ECE 4760 Lab 3 PID Control of 1D Helicopter

Ridhwan Ahmed and Nikita Dolgopolov

Lab Group 3

April 8th, 2025

High Level Overview:

The goal of LAB 3 is to assemble and develop software for a 1D helicopter. The helicopter is assembled with the use of a stick with an IMU sensor and a powerful fan at the end connected to the fixed table via hinge. The sensor is used to get data about the current tilt angle of the assembly and the speed of its motion. To control the motor, PWM is used and the power is supplied via a specially-made optically isolated board.

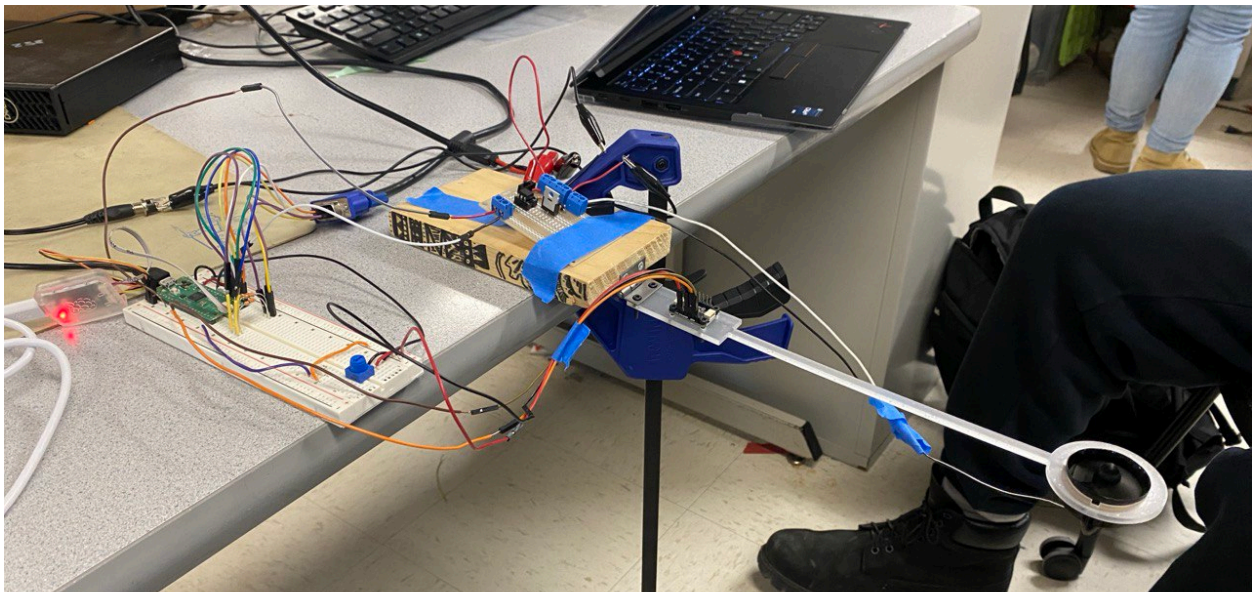

Figure 1: Full Project Assembly

The hardware in use for this project are RP2040 microcontroller, a reset button, a control button, a VGA screen with relevant circuitry and wiring, a motor driving circuit, inertial measurement unit and the motor assembly.

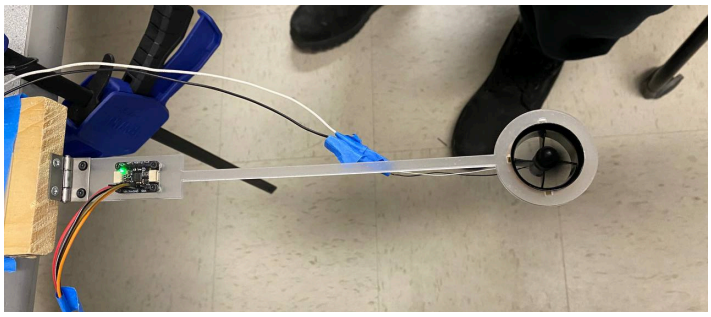

Figure 2: Helicopter Arm Assembly

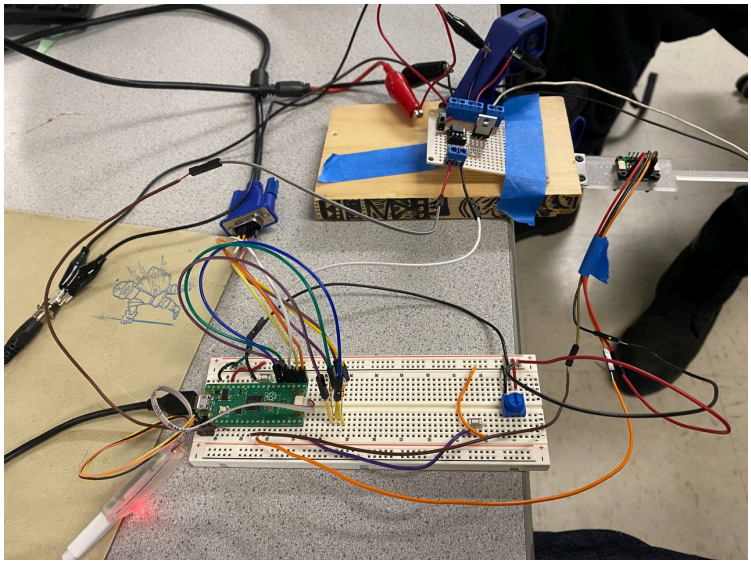

Figure 3: Project Circuitry and Electronics

The Software side of the project includes the following blocks: PID control, fixed point decimal arithmetics for speed, VGA screen communication and graphics, button press debouncing, and filtering of sensor readings.

Software Logic Overview

High-level Overview

The main blocks of the software include sensor reading, PID controls calculation and writing, graphics, and sequence control via button press and debouncing logic. The IMU data from gyroscope and accelerometer is filtered and processed to be sent over to the PID controller for calculation of the control input. The sensor data is also used to plot the position of the arm and other relevant values on the VGA screen. On the button press, a predefined motion sequence is executed by the arm by setting different control inputs to the motor. The interrupt is called at the frequency of 1000Hz, and PWM value is adjusted, sensor is read, and button is debounced inside of the interrupt.service routine.

Sensor Data and Filtering:

The gyroscope outputs information about the angular speed of the arm, while the accelerometer measures force components in different directions. The angle of the arm can be calculated by taking an appropriate arctangent of the acceleration readings with the use of the geometry and direction of gravity vector. The sensor readings are noisy, and gyro tends to drift over time. Therefore, it makes sense to develop a complementary filter that looks at the readings of both of the sensors.To achieve this, the gyro reading as viewed as change in angle (which it pretty much is) and is added to the previous value of the complementary angle value; to this value, a new readings of the accelerometer angle is added with a weight of 1/1000 - an alpha filter of a sort. This ensures that the value of the complementary angle does not change as fast and smoothes it out. This technique, however, turned out to produce a nonzero offset from the true value and was not strong enough by itself to ensure smoothness. Therefore, to compute the

current angle, a value of 17 degrees is added to the value of the complementary angle and then the current angle is passed through the alpha filter with coefficients of 0.5 and 0.5 on its new and old values to obtain a pretty accurate smoothly changing value of the angle of the arm.

PID Controls and PWM

The PID control of the program relies on the current values of the angle of the arm, the rate of its change, the desired angle, and the history of these values. For proportional control only, the proportional error is calculated as the difference between the desired arm angle and the current arm angle; for derivative control, the change in the error is calculated; for the integral control, the proportional error is integrated over time. To limit integrator windup, the integrated term is reset to 20000 in the case that it reaches a value of 1000000 - this can happen once in about 40 seconds of operation if the arm is converging to the desired angle particularly slowly. It is a potential issue for this system due to the fact that the external force (gravity) is very nonlinear and is highest around an angle of 90 degrees and is reducing towards the angles of 0 and 180 degrees.

To calculate the value of the control signal, the proportional error, derivative of the error, and the integral of the error are scaled by weights of proportional gain Kp, derivative gain Kd, and integral gain Ki, respectively, and added together.

We have converged on the following optimal set of control gain values:

```
Kp = 36;
Kd = 8000;
Ki = 0.0065;
```

The control signal is also low-passed to prevent significant spikes in voltage and inductance effects in the motor circuit and are limited to the region between 0 and 94 percent of the maximum motor PWM. Below, a sample plot on the VGA screen is presented. While the gyro and duty cycle values can be noisy and can change rather quickly, the current angle value changes smoothly and closely depicts the actual angle of the arm. It is a well tuned response with a decaying exponential to move from the old desired angle value to the new desired angle value of 120 degrees (with a small margin of error). After enough time, the arm converges to the desired angle with an error of about 1 degree.There are no significant oscillations on the bottom plot and the desired value is reached within about 300-400 milliseconds.

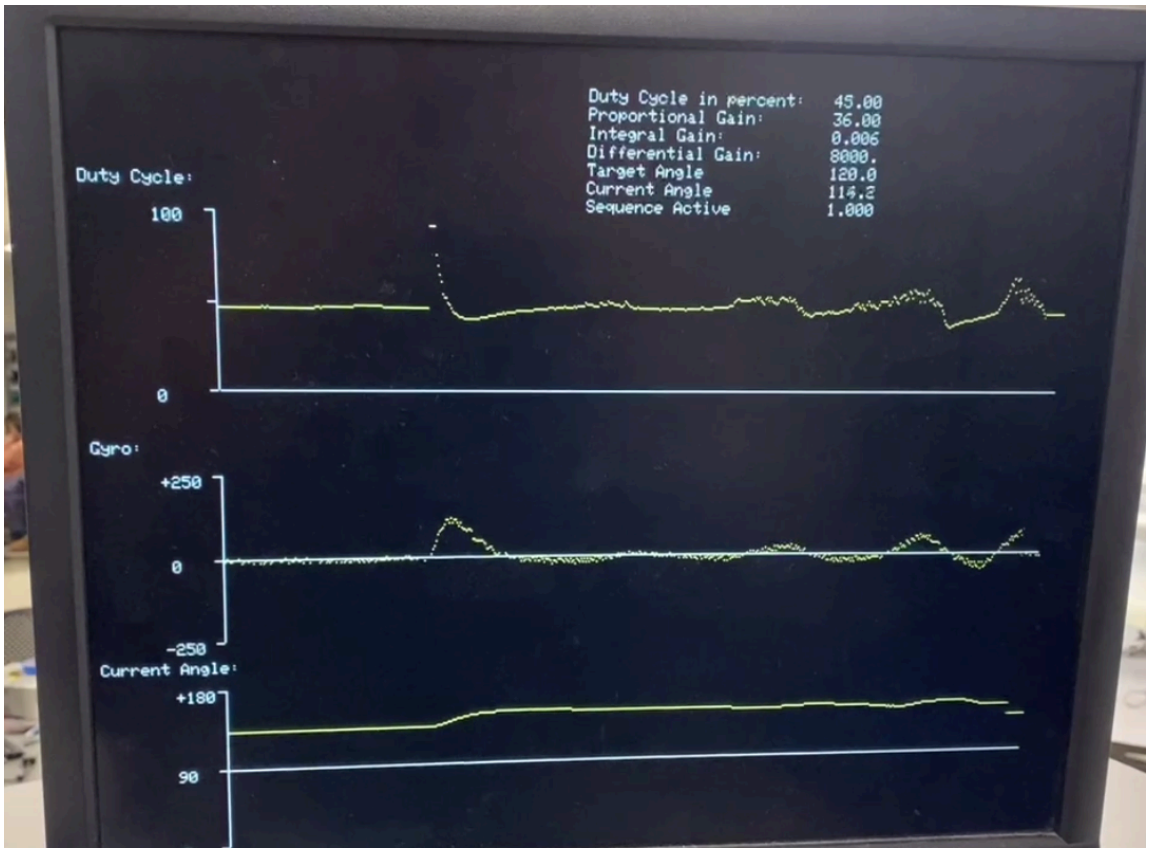

Figure 4: PID Response

Graphics

In the image below, a typical state of the VGA screen is presented: there are three plots for duty cycle written to the motor, gyroscope reading, and for the current arm angle in addition to the outputs of the values of the PID control parameters and angle and duty cycle values.

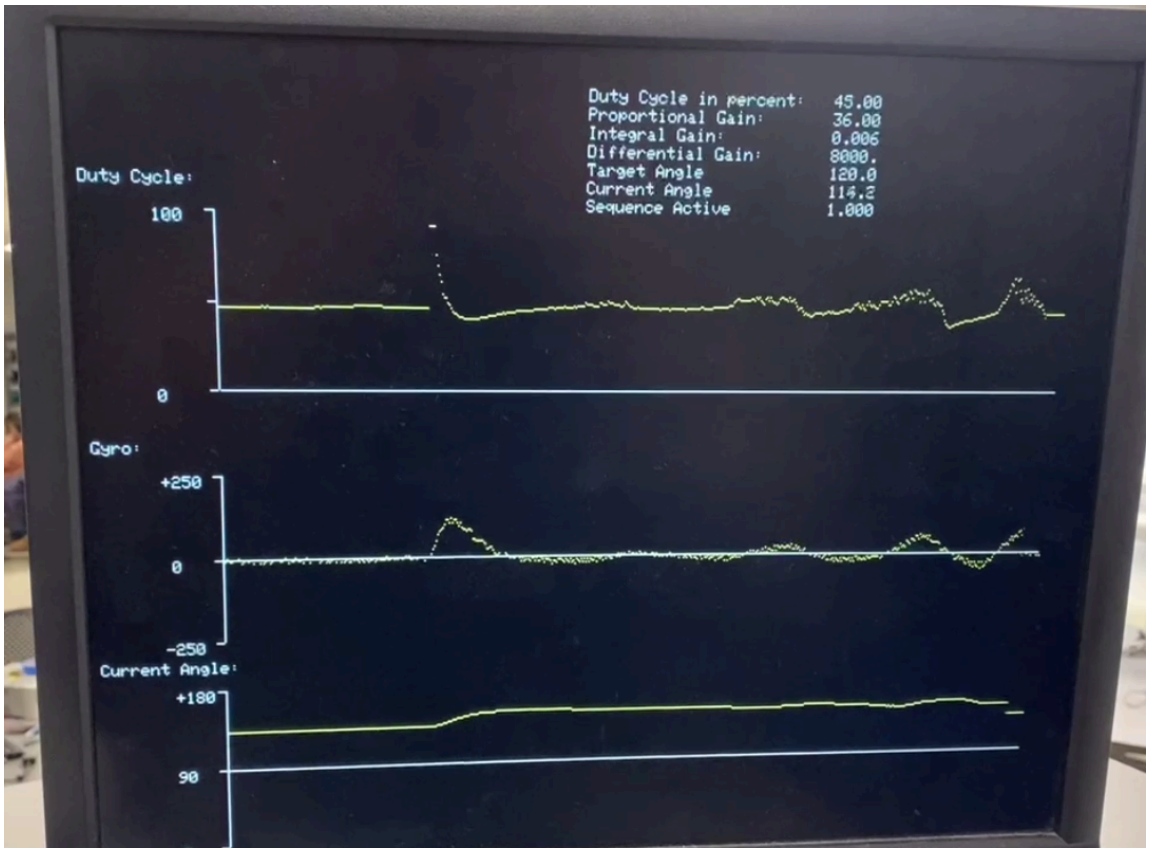

Figure 5: VGA Screen Graphics

To display parameters, the text is broken down into two parts: static and dynamic. The static text includes the part of the message that stays unchanged throughout the program, such as the labels to parameters of PID controls, and is drawn at initialization - we don't need to waste CPU cycles on redrawing it. The dynamic part of the text, such as the actual values of parameters PID parameters and target and current angles, actually has to be updated and is redrawn on every frame. The plot axes are also redrawn on each iteration in case they get erased by a new point on the plot.

To draw the plots, a counter is incremented to increase the x-coordinate of the next points on each plot with a wraparound at the end of the screen to initial coordinate at the start of the graph. The y-values of points are calculated from the current values of the parameters scaled to the size of the plot and converted to int for VGA coordinate. Before drawing the new point, the old point is erased by painting a black column through the plot at the current x-position.

Button Input and Motion Sequence Logic

The button logic is implemented by configuring a designated GPIO pin as an input with its internal pull-up enabled. This means that when the button is not pressed, the pin reads a high voltage (logic 1), and when the button is pressed (assuming the other terminal is tied to ground), it reads low (logic 0). Within the PWM interrupt service routine (ISR) that runs roughly at 1 kHz, the software continuously polls the state of the button. When the button is detected as pressed (i.e. the GPIO input returns 0), the

code marks that the button is being held by setting a flag (button_held = true) and cancels any active sequence (sequence_active = false). In this pressed state, the motor control command is forced to zero (or set to a “motor off” value), ensuring that the beam remains in a safe “vertical down” position (or otherwise defined safe state) while the button is held. Once the button is released (the GPIO reads high again) and if the system was previously in the "button held" state, a transition occurs: the flag button_held is cleared and sequence_active is set to true. At that exact moment, the system records the current millisecond counter as sequence_start to mark the beginning of a timed sequence setting some base parameters for our PID Controller. When sequence_active is true, the ISR uses the elapsed time (the difference between the current time and sequence_start) to update the desired_angle following a predefined sequence of setpoints (for example, horizontal for the first second, then a specified angle above horizontal, etc.). This state machine ensures that the button press–release event triggers an automatic, time‑based change in the beam’s setpoint. Once the sequence completes, control reverts to the idle state until another button press is detected.

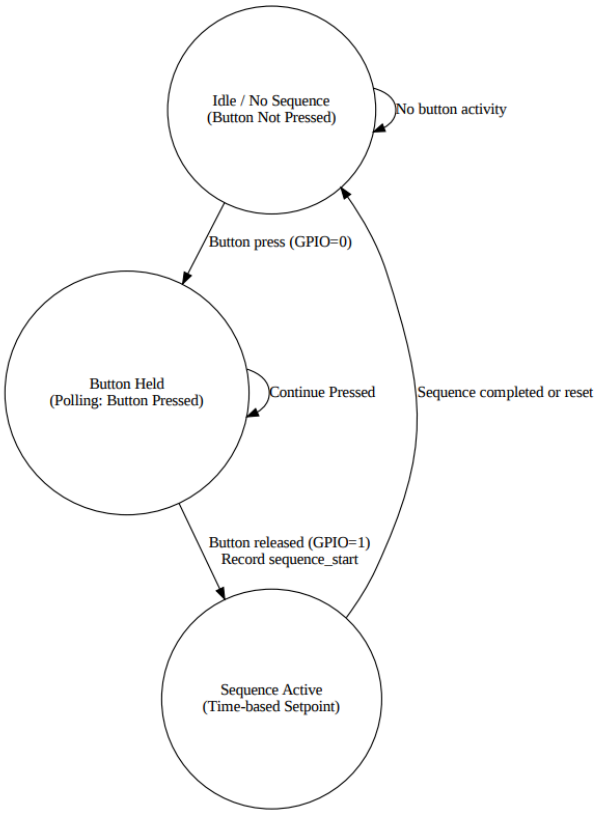

Figure 6: Diagram of Intended FSM For Button Logic

Motor Control Board Overview

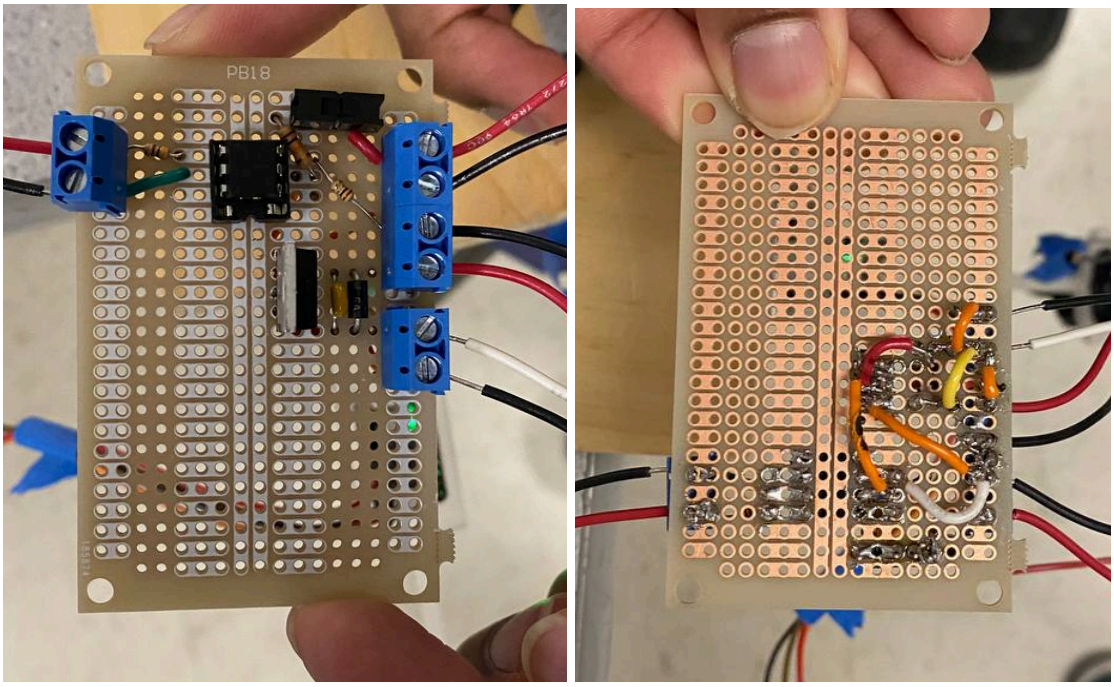

Figure 7: Implementation of Motor Controller

This motor controller is designed to drive a DC motor at higher currents and voltages than a microcontroller pin can handle directly. The schematic below shows the intended layout using a 4N35 optocoupler and an N-channel MOSFET, along with a kill switch circuit. One of the key goals is to provide electrical isolation between the low-voltage control signals (from the microcontroller) and the higher-voltage motor supply. The circuit achieves this by driving the LED side of the 4N35 with a PWM signal from the microcontroller; the phototransistor side of the 4N35, which is physically isolated, switches the gate of the MOSFET. As a result, the microcontroller is protected from voltage spikes or other high-current events on the motor side. The kill switch is placed in series with the motor's power or the MOSFET's supply lines, allowing a user to cut off the motor power safely in case of emergency or while making system changes.

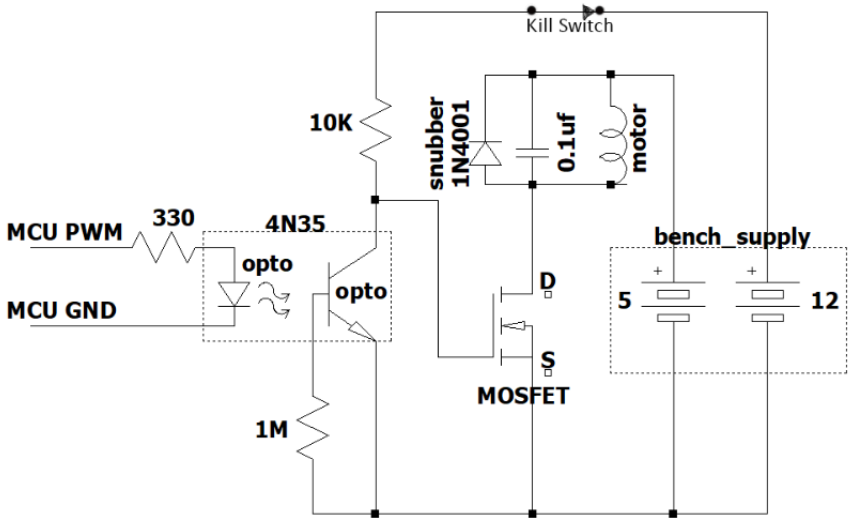

Figure 8: Electrical Schematic for Motor Controller

In the schematic, the bench supply provides around 12 V to the motor, and the MOSFET is configured as a low-side switch: the motor's positive terminal is connected to the 12 V supply, and the motor's negative terminal goes to the drain (D) of the MOSFET. The source (S) of the MOSFET is connected to ground. A 1 MΩ resistor from gate to source is often shown, ensuring the gate remains at ground potential when the circuit is unpowered or the optocoupler transistor is off. This avoids a floating gate, which could accidentally turn the MOSFET on. There is also a protective diode across the motor or within the circuit to deal with inductive voltage spikes generated when the motor is switched off. The optocoupler (4N35), the MOSFET, and the kill switch all work together to yield a compact driver board capable of switching sizable currents at the motor while keeping the microcontroller fully isolated.

Safety Measures of the Motor Controller

Putting the circuit in a general safety perspective, it's important to consider how the system integrates as a whole to address several key safety questions.

A capacitor is placed across the motor supply to help filter high-frequency noise generated by rapid switching of the motor, thus preventing voltage spikes from propagating into the rest of the system. A current limiting resistor is used to restrict the current either through the input LED of the optocoupler or into the MOSFET gate, ensuring that neither the LED nor the switching transistor is driven beyond safe operating limits. The use of optical isolation (via a 4N35 or similar optocoupler) provides a barrier between the microcontroller's low-voltage domain and the higher-power motor circuit, protecting sensitive control electronics from large voltage transients or grounding issues that may occur on the motor side. Finally, a diode is typically placed in parallel with the inductive load (the motor) to clamp high-voltage spikes when the motor current is abruptly interrupted; this diode dissipates the inductive energy safely and prevents damaging voltage surges across the switching transistor.

Together, these components—capacitor, resistor, optocoupler, and diode—significantly enhance the controller's overall safety, reliability, and noise immunity.

Debugging and Testing Strategies

To draw the graphs on the VGA screen, we use a step-by-step approach of implementing changes to the code and looking at the result - it can be tricky to correctly align all elements of the screen otherwise. To tune th ePID controller, we were looking at the overshoot, settling time, and stability metrics - mostly visual. A high overshoot of the arm when moving to a new position indicates that damping needs to be

increased or proportional gain needs to be decreased. High proportional and integral gains gave us faster settling times.
We also noticed that our accelerometer seemed to have an angle offset - it did not output 0 degrees in the 0-degree configuration, so we had to add a value of 17 degrees to it to calibrate the readings.

Appendix:

```
/**
 * V. Hunter Adams (vha3@cornell.edu)
 *
 * This demonstration utilizes the MPU6050.
 * It gathers raw accelerometer/gyro measurements, scales
 * them, and plots them to the VGA display. The top plot
 * shows gyro measurements, bottom plot shows accelerometer
 * measurements.
 *
 * HARDWARE CONNECTIONS
 *  - GPIO 16 ---> VGA Hsync
 *  - GPIO 17 ---> VGA Vsync
 *  - GPIO 18 ---> 330 ohm resistor ---> VGA Red
 *  - GPIO 19 ---> 330 ohm resistor ---> VGA Green
 *  - GPIO 20 ---> 330 ohm resistor ---> VGA Blue
 *  - RP2040 GND ---> VGA GND
 *  - GPIO 8 ---> MPU6050 SDA
 *  - GPIO 9 ---> MPU6050 SCL
 *  - 3.3v ---> MPU6050 VCCf
 *  - RP2040 GND ---> MPU6050 GND
 */


// Include standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
// Include PICO libraries
#include "pico/stdlib.h"
#include "pico/multicore.h"
```

```
// Include hardware libraries
#include "hardware/pwm.h"
#include "hardware/dma.h"
#include "hardware/irq.h"
#include "hardware/adc.h"
#include "hardware/pio.h"
#include "hardware/i2c.h"
// Include custom libraries
#include "vga16_graphics.h"
#include "mpu6050.h"
#include "pt_cornell_rp2040_v1_3.h"

// PUSH BUTTON DEF

#define PUSH_BUTTON 15

// Arrays in which raw measurements will be stored
fix15 acceleration[3], gyro[3];

// character array
char screentext[40];

// draw speed
int threshold = 10 ;

// Some macros for max/min/abs
#define min(a,b) ((a<b) ? a:b)
#define max(a,b) ((a<b) ? b:a)
#define abs(a) ((a>0) ? a:-a)

// semaphore
static struct pt_sem vga_semaphore ;

// Some paramters for PWM
#define WRAPVAL 5000
#define CLKDIV  25.0f
uint slice_num ;

// GPIO we're using for PWM
#define PWM_OUT 4
```

```
// PWM duty cycle
volatile int control ;
volatile int old_control;
volatile int control_lp;


// Arrays in which raw measurements will be stored
fix15 acceleration[3], gyro[3];
float current_angle = 0.0;    // Measured angle (e.g., from accelerometer)
float last_angle = 0.0;
float desired_angle = 8.5;    // Setpoint

fix15 gyro_angle_delta;
fix15 accel_angle;
fix15 complementary_angle;


//Global variables for PID
volatile float Kp = 0.0f;
volatile float Ki = 0.0f;
volatile float Kd = 0.0f;

// 60,25,6000,0.025
//100,45,6000,0.005
// x,35,8000,0.005

// want Kp ~ 34, Kd ~ 3000,

volatile float error = 0.0f;
volatile float last_error = 0.0f;
volatile float integral = 0.0f;
volatile float derivative = 0.0f;

 // For timing-based sequence
 volatile uint32_t ms_count       = 0; // increments ~1 kHz in ISR
 volatile bool     button_held    = false;
 volatile bool     sequence_active = false;
 volatile uint32_t sequence_start  = 0;
```

```
// Interrupt service routine
void on_pwm_wrap() {

    // Clear the interrupt flag that brought us here
    // pwm_clear_irq(pwm_gpio_to_slice_num(5));
    pwm_clear_irq(pwm_gpio_to_slice_num(PWM_OUT));


    ms_count++;
// ----------------- BUTTON FSM ----------------------

    bool button_state = (gpio_get(PUSH_BUTTON)==0);

    if (button_state){
        // case that the button is pressed
        button_held = true;
        sequence_active = false; // indicates to terminate the running
programs
        control = 0; // turns the motor off
        //force the immediate update in this case --> could use any val
for old control
        old_control = -1;
        pwm_set_chan_level(slice_num, PWM_CHAN_A, 0);
    }
    else {
        // Button is not pressed
        if(button_held){
            //transition to the pressed to not pressed conditions
            button_held = false;
            sequence_active = true;
            sequence_start = ms_count;
            Kp = 36;
            Kd = 8000;
            Ki = 0.0065;
        }
    }

    if(sequence_active){
```

```
        //when we set this case --> our trigger to rewrite the desired
angles
        uint32_t t = ms_count - sequence_start; // use this t val to drive
the conditional updates

          // Desired Sequence:
         //  0..1s   => set angle to horizontal
         //  1..5s   => keep horizontal
         //  5..10s  => 30 deg above
         //  10..15s => 30 deg below
         //  15s+    => back to horizontal
        if (t < 100) {
            desired_angle = 30.0f;
        }
        else if (t < 200) {
            desired_angle = 40.0f;
        }
        else if (t < 300) {
            desired_angle = 50.0f;
        }
        else if (t < 400) {
            desired_angle = 60.0f;
        }
        else if (t < 1000) {
            desired_angle = 90.0f;   // horizontal
        }
        else if (t < 5000) {
            desired_angle = 90.0f;   // still horizontal
        }
        else if (t < 10000) {
            desired_angle = 120.0f;  // 30 above
        }
        else if (t < 15000) {
            desired_angle = 60.0f; // 30 below
        }
        else if (t> 20000){
            sequence_active = false;
        }
        else {
            desired_angle = 90.0f;   // back to horizontal
```

```
            // maybe stop sequence ??
            // sequence_active = false;
        }

    }


    // Read the IMU
    // NOTE! This is in 15.16 fixed point. Accel in g's, gyro in deg/s
    // If you want these values in floating point, call fix2float15() on
    // the raw measurements.
    mpu6050_read_raw(acceleration, gyro);
    // float gyro_f = fix2float15(gyro[2]);
    gyro_angle_delta = multfix15(gyro[0], zeropt001) ; //HERE changed to
gyro 0
    accel_angle = multfix15(float2fix15(atan2(acceleration[2],
-acceleration[1])), oneeightyoverpi);

    // current_angle = 7+180/M_PI * atan2(fix2float15(acceleration[2]),
-fix2float15(acceleration[1]));

    // ADJUST COEFFICIENTS HERE FOR FASTER RESPONSE?
    complementary_angle = multfix15(complementary_angle +
gyro_angle_delta, zeropt999) + multfix15(accel_angle, zeropt001); //PLUS
or MINUS gyro angle
    current_angle = fix2float15(complementary_angle) + 17.0;

    float dt = 0.001;

    //alpha filter
    float alpha = 0.5;
    current_angle = alpha*current_angle + (1-alpha)*(last_angle);
    // printf("%f", current_angle);

    error = desired_angle-current_angle;
    integral = integral + error;
    derivative = (error - last_error); //HERE
    if (abs(integral) > 1000000) {
        integral = 20000;
    }
```

```
    control = Kp*(error) + Kd*derivative + Ki * integral;;

    //HERE TO LOW PASS CONTROL
    control_lp = control_lp + ((control - control_lp)>>4) ;
    control = control_lp;

    // limit the control value
    if (control > 4700){
        control = 4700;
    }
    else if (control < 0 ){
        control = 0;
    }

    if (control!=old_control) {
        old_control = control ;
        pwm_set_chan_level(slice_num, PWM_CHAN_A, control);
    }


    last_angle = current_angle;
    last_error = error;

    // Signal VGA to draw
    PT_SEM_SIGNAL(pt, &vga_semaphore);
}

// Thread that draws to VGA display
static PT_THREAD (protothread_vga(struct pt *pt))
{
    // Indicate start of thread
    PT_BEGIN(pt) ;

    // We will start drawing at column 81
    static int xcoord = 81 ;

    // Rescale the measurements for display
    static float OldRange = 500. ; // (+/- 250)
    static float NewRange = 100. ; // (looks nice on VGA)
```

```
  static float OldMin = 0. ;
  static float OldMax = 250. ;

  int plot_height = 100;

  // Control rate of drawing
  static int throttle ;

  // Draw the static aspects of the display
  setTextSize(1) ;
  setTextColor(WHITE);

  // Draw bottom plot
  drawHLine(75, 370, 5, CYAN) ;
  drawHLine(75, 370+plot_height/2, 5, CYAN) ;
  drawHLine(75, 370+plot_height, 5, CYAN) ;
  drawVLine(80, 370, plot_height, CYAN) ;
  sprintf(screentext, "90") ;
  setCursor(50, 420) ;
  writeString(screentext) ;
  sprintf(screentext, "+180") ;
  setCursor(50, 370) ;
  writeString(screentext) ;
  sprintf(screentext, "0") ;
  setCursor(50, 470) ;
  writeString(screentext) ;

  // Draw top plot
  drawHLine(75, 90, 5, CYAN) ;
  drawHLine(75, 90+plot_height/2, 5, CYAN) ;
  drawHLine(75, 90+plot_height, 5, CYAN) ;
  drawVLine(80, 90, plot_height, CYAN) ;
  sprintf(screentext, "100") ;
  setCursor(45, 90) ;
  writeString(screentext) ;
  sprintf(screentext, "0") ;
  setCursor(45, 190) ;
  writeString(screentext) ;

  // Draw middle plot
```

```
    drawHLine(75, 240, 5, CYAN) ;
    drawHLine(75, 240+plot_height/2, 5, CYAN) ;
    drawHLine(75, 240+plot_height, 5, CYAN) ;
    drawVLine(80, 240, plot_height, CYAN) ;
    sprintf(screentext, "+250") ;
    setCursor(45, 240) ;
    writeString(screentext) ;
    sprintf(screentext, "-250") ;
    setCursor(45, 340) ;
    writeString(screentext) ;
    sprintf(screentext, "0") ;
    setCursor(50, 290) ;
    writeString(screentext) ;

    while (true) {

        // Wait on semaphore
        PT_SEM_WAIT(pt, &vga_semaphore);
        // Increment drawspeed controller
        throttle += 1 ;
        // If the controller has exceeded a threshold, draw
        if (throttle >= threshold) {
            // Zero drawspeed controller
            throttle = 0 ;

            // Erase a column
            drawVLine(xcoord, 0, 490, BLACK) ;

            // Draw bottom plot (multiply by 120 to scale from +/-2 to
+/-250)
            drawPixel(xcoord, 420 -
(int)((current_angle)*plot_height/180/2), GREEN) ;
            drawHLine(81, 420, 528, WHITE);
            sprintf(screentext, "Current Angle:") ;
            setCursor(5, 352) ;
            writeString(screentext) ;


            // Draw middle plot
```

```
            drawPixel(xcoord, 290 -
(int)(NewRange*((float)((fix2float15(gyro[0]))-OldMin)/OldRange)), GREEN)
;
            drawHLine(81, 290, 528, WHITE);
            sprintf(screentext, "Gyro:") ;
            setCursor(5, 220) ;
            writeString(screentext) ;

            //Draw top plot HERE
            drawPixel(xcoord, 190 - (int)(control/50), GREEN) ;
            drawHLine(81, 190, 528, WHITE);
            sprintf(screentext, "Duty Cycle:") ;
            setCursor(5, 70) ;
            writeString(screentext) ;

            // Update horizontal cursor
            if (xcoord < 609) {
                xcoord += 1 ;
            }
            else {
                xcoord = 81 ;
            }

            setTextColor2(WHITE, BLACK);
            float toprint = control/50;
            snprintf(screentext, 6, "%f", toprint);
            setCursor(450, 20) ;
            writeString(screentext) ;

             //HERE
            toprint = Kp;
            snprintf(screentext, 6, "%f", toprint);
            setCursor(450, 30) ;
            writeString(screentext) ;

            toprint = Ki;
            snprintf(screentext, 6, "%f", toprint);
            setCursor(450, 40) ;
            writeString(screentext) ;
```

```
            toprint = Kd;
            snprintf(screentext, 6, "%f", toprint);
            setCursor(450, 50) ;
            writeString(screentext) ;

            toprint = desired_angle;
            snprintf(screentext, 6, "%f", toprint);
            setCursor(450, 60) ;
            writeString(screentext) ;

            toprint = current_angle;
            snprintf(screentext, 6, "%f", toprint);
            setCursor(450, 70) ;
            writeString(screentext) ;

            toprint = sequence_active;
            snprintf(screentext, 6, "%f", toprint);
            setCursor(450, 80) ;
            writeString(screentext) ;




            // Print the parameters  //HERE
            sprintf(screentext, "Duty Cycle in percent:") ;
            setCursor(300, 20) ;
            writeString(screentext);

            sprintf(screentext, "Proportional Gain:") ;
            setCursor(300, 30) ;
            writeString(screentext);

            sprintf(screentext, "Integral Gain:") ;
            setCursor(300, 40) ;
            writeString(screentext);

            sprintf(screentext, "Differential Gain:") ;
            setCursor(300, 50) ;
            writeString(screentext);
```

```
            sprintf(screentext, "Target Angle") ;
            setCursor(300, 60) ;
            writeString(screentext);

            sprintf(screentext, "Current Angle") ;
            setCursor(300, 70) ;
            writeString(screentext);

            sprintf(screentext, "Sequence Active") ;
            setCursor(300, 80) ;
            writeString(screentext);
        }
    }
    // Indicate end of thread
    PT_END(pt);
}

// User input thread. User can change draw speed
static PT_THREAD (protothread_serial(struct pt *pt))
{
    PT_BEGIN(pt) ;
    static char classifier ;

    static int input ;
    static int angle_in;
    static int Kp_in;
    static int Kd_in;
    static float Ki_in;

    static float float_in ;

    while(1) {
        sprintf(pt_serial_out_buffer, "input a desired angle (0-180), Kp,
Ki, Kd as ints in [angle,Kp,Kd,Ki]: ");
        serial_write ;
        // spawn a thread to do the non-blocking serial read
        serial_read ;
        // convert input string to number
        // sscanf(pt_serial_in_buffer,"%d", &input) ;
```

```
        sscanf(pt_serial_in_buffer, "%d,%d,%d,%f", &angle_in, &Kp_in,
&Kd_in, &Ki_in);

        Kp = Kp_in;
        Kd = Kd_in;
        Ki = Ki_in;
        if (angle_in > 180) continue ;
        else if (angle_in < 0) continue ;
        else desired_angle = angle_in ;

    }
    PT_END(pt) ;
}

// Entry point for core 1
void core1_entry() {
    pt_add_thread(protothread_vga) ;
    pt_schedule_start ;
}

int main() {

    // Initialize stdio
    stdio_init_all();

    // Initialize VGA
    initVGA() ;


////////////////////////////////////////////////////////////////////////
    ////////////////////////// I2C CONFIGURATION
/////////////////////////////
    i2c_init(I2C_CHAN, I2C_BAUD_RATE) ;
    gpio_set_function(SDA_PIN, GPIO_FUNC_I2C) ;
    gpio_set_function(SCL_PIN, GPIO_FUNC_I2C) ;

    // Pullup resistors on breakout board, don't need to turn on internals
    // gpio_pull_up(SDA_PIN) ;
    // gpio_pull_up(SCL_PIN) ;
```

```
    // MPU6050 initialization
    mpu6050_reset();
    mpu6050_read_raw(acceleration, gyro);


////////////////////////////////////////////////////////////////////////
    ////////////////////////// PWM CONFIGURATION
/////////////////////////////

////////////////////////////////////////////////////////////////////////
    // Tell GPIO's 4,5 that they allocated to the PWM
    gpio_set_function(5, GPIO_FUNC_PWM);
    gpio_set_function(4, GPIO_FUNC_PWM);
    // Tell GPIO PWM_OUT that it is allocated to the PWM
    gpio_set_function(PWM_OUT, GPIO_FUNC_PWM);

    // Find out which PWM slice is connected to GPIO 5 (it's slice 2, same
for 4)
    slice_num = pwm_gpio_to_slice_num(PWM_OUT);

    // Mask our slice's IRQ output into the PWM block's single interrupt
line,
    // and register our interrupt handler
    pwm_clear_irq(slice_num);
    pwm_set_irq_enabled(slice_num, true);
    irq_set_exclusive_handler(PWM_IRQ_WRAP, on_pwm_wrap);
    irq_set_enabled(PWM_IRQ_WRAP, true);

    // This section configures the period of the PWM signals
    pwm_set_wrap(slice_num, WRAPVAL) ;
    pwm_set_clkdiv(slice_num, CLKDIV) ;

    pwm_set_output_polarity (slice_num, 1, 0);
    // This sets duty cycle
    pwm_set_chan_level(slice_num, PWM_CHAN_B, 0);
    pwm_set_chan_level(slice_num, PWM_CHAN_A, 0);

    // Start the channel
    pwm_set_mask_enabled((1u << slice_num));
```

```
    // ADDED: Configure button input
    gpio_init(PUSH_BUTTON);
    gpio_set_dir(PUSH_BUTTON, GPIO_IN);
    gpio_pull_up(PUSH_BUTTON);

    // start core 1
    multicore_reset_core1();
    multicore_launch_core1(core1_entry);

    // start core 0
    pt_add_thread(protothread_serial) ;
    pt_schedule_start ;

}
```