Active Wheel Camber Control

ME 102B – Final Project

Gurmehr Klair, Dawood Junaid, Feynman Barney

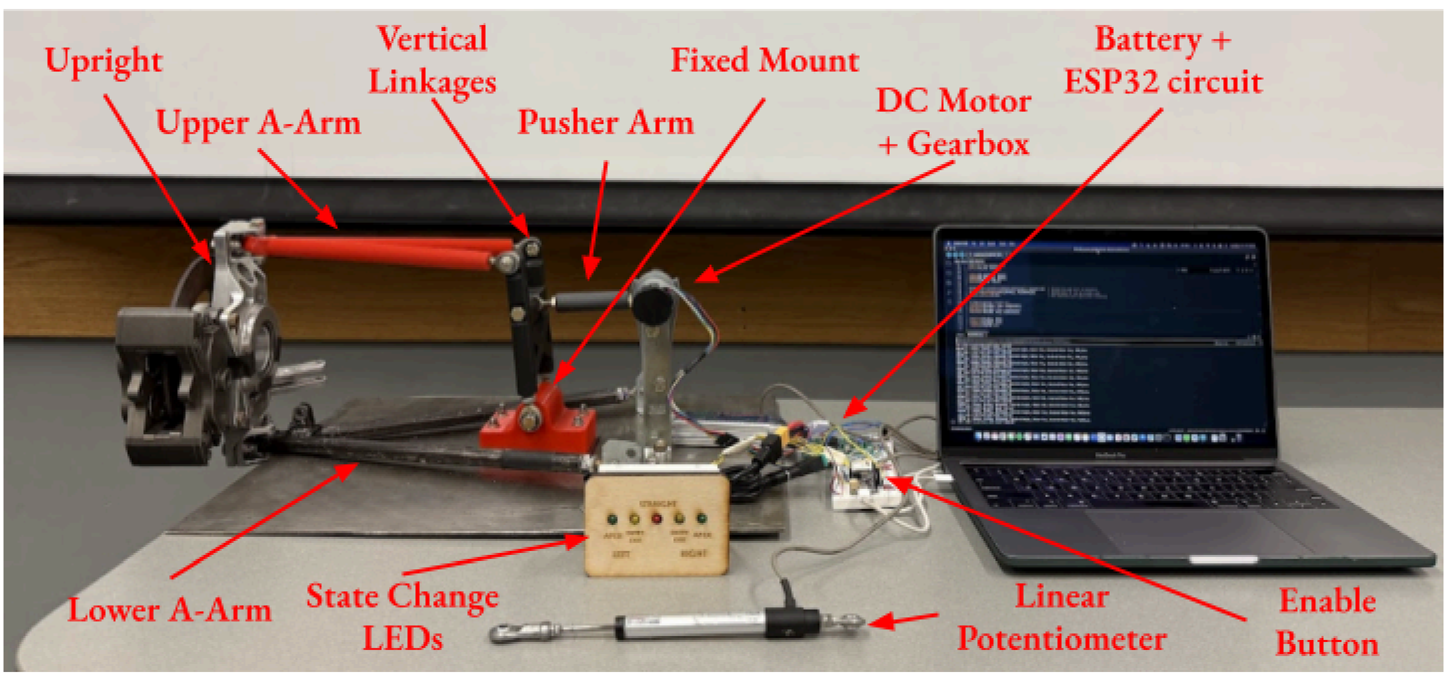
Opportunity

This project is a proof-of-concept mechanism developed with the Berkeley Formula Racing car in mind, to improve vehicle performance by reducing lap times and minimizing tire wear.

High-Level Strategy

We address this opportunity through an active wheel camber control mechanism. When viewed from the front or rear, Camber refers to the angle between a vehicle's vertical axis and the wheel. The ideal camber for straight-line acceleration is 0º to maximize the tire's longitudinal grip. However, some non-zero camber is required when the car is turning to compensate for the car rolling due to lateral load transfer. To maximize the tire's contact patch with the ground during a turn, negative camber is needed for the outside tire, while positive camber is necessary for the inside tire. Active camber control enables the individual tires to maintain optimal camber angles throughout different driving conditions on track, and this dynamic optimization allows the driver to navigate corners more quickly without compromising straight-line driving while evenly wearing the tires, ultimately improving overall performance. The original desired functionality was to control camber within the range of -5 and 5º within a half second of the steering input updating, and we achieved -6 and 6º within 0.8 seconds.

Integrated Physical Device

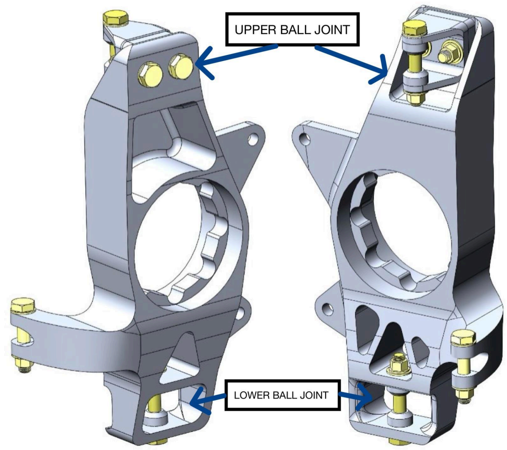


Function-Critical Decisions

The first step in function-critical calculations was a script that takes into consideration the different loads that the upright experiences in a real-life situation based on accelerometer data from the Formula SAE car. By taking moments and doing a force analysis, we were able to simplify the model to two compound cases that cause the greatest loads on our upper ball joint.

1. **Front_Compound_1:** Tight cornering with full-force braking.
2. **Front_Compound_2:** Tightest corner in FSAE competition combined with a worst-case bump.

The loads were applied in simulations to analyze force distribution between the **Upper Ball Joint (UBJ)** and **Lower Ball Joint (LBJ)**. Next, we used the maximum forces in each direction from the two different loadcases and used those forces for all further calculations.



To actuate the pivoting motion of the upright about the LBJ and counter forces acting on the UBJ, a motor was required to generate sufficient torque while meeting design constraints such as cost, efficiency, and response time. To achieve the required torque and speed, we selected a 12V brushed DC motor with a 131.25:1 metal gearbox and encoder with a resolution of 64 counts per revolution. It has a stalling torque of 45 kg-cm (4.41 Nm or 38.99 in-lbf). For the system to actuate quickly enough while the car is turning, we require the motor to turn 12 degrees in 0.5 seconds, or 4RPM. At this speed, the motor produces 34.76in-lbf. To ensure a load that is maximum 60% of the stalling torque, the motor should only provide 23.4in-lbf for actuation.

Force at Actuation Point (A)

Distance from pivot to actuation point = 0.8 in
Torque applied at actuation point = 23.4 in-lbf

$$F_A = \frac{\text{Torque}}{\text{Lever Arm}} = \frac{23.4\,\text{in-lbf}}{0.8\,\text{in}} = 29.25\,\text{lbf}$$

Force at Upper Ball Joint (Point C)

Distance from LBJ to UBJ = 6.5 in
Transfer arm distance = 3.94 in

$$F_C = \frac{F_A \times \text{Transfer Arm}}{\text{Lever Arm}} = \frac{29.25\,\text{lbf} \times 3.94\,\text{in}}{6.5\,\text{in}} = 17.72\,\text{lbf}$$

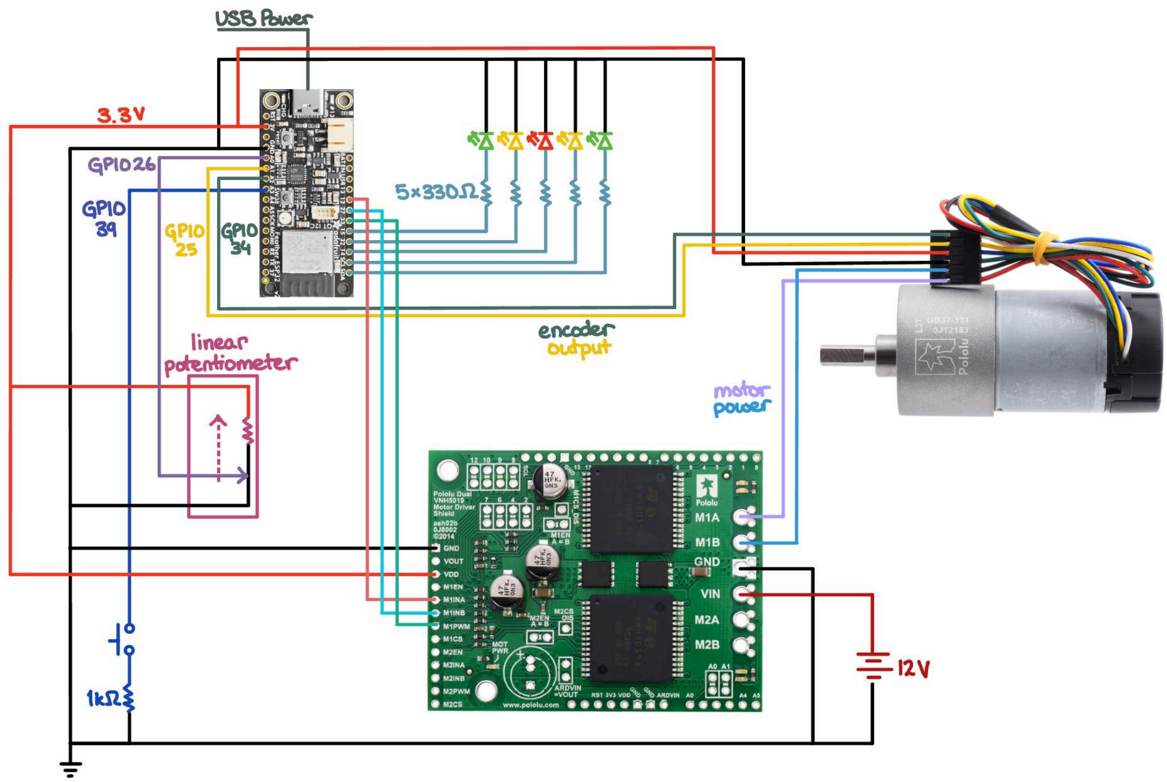
The simulated forces acting on the UBJ were resolved into a moment about the LBJ, providing the required torque. Based on how much force the motor can apply on the UBJ, we scaled down the force by a factor of 10, so that our effective lateral force is 12.2lbf while the motor applied 17.72lbf onto the UBJ with a 5.48lbf net force causing the upright to pivot along the fixed LBJ point. The forces in the other two directions will be resolved and triangulated onto the fixed points through the tubes.

We then were able to finalize material selection, concluding that the lower a-arm, rod-ends and mounts should be metal while the rest of the components could be 3D printed. Utilizing the new loads, we ran a simple FEA simulation to ensure that the parts were under a 3.5 factor of safety, due to the unreliable material properties of 3D-printed parts.

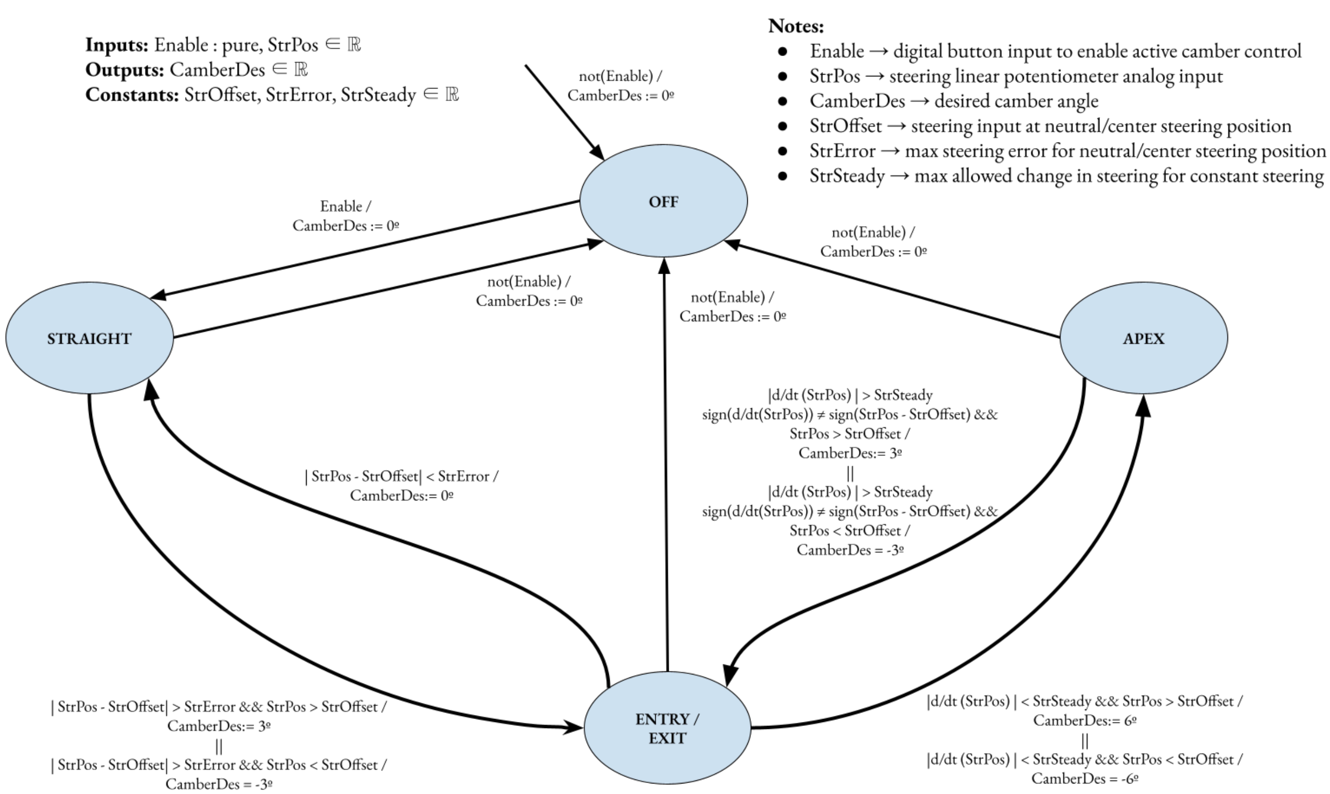
Reflection

In the initial stages of the project, our team was very excited about the project idea due to its novelty and potential to significantly improve the car's performance. However, the project turned out to be much more work than originally anticipated, and our initial mechanism design did not work out. We were able to adapt the project idea to simplify the components and manufacturing required, but it was an ongoing process to balance our ambitions for the project with time and cost constraints. If we were to do this project again, we would have started with a simpler prototype in the initial design phase and worked more closely during the integration phase.

Circuit Diagram



State Transition Diagram



Front-Left Corner of the Car

- Steering left → steering input greater than StrOffset, inside tire
- Steering right → steering input less than StrOffset, outside tire

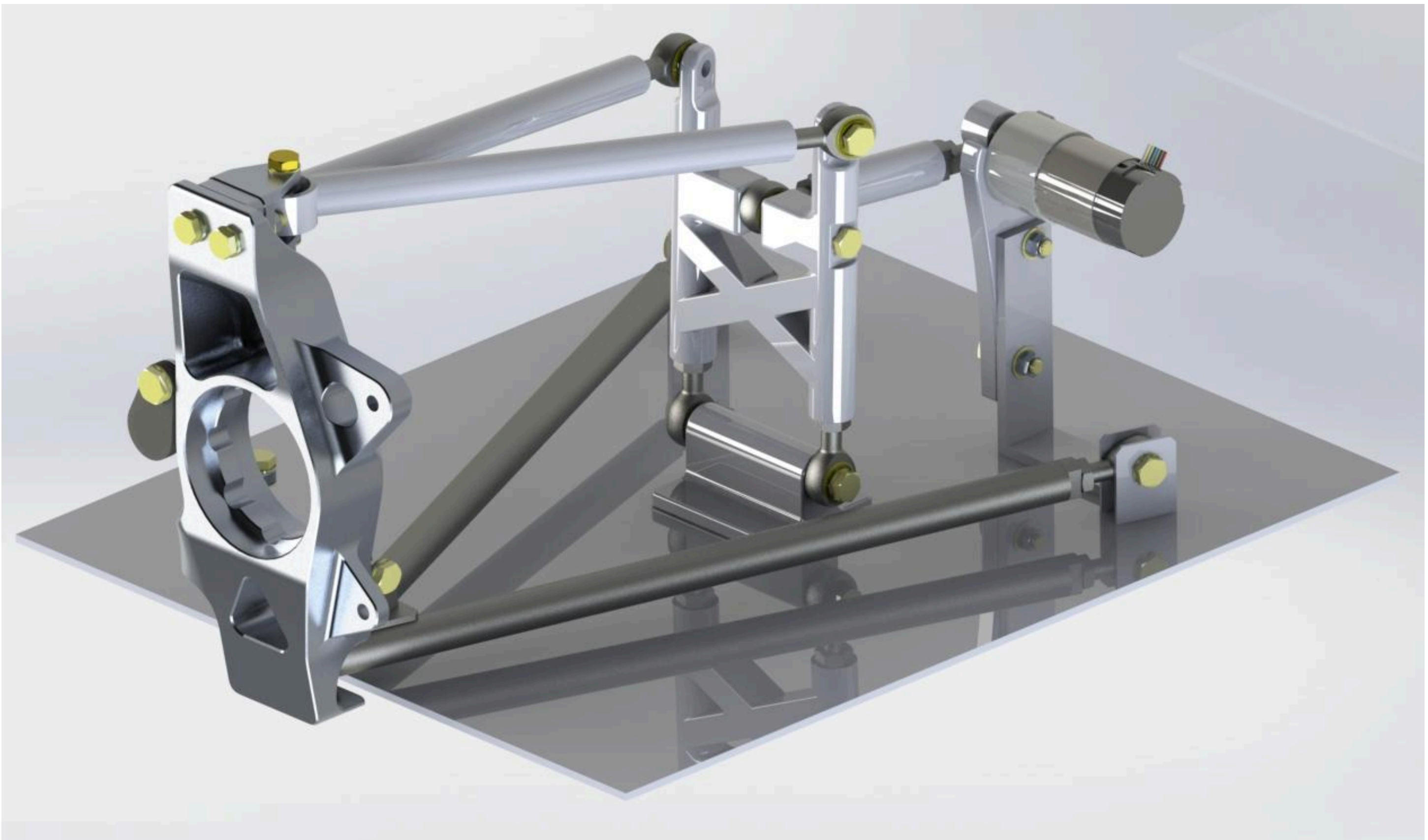
Appendix

Bill of Materials

Required Parts	Material/ Listing Name	Quantity	Cost	Supplier
Hardware				
Filament for Upper A-Arms	Standard Bambu Labs PLA	66.69 cc	$ 1.67	Amazon
Filament for Vertical Linkage & Mount	Standard Bambu Labs PLA	174.59 cc	$ 5.24	Amazon
Filament for Motor Linkage & Mount	Standard Bambu Labs PLA	37.7 cc	$ 0.94	Amazon
Filament for Pushbar	Standard Bambu Labs PLA	5 cc	$ 0.15	Amazon
Upright	Aluminum CNC'd Suspension Upright	1	$ -	Berkeley Formula Racing
Tubes for Lower A-Arms	Welded Aluminum Tubing	1 set	$ -	Berkeley Formula Racing
Ball Joints		2	$ -	Berkeley Formula Racing
AN3 Bolts		14	$ 24.00	Amazon
AN3 Washers		28	$ 2.40	McMaster-Carr
AN3 KNuts		14	$ 28.00	McMaster-Carr
Rod Ends		8	$ -	Berkeley Formula Racing
L-brackets	Various sizes for mounting	3	$ -	Berkeley Formula Racing
Mounting Plate	1/8" Steel Plate	1	$ -	Berkeley Formula Racing
Electronics				
Motor	131:1 Metal Gearmotor 37Dx73L mm 12V with 64 CPR Encoder (Helical Pinion)	1	$ 51.95	Pololu
Motor Driver	Pololu Dual VNH5019 Motor Driver Shield for Arduino	1	$ 59.95	Pololu
2 Pack Battery & Charger	2 Pack XT60 Plug 12V 3S 2200mAh Lipo Battery and 1 Charger 12V 2200mAh XT60 2	1	$ 39.99	Amazon
Battery Adapter	XT60 to DC5521 Power Cable, XT-60 Male to DC 5.5mm x 2.1mm Male Adapter Cord	1	$ 9.99	Amazon
Linear Potentiometer	Texense Pulling Rod Linear Sensor	1	$ -	Berkeley Formula Racing
ESP32	Huzzah Feather V2 ESP32	1	$ -	Lab Kit
Button	Push Button	1	$ -	Lab Kit
Resistors	330 & 1k Ohm resistors	6	$ -	Lab Kit

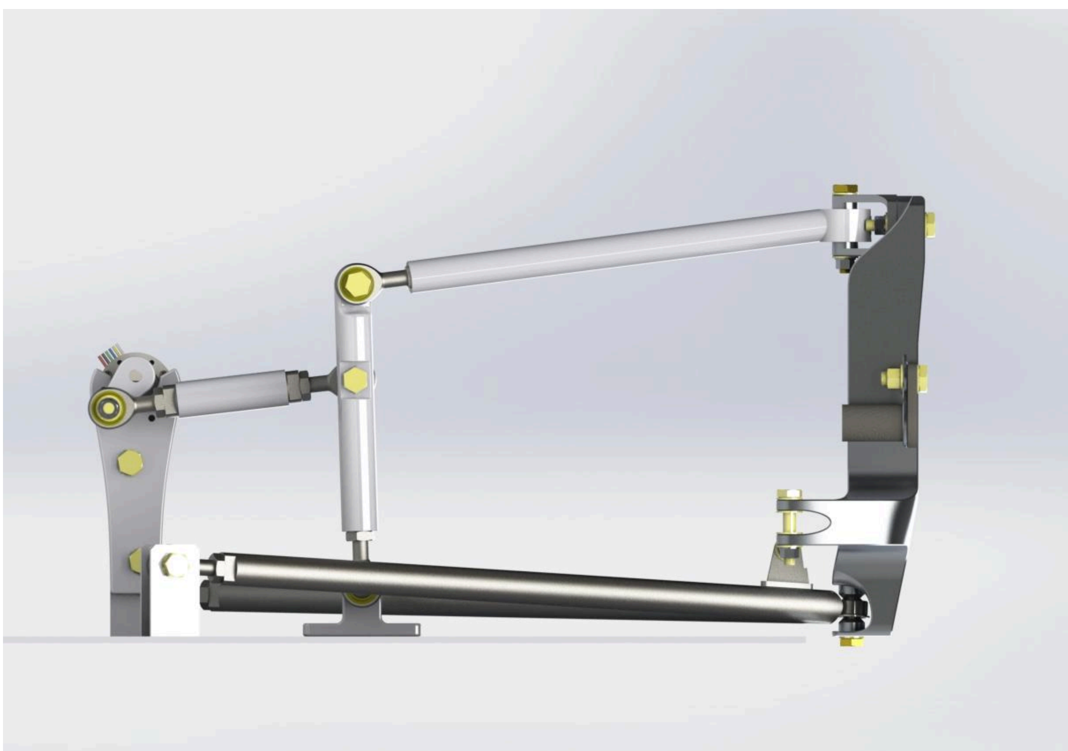
LEDs	Red, yellow, green LEDs	5	$ -	Lab Kit
Wires	Various	-	$ -	Lab Kit
Total Cost			**$ 224.28**	

CAD Images



Isometric View

Top View



Front View

Code

ACC_Code_102B_Final.ino

```
#include <Arduino.h>
#include <ESP32Encoder.h>

#define M1_IN_A 12
#define M1_IN_B 27
#define M1_PWM 33
#define M1_SENSOR_A 25
#define M1_SENSOR_B 34
#define STR_POS 26
#define BTN 39
#define LED_RIGHT_GREEN 15
#define LED_RIGHT_YELLOW 32
#define LED_STRAIGHT_RED 14
#define LED_LEFT_YELLOW 20
#define LED_LEFT_GREEN 22

ESP32Encoder encoder;

// State Machine States
enum State {
  OFF,
  STRAIGHT,
  ENTRY_EXIT,
  APEX
} currentState = OFF;
bool ACC_Enable = false;

// Steering Constants & Variables
const int straightOffset = 2047;
const int maxStraightError = 750;
const double maxSteadyError = 70;
int strPos = 0;
const int windowSize = 5;
int strValues[windowSize];
double strSpeed = 0;

// Desired Camber Angles
double straightCamber = 0;
double eLeft = 3;
double eRight = -3;
double apexLeft = 6;
double apexRight = -6;
double camberAngle = straightCamber;

// Motor Variables
const int thetaMax = 8400;  // encoder counts per 1 revolution of output shaft
int a = -120;           // encoder ticks per angle // TODO
int b = 2048;          // encoder ticks for zero  // TODO
int theta = b;
int thetaDes = 0;
int error = 0;
int sumError = 0;
int lastError = 0;

double Kp = 0.4;  // TODO
double Ki = 0.05;
double Kd = -0.05;
double X;

// Interrupt Variables
```

ACC_Code_102B_Final.ino

```
volatile bool buttonPressed = false;
int debounceDelay = 500;                 // ms
unsigned long lastButtonPressTime = 0;  // To track button press time

volatile int count = 0;
volatile bool deltaT = false;  // check timer interrupt
hw_timer_t* timer0 = NULL;
portMUX_TYPE timerMux0 = portMUX_INITIALIZER_UNLOCKED;
const int timerFreq = 1000000;  // 1 us / 1MHz
const int alarmFreq = 100000;   // 10,000 * 1 us = 10 ms

// PWM Properties
const int freq = 5000;
const int resolution = 8;
const int MAX_PWM_VOLTAGE = 255;
const int NOM_PWM_VOLTAGE = 150;

// Interrupt Services
void IRAM_ATTR isr_btn() {  // the function to be called when interrupt is triggered
  unsigned long currentMillis = millis();
  if (currentMillis - lastButtonPressTime >= debounceDelay) {
    buttonPressed = true;
    lastButtonPressTime = currentMillis;
  }
}

void IRAM_ATTR onTime0() {
  portENTER_CRITICAL_ISR(&timerMux0);
  count = encoder.getCount();
  encoder.clearCount();
  deltaT = true;  // the function to be called when timer interrupt is triggered
  portEXIT_CRITICAL_ISR(&timerMux0);
}

void setup() {
  // Put your setup code here, to run once:
  Serial.begin(115200);

  pinMode(M1_IN_A, OUTPUT);
  pinMode(M1_IN_B, OUTPUT);
  digitalWrite(M1_IN_A, LOW);  // Set initial state of motor to off
  digitalWrite(M1_IN_B, LOW);
  pinMode(M1_PWM, OUTPUT);

  pinMode(M1_SENSOR_A, INPUT);
  pinMode(M1_SENSOR_B, INPUT);
  pinMode(STR_POS, INPUT);

  ESP32Encoder::useInternalWeakPullResistors = puType::up;  // Enable the weak pull up resistors
  encoder.attachHalfQuad(M1_SENSOR_A, M1_SENSOR_B);         // Attach pins for use as encoder pins
  encoder.setCount(2048);                                   // Set starting count value after attaching

  // Attach channel to GPIO pin
  ledcAttach(M1_IN_A, freq, resolution);
  ledcAttach(M1_IN_B, freq, resolution);
  ledcAttach(M1_PWM, freq, resolution);

  ledcWrite(M1_IN_A, LOW);
  ledcWrite(M1_IN_B, LOW);
  ledcWrite(M1_PWM, 0);
```

ACC_Code_102B_Final.ino

```

  // Set up button
  attachInterrupt(BTN, isr_btn, RISING);

  // Initilize timers
  timer0 = timerBegin(timerFreq);          // timer 0
  timerAttachInterrupt(timer0, &onTime0);  // edge (not level) triggered
  timerAlarm(timer0, alarmFreq, true, 0);  // autoreload enabled, infinite reloads

  // Initialize steering values array
  for (int i = 0; i < windowSize; i++) {
    strValues[i] = 0;
  }

  // Initialize LED pins
  pinMode(LED_RIGHT_GREEN, OUTPUT);
  pinMode(LED_RIGHT_YELLOW, OUTPUT);
  pinMode(LED_STRAIGHT_RED, OUTPUT);
  pinMode(LED_LEFT_YELLOW, OUTPUT);
  pinMode(LED_LEFT_GREEN, OUTPUT);

  digitalWrite(LED_RIGHT_GREEN, HIGH);
  digitalWrite(LED_RIGHT_YELLOW, HIGH);
  digitalWrite(LED_STRAIGHT_RED, HIGH);
  digitalWrite(LED_LEFT_YELLOW, HIGH);
  digitalWrite(LED_LEFT_GREEN, HIGH);
  delay(2000);

  digitalWrite(LED_RIGHT_GREEN, LOW);
  digitalWrite(LED_RIGHT_YELLOW, LOW);
  digitalWrite(LED_STRAIGHT_RED, LOW);
  digitalWrite(LED_LEFT_YELLOW, LOW);
  digitalWrite(LED_LEFT_GREEN, LOW);
}

void loop() {
  // put your main code here, to run repeatedly:
  if (deltaT) {
    portENTER_CRITICAL(&timerMux0);
    deltaT = false;
    portEXIT_CRITICAL(&timerMux0);

    // Get steering angle and update array of last 10 steering values
    strPos = analogRead(STR_POS);
    for (int i = 0; i < (windowSize - 1); i++) {
      strValues[i] = strValues[i + 1];
    }
    strValues[windowSize - 1] = strPos;
    updateSteeringSpeed();

    // State Machine Logic
    switch (currentState) {
      case OFF:
        camberAngle = straightCamber;
        if (CheckForButtonPress()) {
          currentState = STRAIGHT;
        }
        setLEDPIN(0);
        break;
      case STRAIGHT:
```

ACC_Code_102B_Final.ino

```
        if (!CheckIfStraight()) {
          camberAngle = (TurningLeft()) ? eLeft : eRight;
          currentState = ENTRY_EXIT;
        }
        if (CheckForButtonPress()) {
          currentState = OFF;
        }
        setLEDPIN(3);
        break;
      case ENTRY_EXIT:
        if (SteeringSteady()) {
          camberAngle = (TurningLeft()) ? apexLeft : apexRight;
          currentState = APEX;
        }
        if (CheckIfStraight()) {
          camberAngle = straightCamber;
          currentState = STRAIGHT;
        }
        if (CheckForButtonPress()) {
          currentState = OFF;
        }
        if (TurningLeft()) {
          setLEDPIN(4);
        } else {
          setLEDPIN(2);
        }
        break;
      case APEX:
        if (!SteeringSteady() && ReturningToStraight()) {
          camberAngle = (TurningLeft()) ? eLeft : eRight;
          currentState = ENTRY_EXIT;
        }
        if (CheckForButtonPress()) {
          currentState = OFF;
        }
        if (TurningLeft()) {
          setLEDPIN(5);
        } else {
          setLEDPIN(1);
        }
        break;
    }

    controlMotor();

    plotControlData();
  }
}

// Calculate Steering Speed
void updateSteeringSpeed() {
  double sum = 0;
  for (int i = 0; i < (windowSize - 1); i++) {
    sum += double(strValues[i + 1] - strValues[i]) * (timerFreq / alarmFreq);
  }
  strSpeed = sum / (windowSize - 1);
}

// State Machine Checks
bool CheckForButtonPress() {
```

ACC_Code_102B_Final.ino

```
  if (buttonPressed == true) {
    buttonPressed = false;
    return true;
  } else {
    return false;
  }
}

bool CheckIfStraight() {
  return (abs(strPos - straightOffset) < maxStraightError);
}

bool TurningLeft() {
  return (strPos > straightOffset);
}

bool SteeringSteady() {
  updateSteeringSpeed();
  return (abs(strSpeed) < maxSteadyError);
}

bool ReturningToStraight() {
  updateSteeringSpeed();
  if (TurningLeft()) {
    return (strSpeed < 0);
  } else {
    return (strSpeed > 0);
  }
}

void controlMotor() {
  // Update theta from encoder count delta
  theta += count;
  // Convert desired camber angle to desired theta
  thetaDes = camberAngle * a + b;

  // Control
  error = thetaDes - theta;
  sumError += error;

  double P = Kp * error;
  double I = Ki * sumError;
  double D = Kd * (error - lastError);
  X = P + I + D;

  // Clip output & anti-windup
  if (X > MAX_PWM_VOLTAGE) {
    X = MAX_PWM_VOLTAGE;
    sumError -= error;
  } else if (X < -MAX_PWM_VOLTAGE) {
    X = -MAX_PWM_VOLTAGE;
    sumError -= error;
  }

  if (currentState == OFF) {
    ledcWrite(M1_IN_A, LOW);
    ledcWrite(M1_IN_B, LOW);
    ledcWrite(M1_PWM, 0);
    return;
  }
```

ACC_Code_102B_Final.ino

```

  // Map X to motor direction
  if (X > 0) {
    ledcWrite(M1_IN_A, LOW);
    ledcWrite(M1_IN_B, MAX_PWM_VOLTAGE);
    ledcWrite(M1_PWM, X);
  } else if (X < 0) {
    ledcWrite(M1_IN_A, MAX_PWM_VOLTAGE);
    ledcWrite(M1_IN_B, LOW);
    ledcWrite(M1_PWM, -X);
  } else {
    ledcWrite(M1_IN_A, LOW);
    ledcWrite(M1_IN_B, LOW);
    ledcWrite(M1_PWM, 0);
  }
  lastError = error;
}

void setLEDPIN(int led) {
   switch (led) {
      case 0:
        digitalWrite(LED_RIGHT_GREEN, LOW);
        digitalWrite(LED_RIGHT_YELLOW, LOW);
        digitalWrite(LED_STRAIGHT_RED, LOW);
        digitalWrite(LED_LEFT_YELLOW, LOW);
        digitalWrite(LED_LEFT_GREEN, LOW);
        break;
      case 1:
        digitalWrite(LED_RIGHT_GREEN, HIGH);
        digitalWrite(LED_RIGHT_YELLOW, LOW);
        digitalWrite(LED_STRAIGHT_RED, LOW);
        digitalWrite(LED_LEFT_YELLOW, LOW);
        digitalWrite(LED_LEFT_GREEN, LOW);
        break;
      case 2:
        digitalWrite(LED_RIGHT_GREEN, LOW);
        digitalWrite(LED_RIGHT_YELLOW, HIGH);
        digitalWrite(LED_STRAIGHT_RED, LOW);
        digitalWrite(LED_LEFT_YELLOW, LOW);
        digitalWrite(LED_LEFT_GREEN, LOW);
        break;
      case 3:
        digitalWrite(LED_RIGHT_GREEN, LOW);
        digitalWrite(LED_RIGHT_YELLOW, LOW);
        digitalWrite(LED_STRAIGHT_RED, HIGH);
        digitalWrite(LED_LEFT_YELLOW, LOW);
        digitalWrite(LED_LEFT_GREEN, LOW);
        break;
      case 4:
        digitalWrite(LED_RIGHT_GREEN, LOW);
        digitalWrite(LED_RIGHT_YELLOW, LOW);
        digitalWrite(LED_STRAIGHT_RED, LOW);
        digitalWrite(LED_LEFT_YELLOW, HIGH);
        digitalWrite(LED_LEFT_GREEN, LOW);
        break;
      case 5:
        digitalWrite(LED_RIGHT_GREEN, LOW);
        digitalWrite(LED_RIGHT_YELLOW, LOW);
        digitalWrite(LED_STRAIGHT_RED, LOW);
        digitalWrite(LED_LEFT_YELLOW, LOW);
```

```
      digitalWrite(LED_LEFT_GREEN, HIGH);
      break;
    }
}

void plotControlData() {
  Serial.println("MOTOR 1 - State, StrPos, StrSpeed, Desired Angle, Motor Pos, Desired Motor Pos, PWM_Duty");
  Serial.print(stateToString(currentState));
  Serial.print(" ");
  Serial.print(strPos - straightOffset);
  Serial.print(" ");
  Serial.print(strSpeed);
  Serial.print(" ");
  Serial.print(camberAngle);
  Serial.print(" ");
  Serial.print(theta);
  Serial.print(" ");
  Serial.print(thetaDes);
  Serial.print(" ");
  Serial.println(X);
}

const char* stateToString(State state) {
  switch (state) {
    case OFF: return "OFF";
    case STRAIGHT: return "STRAIGHT";
    case ENTRY_EXIT: return "ENTRY_EXIT";
    case APEX: return "APEX";
  }
}
```