f*I_0*R_m)];

% Model linearized about theta_a = pi
A_bot = ...
    [0, -Tau_g/I_0, Kt*Kb/(R_m*I_0);
     1, 0,          0;
     0, Tau_g/I_0,  -Kb*Kt*(I_0+I_f)/(I_f*I_0*R_m)];

B = [-Kt/(R_m*I_0);
     0;
     Kt*(I_0+I_f)/(I_0*I_f*R_m)];
C = [1,0,0;
     0,1,0;
     0,0,1];
D = 0;

%% Calculate Control Gains for Stabilizing About Theta_A = 0
Q_top = [0.1, 0,   0;   % penalizes w_a
         0,   1,   0;   % penalizes theta_a
         0,   0,   0.1];% penalizes w_f
% large R penalizes control effort
R_top = 10;
[K_top,S,cl_poles_top] = lqr(A_top,B,Q_top,R_top);
K_top = -K_top;

%% Calculate Control Gains for Stabilizing About Theta_A = pi
% Q_bot = [0.1, 0,   0;   % penalizes w_a
%          0,   10,  0;   % penalizes theta_a
%          0,   0,   0.1];% penalizes w_f
% % large R penalizes control effort
% R_bot = 1;
% [K_bot,S,cl_poles_bot] = lqr(A_bot,B,Q_bot,R_bot);
% K_bot = -K_bot
K_bot = -place(A_bot,B,[-5,-6,-7]);

%% physical limits of system
sat = 12;  % voltage saturation
```

```
I_max = 3; % motor current saturation

control = @(x) Control(x,sat,I_max,Kb,R_m,K_top,K_bot,Tau_g,I_0,I_f);

% Nonlinear Model
x_dot = @(t,x) ...
    [
     % arm angular acceleration
     Kb*Kt/(I_0*R_m)*x(3)...
     + Tau_g/I_0*sin(x(2))...
     - Kt/(I_0*R_m)*control(x);
     % arm angular velocity
     x(1);
     % flywheel angular acceleration
     -Kb*Kt*(I_0+I_f)/(I_f*I_0*R_m)*x(3)...
     - Tau_g/I_0*sin(x(2))...
     + Kt*(I_0+I_f)/(I_f*I_0*R_m)*control(x)
    ];

% flywheel kinetic energy
flywheel_energy = @(x) 0.5*I_f*(x(:,1)+x(:,3)).^2;

% arm kinetic energy
arm_K_energy = @(x) 0.5*I_0*x(:,1).^2;

% pendulum potential energy
arm_P_energy = @(x) Tau_g*(1+cos(x(:,2)));

% motor current
current = @(x) (control(x)-x(:,3)*Kb)/R_m;

% normalized flywheel height
height = @(x) L_arm*cos(x(2));

tspan = [0,5];
%x_0 = [0,5*pi/180,0];
%x_0 = [-5.2283,0.1699,31.9262];
x_0 = [0,pi,0];

%% Simulate System
options = odeset('MaxStep',0.01);
[t,state] = ode45(x_dot, tspan, x_0, options);

%% Plot Simulation Results

% % simulated ROS
% load('Stable_ICs_2_Imax3.mat')
% shrink_factor = 1; % 0.5 default
% k = boundary(Stable_ICs,shrink_factor);
% w_a_min = -15;
% w_a_max = 15;
% theta_a_min = -3;
% theta_a_max = 3;
% w_f_min = -40; % no load speed of motor
% w_f_max = 40;
%
% % parameters for fit ROS surface
```

```
% n_points = 100;
% fit_theta_a = linspace(-pi,pi,n_points);
% fit_w_f = linspace(-36,36,n_points);
% [FIT_THETA_A,FIT_W_F] = meshgrid(fit_theta_a,fit_w_f);
% FIT_W_A_T = -11.7*sin(0.5*FIT_THETA_A)-1*FIT_W_F/35 + 0.8;
% FIT_W_A_B = -11.7*sin(0.5*FIT_THETA_A)-1*FIT_W_F/35 - 0.8;
%
% figure(1)
% delete(findall(gcf,'Type','light'))
% plot3(state(:,1),state(:,2),state(:,3),'b-','LineWidth',2)
% hold on
% light('Position',[-40 4 60],'Style','local')
% light('Position',[40 -4 60],'Style','local')
% trisurf(k,Stable_ICs(:,1),Stable_ICs(:,2),Stable_ICs(:,3),...
%     'FaceColor',[0 0.5 0],'FaceAlpha',0.2,'LineStyle','none',...
%     'FaceLighting','gouraud')
% surf(FIT_W_A_T,FIT_THETA_A,FIT_W_F,'EdgeColor','none','FaceColor','blue',...
%     'FaceAlpha',0.2,'FaceLighting','gouraud')
% surf(FIT_W_A_B,FIT_THETA_A,FIT_W_F,'EdgeColor','none','FaceColor','blue',...
%     'FaceAlpha',0.2,'FaceLighting','gouraud')
% hold off
%
% ah = gca;
% title('Inverted Pendulum ROS');
% xlabel('w_a [rad/s]');
% ylabel('theta_a [rad]');
% zlabel('w_f [rad/s]');
% set(ah,'Xlim',[-20,20])
% set(ah,'Ylim',[2*theta_a_min,2*theta_a_max])
% set(ah,'Zlim',[w_f_min,w_f_max])
% set(ah,'FontSize',14);
% set(ah,'TitleFontSizeMultiplier',1.2);
% set(ah,'LineWidth',1);
% set(ah,'PlotBoxAspectRatio',[10,10,6])
% grid on

figure(2)
plot(t,state(:,1),'b-','LineWidth',2)
ah = gca;
title('Pendulum Angular Speed');
xlabel('Time [s]');
ylabel('Angular Speed [rad/s]');
set(ah,'FontSize',12);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on

figure(3)
plot(t,state(:,2),'b-','LineWidth',2)
ah = gca;
title('Pendulum Angle');
xlabel('Time [s]');
ylabel('Pendulum Angle [rad]');
set(ah,'FontSize',12);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on
```

```
figure(4)
plot(t,state(:,3),'b-','LineWidth',2)
ah = gca;
title('Flywheel Angular Speed');
xlabel('Time [s]');
ylabel('Flywheel Angular Speed [rad/s]');
set(ah,'FontSize',12);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on

figure(5)
plot(t,control(state),'b-','LineWidth',2)
ah = gca;
title('Control Effort');
xlabel('Time [s]');
ylabel('Input Voltage [V]');
set(ah,'FontSize',12);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on

figure(6)
plot(t,current(state),'b-','LineWidth',2)
ah = gca;
title('Motor Current');
xlabel('Time [s]');
ylabel('Motor Current [A]');
set(ah,'FontSize',12);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on

% E_a = arm_K_energy(state) + arm_P_energy(state);
% E_f = flywheel_energy(state);
% figure(7)
% plot(t,E_a,'b-','LineWidth',2)
% hold on
% plot(t,2*Tau_g+E_f,'k:','LineWidth',1)
% plot(t,2*Tau_g-E_f,'k:','LineWidth',1)
% hold off
% ah = gca;
% title('Pendulum Arm Energy +/- Flywheel Energy');
% xlabel('Time [s]');
% ylabel('System Energy [J]');
% set(ah,'FontSize',12);
% set(ah,'TitleFontSizeMultiplier',1.2);
% set(ah,'LineWidth',1);
% legend('E_a','E_a +/- E_f')
% grid on
%
% figure(8)
% plot(t,sufficient_E(state,Tau_g,I_0,I_f),'b-','LineWidth',2)
% ah = gca;
% title('Sufficient Energy');
% xlabel('Time [s]');
```

```
% ylabel('Sufficient Energy [logical]');
% set(ah,'FontSize',12);
% set(ah,'TitleFontSizeMultiplier',1.2);
% set(ah,'LineWidth',1);
% grid on
%
% figure(9)
% plot(t,in_small_angle(state),'b-','LineWidth',2)
% ah = gca;
% title('In Small Angle');
% xlabel('Time [s]');
% ylabel('In Small Angle [logical]');
% set(ah,'FontSize',12);
% set(ah,'TitleFontSizeMultiplier',1.2);
% set(ah,'LineWidth',1);
% grid on

% u = control(state);
% cur = current(state);
% state2video(state,u,cur,t,L_arm,50);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Helper Functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Functions for bounding control input

% limits u between -sat to sat
function u = saturation(u_in,sat)
    u = min(max(u_in,-sat),sat);
end

% limits u such that the max current in the motor is I_max
function u = current_limit(u_in,wf,I_max,Kb,R_m)
    u = min(max(u_in,wf*Kb-I_max*R_m),wf*Kb+I_max*R_m);
end

%% Function for angle rollover

% limits theta between -pi and pi
function x = limit_theta(x)
    if(length(x)>3) % x is a multidimensional array
        x(:,2) = mod(x(:,2)+pi,2*pi)-pi;
    else % x is a single state during simulation and has dimension 3x1
        x(2) = mod(x(2)+pi,2*pi)-pi;
    end
end

%% Funtions for determining if state is within certain subsets of state space

% returns 1 if x is in ROA for assumed I_max
function logic = in_ROA(x)
    x = limit_theta(x);
    a = -11.7;
    slope = 1/35;
    width = 0.8; % 0.8;
    if(length(x)>3) % x is a multidimensional array
```

```
        logic = (x(:,1)<(a*sin(0.5*x(:,2))-slope*x(:,3)+width)...
                & (x(:,1)>(a*sin(0.5*x(:,2))-slope*x(:,3)-width)));
    else % x is a single state during simulation and has dimension 3x1
        logic = (x(1)<(a*sin(0.5*x(2))-slope*x(3)+width)...
                & (x(1)>(a*sin(0.5*x(2))-slope*x(3)-width)));
    end
end

% energy threshold
% returns true if the system has sufficient energy to get to vertical
function logic = sufficient_E(x,Tau_g,I_0,I_f)
    x = limit_theta(x);
    E_eq = 2*Tau_g;
    buf = 0.2; % [J]
    if(length(x)>3) % x is a multidimensional array
        E_arm = (Tau_g*(1+cos(x(:,2))) + 0.5*I_0*x(:,1).^2);
        E_f = 0.5*I_f*(x(:,1)+x(:,3)).^2;
        logic = (x(:,1)>=0).*(E_arm>(E_eq-E_f+buf)) + (x(:,1)<0).*(E_arm>(E_eq+E_f+buf));
    else % x is a single state during simulation and has dimension 3x1
        E_arm = Tau_g*(1+cos(x(2))) + 0.5*I_0*x(1)^2;
        E_f = 0.5*I_f*(x(1)+x(3))^2;
        logic = (x(1)>=0)*(E_arm>(E_eq-E_f+buf)) + (x(1)<0)*(E_arm>(E_eq+E_f+buf));
    end
end

% small angle test
% returns true if the system is with +/- 0.5 rad
function logic = in_small_angle(x)
    x = limit_theta(x);
    if(length(x)>3) % x is a multidimensional array
        logic = (x(:,2)>-0.5) & (x(:,2)<0.5);
    else % x is a single state during simulation and has dimension 3x1
        logic = (x(2)>-0.5) & (x(2)<0.5);
    end
end

%% Full State Feedback Controllers (used by ode45 solver)

% calculates control effort for stabilization around theta_a = 0
% given voltage saturation and current limiting to I_max.
function u = u_top(x,sat,I_max,Kb,R_m,K_top)
    x = limit_theta(x);
    if(length(x)>3) % x is the output of a simulation and has dimension Nx3
        u = saturation(current_limit(...
            K_top(1)*x(:,1)+K_top(2)*x(:,2)+K_top(3)*x(:,3),...
            x(:,3),I_max,Kb,R_m),sat);
    else % x is a single time step during a ode45 simulation and has dimensions 3x1
        u = saturation(current_limit(...
            K_top(1)*x(1)+K_top(2)*x(2)+K_top(3)*x(3),...
            x(3),I_max,Kb,R_m),sat);
    end
end

% calculates control effort for stabilization around theta_a = pi
% given voltage saturation and current limiting to I_max.
function u = u_bot(x,sat,I_max,Kb,R_m,K_bot)
    x_eq_bot = [0,pi,0];
```

```
    if(length(x)>3) % x is the output of a simulation and has dimension Nx3
        x(:,2) = x(:,2)-pi;
        x = limit_theta(x);
        u = saturation(current_limit(...
            K_bot(1)*(x(:,1)-x_eq_bot(1))+K_bot(2)*x(:,2)+...
            K_bot(3)*(x(:,3)-x_eq_bot(3)),x(:,3),I_max,Kb,R_m),sat);
    else % x is a single time step during a ode45 simulation and has dimensions 3x1
        x(2) = x(2)-pi;
        x = limit_theta(x);
        u = saturation(current_limit(...
            K_bot(1)*(x(1)-x_eq_bot(1))+K_bot(2)*x(2)+...
            K_bot(3)*(x(3)-x_eq_bot(3)),x(3),I_max,Kb,R_m),sat);
    end
end

% calculates control effort for swing up
% given voltage saturation and current limiting to I_max.
% this is simply a bang-bang controller
function u = u_swingup(x,sat,I_max,Kb,R_m)
    if(length(x)>3) % x is the output of a simulation and has dimension Nx3
        u = saturation(current_limit(...
            -12*sign(x(:,1)),x(:,3),I_max,Kb,R_m),sat);
    else % x is a single time step during a ode45 simulation and has dimensions 3x1
        u = saturation(current_limit(...
            -12*sign(x(1)),x(3),I_max,Kb,R_m),sat);
    end
end

% Calculates control effort to set the motor current to zero
% This will ensure no more energy is added or removed from the system
function u = u_no_energy(x,sat,Kb)
    if(length(x)>3) % x is the output of a simulation and has dimension Nx3
        u = saturation(Kb*x(:,3),sat);
    else % x is a single time step during a ode45 simulation and has dimensions 3x1
        u = saturation(Kb*x(3),sat);
    end
end

%% Wrapper Control Function
function u = Control(x,sat,I_max,Kb,R_m,K_top,K_bot,Tau_g,I_0,I_f)

    u = in_ROA(x).*u_top(x,sat,I_max,Kb,R_m,K_top)...
       + not(in_ROA(x)).*u_swingup(x,sat,I_max,Kb,R_m);
    %u = u_top(x,sat,I_max,Kb,R_m,K_top);
    %u = u_bot(x,sat,I_max,Kb,R_m,K_bot);
    %u = u_swingup(x,sat,I_max,Kb,R_m);
    %u = u_no_energy(x,sat,Kb);
    %u = not(in_small_angle(x)).*u_no_energy(x,sat,Kb)+... % stop adding energy to the system
    %    (in_small_angle(x)).*u_top(x,sat,I_max,Kb,R_m,K_top); % stabilize at top
    %u = not(sufficient_E(x,Tau_g,I_0,I_f)).*u_swingup(x,sat,I_max,Kb,R_m)+... % keep swinging
up
    %    (sufficient_E(x,Tau_g,I_0,I_f)).*u_no_energy(x,sat,Kb); % stop adding energy to the
system
    %u = u_top(x,sat,I_max,Kb,R_m,K_top);

end
```

Appendix 3b: Region_of_attraction.m

```
clear all
clc

Kt = 0.2497;
Kb = 0.3472;
R_m = 2.2;
L_arm = 0.25;
% the values below were computed using calculate_parameters.m
I_f = 0.00475;
I_0 = 0.0657;
Tau_g = 2.468;

% Model linearized about theta_a = 0
A_top = ...
    [0, Tau_g/I_0,  Kt*Kb/(R_m*I_0);
     1, 0,          0;
     0, -Tau_g/I_0, -Kb*Kt*(I_0+I_f)/(I_f*I_0*R_m)];

B = [-Kt/(R_m*I_0);
     0;
     Kt*(I_0+I_f)/(I_0*I_f*R_m)];
C = [1,0,0;
     0,1,0;
     0,0,1];
D = 0;

%% Calculate Control Gains for Stabilizing About Theta_A = 0
Q_top = [0.1, 0,   0;   % penalizes w_a
         0,   1,   0;   % penalizes theta_a
         0,   0,   0.1];% penalizes w_f
% large R penalizes control effort
R_top = 10;
[K_top,S,cl_poles_top] = lqr(A_top,B,Q_top,R_top);
K_top = -K_top;

%% physical limits of system
sat = 12;
I_max = 3;

control = @(x) Control(x,sat,I_max,Kb,R_m,K_top);

%% Nonlinear Model
x_dot = @(t,x) ...
    [
     % arm angular acceleration
     Kb*Kt/(I_0*R_m)*x(3)...
     + Tau_g/I_0*sin(x(2))...
     - Kt/(I_0*R_m)*control(x);
     % arm angular velocity
     x(1);
     % flywheel angular acceleration
     -Kb*Kt*(I_0+I_f)/(I_f*I_0*R_m)*x(3)...
     - Tau_g/I_0*sin(x(2))...
     + Kt*(I_0+I_f)/(I_f*I_0*R_m)*control(x)];

tspan = [0,4];
```

```
%%
N = 201;
w_a_min = -15;
w_a_max = 15;
N_w_a = N;
theta_a_min = -3;
theta_a_max = 3;
N_theta_a = N;
w_f_min = -40; % no load speed of motor
w_f_max = 40;
N_w_f = 50;

W_A = linspace(w_a_min,w_a_max,N_w_a);
THETA_A = linspace(theta_a_min,theta_a_max,N_theta_a);
W_F = linspace(w_f_min,w_f_max,N_w_f);

Stable_ICs = [0,0,0];

% for i=1:N_w_a
%     i
%     for j=1:N_theta_a
%         for k=1:N_w_f
%             x_0 = [W_A(i),THETA_A(j),W_F(k)];
%             [t,state] = ode45(x_dot, tspan, x_0);
%             if((state(end,1)^2<0.01)&&(state(end,2)^2<0.0001)&&...
%                     (state(end,3)^2<0.01)&&(max(abs(state(:,2)))<3))
%                 Stable_ICs = vertcat(Stable_ICs,x_0);
%             end
%         end
%     end
% end

%Stable_ICs = Stable_ICs(2:end,:); % remove first point

load('Stable_ICs_2_Imax3.mat')

% smooth3, alphashape, trisurf (with boundary)
shrink_factor = 1; % 0.5 default
k = boundary(Stable_ICs,shrink_factor);
%Stable_ICs = Stable_ICs(k,:); % remove internal points

%% make fit surface
n_points = 100;
fit_theta_a = linspace(-4,4,n_points);
fit_w_f = linspace(-40,40,n_points);
[FIT_THETA_A,FIT_W_F] = meshgrid(fit_theta_a,fit_w_f);
FIT_W_A_T = -11.7*sin(0.5*FIT_THETA_A)-1*FIT_W_F/35 + 0.8;
FIT_W_A_B = -11.7*sin(0.5*FIT_THETA_A)-1*FIT_W_F/35 - 0.8;

figure(1)
%scatter3(Stable_ICs(:,1),Stable_ICs(:,2),Stable_ICs(:,3),'k.')
%hold on
% plot interpolated surface
trisurf(k,Stable_ICs(:,1),Stable_ICs(:,2),Stable_ICs(:,3),...
    'FaceColor',[0 0.5 0],'FaceAlpha',0.6,'LineStyle','none',...
    'FaceLighting','gouraud')
```

```
hold on
% plot fit surface
surf(FIT_W_A_T,FIT_THETA_A,FIT_W_F,'EdgeColor','none','FaceColor','blue',...
    'FaceAlpha',0.5,'FaceLighting','gouraud')
surf(FIT_W_A_B,FIT_THETA_A,FIT_W_F,'EdgeColor','none','FaceColor','blue',...
    'FaceAlpha',0.5,'FaceLighting','gouraud')
% make lights for shading
delete(findall(gcf,'Type','light'))
light('Position',[-40 4 60],'Style','local')
light('Position',[40 -4 60],'Style','local')

hold off

ah = gca;
title('Inverted Pendulum ROS');
xlabel('w_a [rad/s]');
ylabel('theta_a [rad]');
zlabel('w_f [rad/s]');
set(ah,'Xlim',[2*w_a_min,2*w_a_max])
set(ah,'Ylim',[2*theta_a_min,2*theta_a_max])
set(ah,'Zlim',[w_f_min,w_f_max])
set(ah,'FontSize',14);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
set(ah,'PlotBoxAspectRatio',[10,10,6])
grid on

%% make video of ROA

f2 = figure(2);
set(f2,'Position',[0,0,1120,840]);
trisurf(k,Stable_ICs(:,1),Stable_ICs(:,2),Stable_ICs(:,3),...
    'FaceColor',[0 0.5 0],'FaceAlpha',0.6,'LineStyle','none',...
    'FaceLighting','gouraud')
hold on
theta_plot = linspace(-pi,pi,100);
w_plot = -11.7*sin(0.5*theta_plot);
w_f_plot = 35*ones(100,1);
plot3(w_plot,theta_plot,w_f_plot,'k-','LineWidth',2)
hold off
delete(findall(gcf,'Type','light'))
light('Position',[-40 4 60],'Style','local')
light('Position',[40 -4 60],'Style','local')
ah = gca;
title('Inverted Pendulum ROS');
xlabel('w_a [rad/s]');
ylabel('theta_a [rad]');
zlabel('w_f [rad/s]');
set(ah,'Xlim',[2*w_a_min,2*w_a_max])
set(ah,'Ylim',[2*theta_a_min,2*theta_a_max])
set(ah,'Zlim',[w_f_min,w_f_max])
set(ah,'FontSize',14);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
set(ah,'PlotBoxAspectRatio',[10,10,6])
view(90,90)
grid on
```

```
% set(ah,'CameraViewAngleMode','Manual')
% OptionZ.FrameRate=30;OptionZ.Duration=10;OptionZ.Periodic=true;
% CaptureFigVid([-90,25;-45,25;0,25;45,25;90,25;90,90],'ROA_fig',OptionZ)
% view(30,25)

%% Functions for bounding control input

% limits u between -sat to sat
function u = saturation(u_in,sat)
    u = min(max(u_in,-sat),sat);
end

% limits u such that the max current in the motor is I_max
function u = current_limit(u_in,wf,I_max,Kb,R_m)
    u = min(max(u_in,wf*Kb-I_max*R_m),wf*Kb+I_max*R_m);
end

%% Function for angle rollover

% limits theta between -pi and pi
function x = limit_theta(x)
    x(2) = mod(x(2)+pi,2*pi)-pi;
end

%% Full State Feedback Controllers (used by ode45 solver)

% calculates control effort for stabilization around theta_a = 0
% given voltage saturation and current limiting to I_max.
function u = ode45_u_top(x,sat,I_max,Kb,R_m,K_top)
    u = saturation(current_limit(...
    K_top(1)*x(1)+K_top(2)*x(2)+K_top(3)*x(3),x(3),I_max,Kb,R_m),sat);
end

%% Wrapper Control Function
function u = Control(x,sat,I_max,Kb,R_m,K_top)
    u = ode45_u_top(limit_theta(x),sat,I_max,Kb,R_m,K_top);
end
```

Appendix 3c: State_to_video.m

```
function [] = state2video(state_in,u_in,current_in,t_in,L_arm,fps)
% This funtion takes a simulated sequence of states and the corresponding
% times. The function saves a video of the system response over time.

% interpolate simulated data to have a fixed time step
t = (0:1/fps:t_in(end)).';
w_a = interp1(t_in,state_in(:,1),t);
theta_a = interp1(t_in,state_in(:,2),t);
w_f = interp1(t_in,state_in(:,3),t);
state = horzcat(w_a,theta_a,w_f);
u = interp1(t_in,u_in,t);
current = interp1(t_in,current_in,t);

% ROS surface plot
n_points = 100;
fit_theta_a = linspace(-pi,pi,n_points);
fit_w_f = linspace(-35,35,n_points);
[FIT_THETA_A,FIT_W_F] = meshgrid(fit_theta_a,fit_w_f);
FIT_W_A_T = -11.7*sin(0.5*FIT_THETA_A)-1*FIT_W_F/35 + 0.8;
FIT_W_A_B = -11.7*sin(0.5*FIT_THETA_A)-1*FIT_W_F/35 - 0.8;
ROS_w_a = vertcat(FIT_W_A_T(:),FIT_W_A_B(:));
ROS_theta_a = vertcat(FIT_THETA_A(:),FIT_THETA_A(:));
ROS_w_f = vertcat(FIT_W_F(:),FIT_W_F(:));
ROS_surf = horzcat(ROS_w_a,ROS_theta_a,ROS_w_f);
shrink_factor = 1;
k = boundary(ROS_surf,shrink_factor);

fig = figure(10);
set(fig,'Position',[200,200,1000,500]);
axis tight manual
set(gca,'nextplot','replacechildren');

% make and open video file
v = VideoWriter('test_video.mp4','MPEG-4');
v.FrameRate = fps/2;
v.Quality = 75;

open(v);

for i = 1:length(t)
    % delete old annotations
    delete(findall(gcf,'type','annotation'))

    %% plot physical system
    subplot(2,2,[1,3]);
    % plot horizantal line
    plot([-2*L_arm,2*L_arm],[0,0],'k:','Linewidth',2)
    hold on
    % plot pendulum arm
    plot([0,L_arm*sin(theta_a(i))],[0,L_arm*cos(theta_a(i))],...
        '-','Color',[0.4,0.4,0.4],'Linewidth',4)
    % plot flywheel
    plot(L_arm*sin(theta_a(i)),L_arm*cos(theta_a(i)),...
        'o','Color',[0.4,0.4,0.4],'MarkerSize',85,'Linewidth',2)
    % plot motor
    plot(L_arm*sin(theta_a(i)),L_arm*cos(theta_a(i)),...
```

```
        '.','Color',[0.4,0.4,0.4],'MarkerSize',50,'Linewidth',2)
    % plot axis of rotation
    plot(0,0,'.','Color',[0,0,0],'MarkerSize',30,'Linewidth',2)
    % add time in textbox
    dim = [.15 .2 .06 .05];
    str = "t = " + num2str(round(t(i),2),3);
    annotation('textbox',dim,'String',str,'FontSize',14);
    hold off
    title('Flywheel Pendulum Simulated Response');
    xlabel('X [m]');
    ylabel('Y [m]');
    axis equal
    set(gca,'XLim',[-2*L_arm,2*L_arm]);
    set(gca,'YLim',[-2*L_arm,2*L_arm]);
    xticks([-0.5 -0.25 0 0.25 0.5])
    yticks([-0.5 -0.25 0 0.25 0.5])
    set(gca,'FontSize',14);
    set(gca,'TitleFontSizeMultiplier',1.2);
    set(gca,'LineWidth',1);
    grid on

    %% plot control effort over time
    subplot(2,2,2);
    plot(t(1:i),u(1:i),'b','LineWidth',2)
    title('Control Effort vs Time');
    xlabel('time [s]');
    ylabel('Motor Drive Volatge [V]');
    set(gca,'XLim',[0,t(end)]);
    set(gca,'YLim',[-13,13]);
    yticks([-12 -6 0 6 12])
    set(gca,'FontSize',12);
    set(gca,'TitleFontSizeMultiplier',1.2);
    set(gca,'LineWidth',1);
    grid on

    %% plot current effort over time
    subplot(2,2,4);
    plot(t(1:i),current(1:i),'b','LineWidth',2)
    title('Motor Current vs Time');
    xlabel('time [s]');
    ylabel('Motor Current [V]');
    set(gca,'XLim',[0,t(end)]);
    set(gca,'YLim',[-(1+round(max(abs(current)))),(1+round(max(abs(current))))]);
    set(gca,'FontSize',12);
    set(gca,'TitleFontSizeMultiplier',1.2);
    set(gca,'LineWidth',1);
    grid on

    %% plot control effort over time
%     subplot(2,2,[2,4]);
%     trisurf(k,ROS_surf(:,1),ROS_surf(:,2),ROS_surf(:,3),...
%     'FaceColor',[0 0.6 0],'FaceAlpha',0.5,'LineStyle','none',...
%     'FaceLighting','gouraud')
%     hold on
%     plot3(state(1:i,1),state(1:i,2),state(1:i,3),'k-','LineWidth',2)
%     delete(findall(gcf,'Type','light'))
%     light('Position',[-40 4 60],'Style','local')
```

```
%     light('Position',[40 -4 60],'Style','local')
%
%     title('State Over Time');
%     xlabel('{\omega}_a [rad/s]');
%     ylabel('{\theta}_a [rad]');
%     zlabel('{\omega}_f [rad/s]');
%     set(gca,'XLim',[-20,20]);
%     set(gca,'YLim',[-2*pi,2*pi]);
%     set(gca,'ZLim',[-36,36]);
%     set(gca,'FontSize',14);
%     set(gca,'TitleFontSizeMultiplier',1.2);
%     set(gca,'LineWidth',1);
%     set(gca,'PlotBoxAspectRatio',[10,10,6])
%     view([0,0,1])
%     grid on


    % save figure as a frame of a video
    f = getframe(gcf);
    writeVideo(v,f);
end
close(v);
end
```

Appendix 3d: calculate _parameters.m

```
% all numbers are in SI units

g = 9.81;

%% Motor
Kt = 0.2497;
Kb = 0.3472;
R_motor = 2.2;

%% Flywheel
ro = 0.1;
ri = 0.0875;
t = 0.0254;
rho = 2700;

% mass
m_rim = pi*(ro^2-ri^2)*t*rho;
m_spokes = 0.00762*0.01524*4*ri*rho;
m_bracket = 5/9*0.00258*0.0127*rho;
M_flywheel = m_rim + m_spokes + m_bracket;

% moment of inertia
I_rim = 0.5*m_rim*(ro^2+ri^2);
I_spokes = 1/3*m_spokes*ri^2;
I_bracket = 1/3*m_bracket*0.0254^2;
I_flywheel = I_rim + I_spokes + I_bracket

%% Entire Pendulum
L_arm = 0.25;
Tau_g_arm = 0.84;
I_arm = 0.019456;

I_0 = I_flywheel + M_flywheel*L_arm^2 + I_arm
Tau_g = M_flywheel*L_arm*g + Tau_g_arm
```

Appendix 3d: calculate _parameters.m

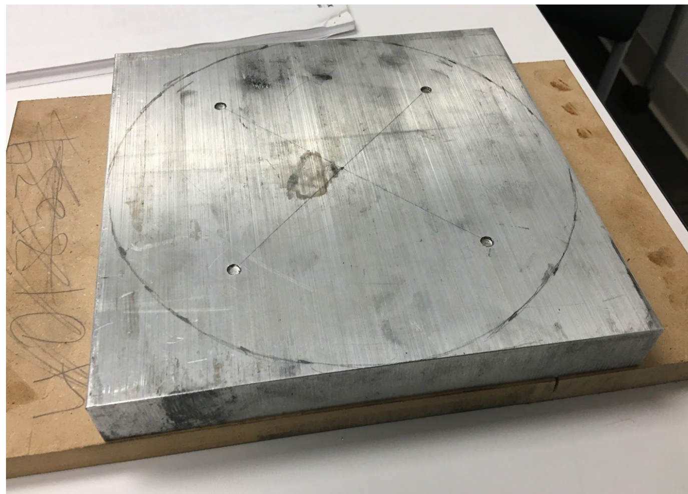
```
% Load csv data
% data1 = csvread("test_csv.txt");
% data2 = csvread("inverted_stabilization_test_12-4.txt");
% data3 = csvread("first_avg_test.txt");
% data4 = csvread("second_avg_test.txt");
swing_up = csvread("swing_up.txt");

% vectorize data
% w_a_1 = data1(:,1);     w_a_2 = data2(:,1);     w_a_3 = data3(:,1);     w_a_4 = data4(:,1);
% theta_a_1 = data1(:,2); theta_a_2 = data2(:,2); theta_a_3 = data3(:,2); theta_a_4 =
data4(:,2);
% w_f_1 = data1(:,3);     w_f_2 = data2(:,3);     w_f_3 = data3(:,3);     w_f_4 = data4(:,3);
% u_1 = data1(:,4);       u_2 = data2(:,4);       u_3 = data3(:,4);       u_4 = data4(:,4);
w_a = swing_up(:,1);
theta_a = swing_up(:,2);
w_f = swing_up(:,3);
u = swing_up(:,4);
t = (0:1:length(w_a)-1)*1/75;

% shift angle data
% theta_a_1 = (theta_a_1 >= 0).*theta_a_1 + (theta_a_1 < 0).*(theta_a_1 + 2*pi);
% theta_a_2 = (theta_a_2 >= 0).*theta_a_2 + (theta_a_2 < 0).*(theta_a_2 + 2*pi);
% theta_a_3 = (theta_a_3 >= 0).*theta_a_3 + (theta_a_3 < 0).*(theta_a_3 + 2*pi);
% theta_a_4 = (theta_a_4 >= 0).*theta_a_4 + (theta_a_4 < 0).*(theta_a_4 + 2*pi);
theta_a = (theta_a >= 0).*theta_a + (theta_a < 0).*(theta_a + 2*pi);
theta_a = ((theta_a >= 0.5).*theta_a + (theta_a < 0.5).*(theta_a + 2*pi))-2*pi;

figure(1)
plot(t(1:7*75),theta_a(76:8*75),'b-','LineWidth',2)
ah = gca;
title('Pendulum Arm Angle vs Time (Swingup)');
xlabel('Time [s]');
ylabel('Pendulum Angle [rad]');
set(ah,'XLim',[0,7]);
%set(ah,'YLim',[-2,12]);
yticks([-2*pi -1.5*pi -pi -pi/2 0])
yticklabels({'-2\pi','-1.5\pi','-\pi','-0.5\pi','0'})
set(ah,'FontSize',14);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on

figure(2)
plot(t(1:9*75),theta_a(8*75+1:17*75)*180/pi,'b-','LineWidth',2)
ah = gca;
title('Pendulum Arm Angle vs Time (Disturbance Rejection)');
xlabel('Time [s]');
ylabel('Pendulum Angle [degrees]');
set(ah,'XLim',[0,9]);
set(ah,'YLim',[-7,7]);
set(ah,'FontSize',14);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on
```

```
figure(3)
plot(t(1:5*75),theta_a(18*75+1:23*75),'b-','LineWidth',2)
ah = gca;
title('Pendulum Arm Angle vs Time (Balance Down)');
xlabel('Time [s]');
ylabel('Pendulum Angle [rad]');
set(ah,'XLim',[0,5]);
set(ah,'YLim',[-5.5,1]);
yticks([-1.5*pi -pi -pi/2 0])
yticklabels({'-1.5\pi','-\pi','-0.5\pi','0'})
set(ah,'FontSize',14);
set(ah,'TitleFontSizeMultiplier',1.2);
set(ah,'LineWidth',1);
grid on
```

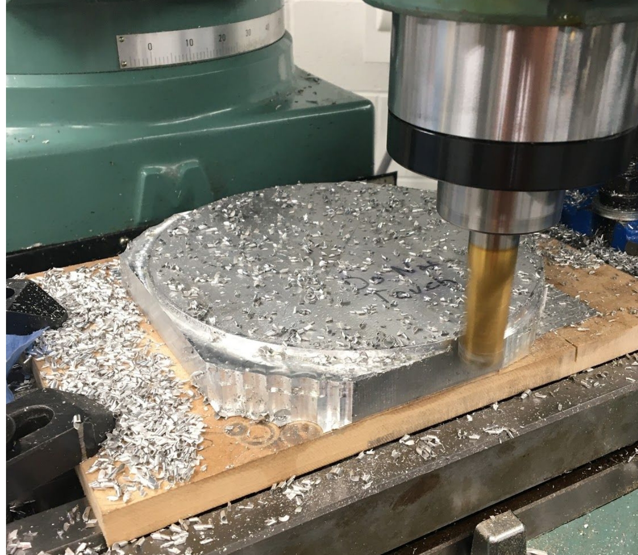
Appendix 4: Machining

Bearing mounts

Motor mounting bracket

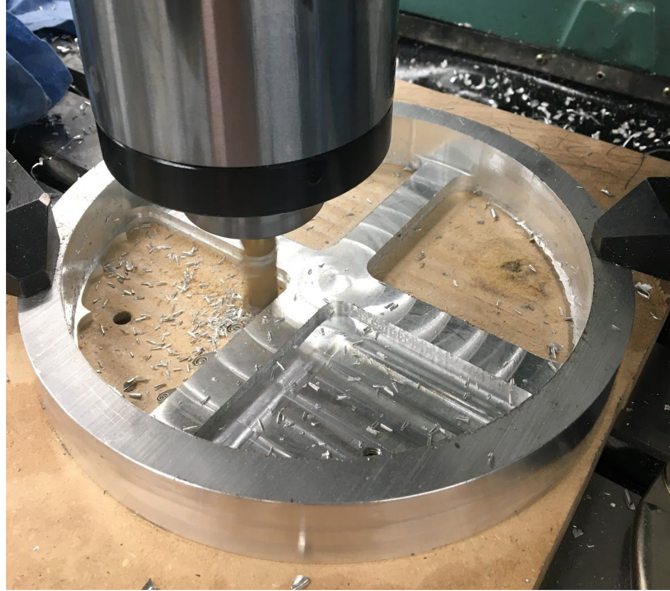


Flywheel stock



Machining flywheel rim

Machining flywheel interior



Machining flywheel spokes