School of Engineering and Computer Science
Electrical and Computer Engineering Dept.
Oakland University

ECE 5/4520
Automotive Mechatronics I

Course project

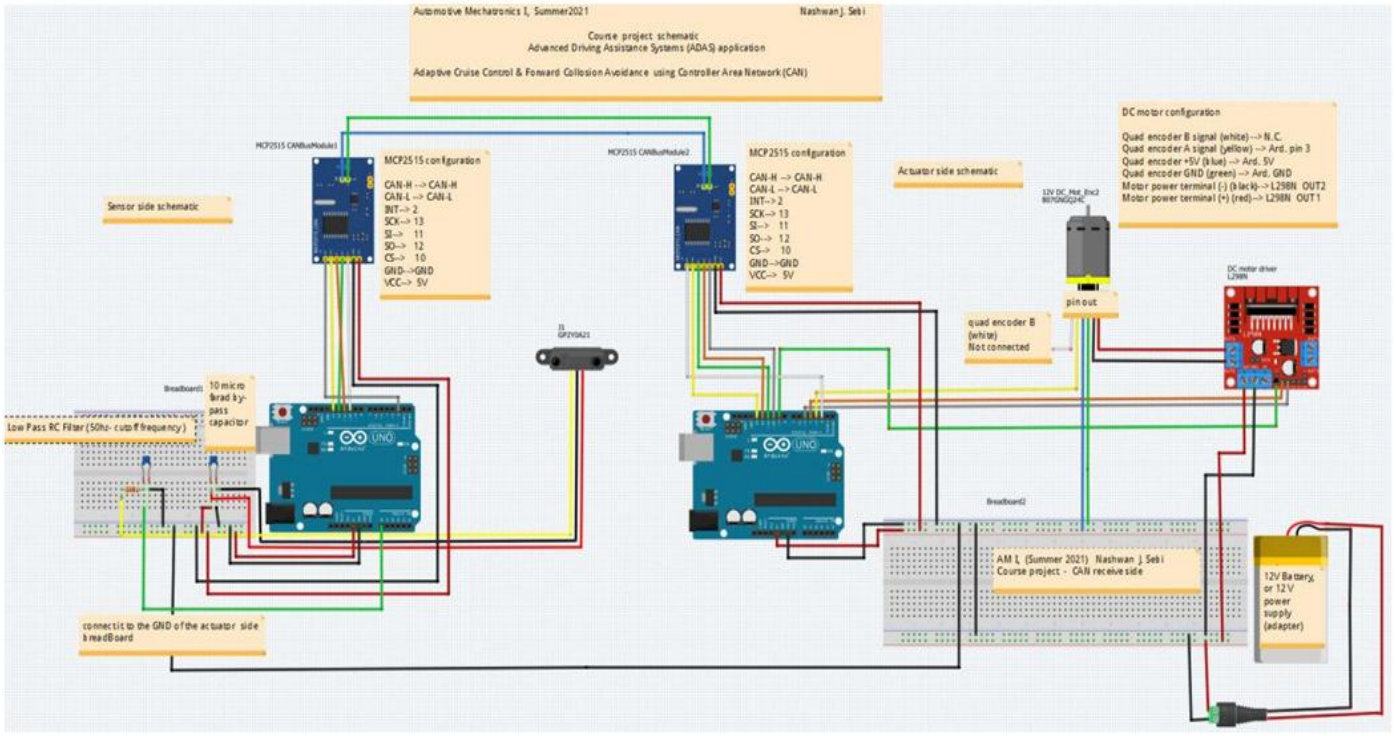
Adaptive Cruise Control (ACC) and Forward Collision Avoidance (FCA) using Controller Area Network (CAN)

Student Name: Yousif Fatohi, Peter Younan, Caleb Cook, Joshua Pagonas
Instructor Name: Dr. Sebi

1. Completed Code (TX and RX Sides)

Screenshots of the completed transmit (TX) and receive (RX) codes, captured directly from the Arduino IDE.

1.1a Circuit Diagram



1.1b Transmit (TX) Side: IR Sensor Node

The TX node reads the Sharp GP2Y0A21 IR distance sensor on analog pin A1, converts each reading to centimeters, applies a 40 sample moving average, and transmits the averaged distance in byte 8 of a CAN frame (ID 0x07) at 500 kbps through CAN module #1.

```
// IR sensor board: reads the IR sensor and sends the distance over CAN (ID 0x07, distance in cm in byte 7)
#include "Arduino_RouterBridge.h"
#include <SPI.h>

const int IR_PIN = A1;            // IR sensor signal
const int MCP_CS = 10;            // MCP2515 CS (SCK 13, SI 11, SO 12)

#define DIST_ID 0x07              // CAN ID the motor board listens for
#define AVG_SAMPLES 40            // moving average over about 0.4 s

unsigned char distance[8] = {0};  // BLANK 1: CAN payload, distance in cm goes in byte 7

int samples[AVG_SAMPLES];
long sum = 0;
int idx = 0;
bool canReady = false;
unsigned long lastSend = 0, lastCheck = 0;

// ---------- MCP2515 ----------
// Sends n bytes in one chip-select, returns the last byte read back
byte mcp(const byte *out, int n)
{
  byte in = 0;
  SPI.beginTransaction(SPISettings(1000000, MSBFIRST, SPI_MODE0));
  digitalWrite(MCP_CS, LOW);
  for (int i = 0; i < n; i++) in = SPI.transfer(out[i]);
  digitalWrite(MCP_CS, HIGH);
  SPI.endTransaction();
  return in;
}

byte mcpRead(byte addr)                    { byte b[] = {0x03, addr, 0};          return mcp(b, 3); }
void mcpWrite(byte addr, byte v)           { byte b[] = {0x02, addr, v};          mcp(b, 3); }
void mcpBitModify(byte addr, byte m, byte v)  { byte b[] = {0x05, addr, m, v};    mcp(b, 4); }
void mcpCommand(byte c)                    { mcp(&c, 1); }

bool canBegin()
{
  mcpCommand(0xC0);                              // reset
  delay(10);
```

```
    delay(10);
    if ((mcpRead(0x0E) & 0xE0) != 0x80) return false;  // chip not answering

    mcpWrite(0x2A, 0x00);                              // CNF1  (500 kbps, 16 MHz crystal)
    mcpWrite(0x29, 0xF0);                              // CNF2
    mcpWrite(0x28, 0x86);                              // CNF3
    mcpBitModify(0x0F, 0xE0, 0x00);                    // normal mode

    for (int i = 0; i < 50; i++)
    {
      if ((mcpRead(0x0E) & 0xE0) == 0x00) return true;
      delay(2);
    }
    return false;
  }

  void canSend()
  {
    if (mcpRead(0x30) & 0x08)                          // last frame never ACKed: abort it
    {
      mcpBitModify(0x30, 0x08, 0x00);
      for (int i = 0; i < 20 && (mcpRead(0x30) & 0x08); i++) delayMicroseconds(100);
    }

    byte frame[14] = {0x40, DIST_ID >> 3, (DIST_ID & 0x07) << 5, 0, 0, 8};   // load TX buffer 0: ID + 8 bytes
    for (int i = 0; i < 8; i++) frame[6 + i] = distance[i];
    mcp(frame, 14);
    mcpCommand(0x81);                                  // send it
  }

  // ---------- IR sensor ----------
  int adcToCm(int adc)
  {
    long counts5V = (long)adc * 33L / 50L;             // 3.3 V reading -> 5 V scale
    int cm = (int)(6787L / (counts5V + 3L)) - 4;
    return constrain(cm, 0, 255);                      // must fit in one CAN byte
  }

  void setup()
  {
    Bridge.begin();
```

```
  void setup()
  {
    Bridge.begin();
    analogReadResolution(10);

    pinMode(MCP_CS, OUTPUT);
    digitalWrite(MCP_CS, HIGH);
    SPI.begin();
    canReady = canBegin();

    int first = adcToCm(analogRead(IR_PIN));           // start the average at the real reading
    for (int i = 0; i < AVG_SAMPLES; i++) samples[i] = first;
    sum = (long)first * AVG_SAMPLES;
  }

  void loop()
  {
    // moving average in cm
    int cm = adcToCm(analogRead(IR_PIN));
    sum += cm - samples[idx];
    samples[idx] = cm;
    idx = (idx + 1) % AVG_SAMPLES;

    distance[7] = sum / AVG_SAMPLES;                   // BLANK 2

    if (canReady && millis() - lastSend >= 50)         // one CAN frame every 50 ms
    {
      lastSend = millis();
      canSend();
    }

    if (millis() - lastCheck >= 1000)                  // set the MCP2515 up again if it lost power/setup
    {
      lastCheck = millis();
      if (!canReady || (mcpRead(0x0E) & 0xE0) != 0x00) canReady = canBegin();
    }

    delay(10);
  }
```

Figure 1: Completed TX side code (IR reading, averaging, and CAN transmit)

1.2 Receive (RX) Side: Motor Control Node

The RX node reads the distance from byte 8 of each CAN frame (ID 0x07) through CAN module #2 and converts it to a target speed: 20% of full speed at 9 cm or closer, rising 5% per cm to 100% at 25 cm or farther. A measured motor curve turns that speed into a PWM value of 0 to 255, which pin 9 applies to the L298N driver. The speed eases toward the target with a 150 ms time constant, and if no distance frame arrives for one second the motor slows to the 20% crawl speed instead of stopping. Encoder channel A on pin 3 triggers an interrupt on each falling edge, and every 500 ms the pulse count (540 pulses per output shaft revolution) gives the motor speed in rpm and the linear velocity of the 3 cm radius wheel.

```
#include "Arduino_RouterBridge.h"   // UNO Q: Bridge to the Linux side (Python tab)

#define ENCODEROUTPUT 540     // encoder pulses per output-shaft revolution
#define PWM 9                 // L298N ENA. No pinMode() on this pin: on the UNO Q it stops analogWrite()
#define encoder0pinA 3        // encoder A (yellow)

int M_left = 4;               // L298N IN1
int E_left = 5;               // L298N IN2
const int MCP_CS = 10;        // MCP2515 CS
const int PIN_MOSI = 11;      // MCP2515 SI
const int PIN_MISO = 12;      // MCP2515 SO
const int PIN_SCK = 13;       // MCP2515 SCK

// Speed vs distance (same values as the old table): 20 % at 9 cm or closer, +5 % per cm, 100 % at 25 cm or farther
#define MIN_CM 9
#define MIN_PCT 20            // also the crawl speed when CAN data is lost
#define PCT_PER_CM 5
#define SMOOTH_MS 150         // speed eases toward the table value with this time constant: lower = snappier, higher = smoother
#define START_KICK_MS 150     // first start: full power this long to break static friction

// Measured motor curve, {speed %, PWM}
const byte CURVE[][2] = {{0, 0}, {7, 35}, {23, 50}, {34, 60}, {48, 80}, {70, 120}, {82, 160}, {89, 200}, {100, 255}};

#define CAN_LOST_MS 1000      // no distance frame for this long: ease down to the crawl speed (never stops)
#define STATUS_MS 500         // status to the Python tab and the Serial Monitor this often
#define DIST_ID 0x07          // CAN ID sent by the sensor board
#define DIST_BYTE 7           // byte that holds the distance in cm

volatile unsigned long encoderValue = 0;
unsigned long kickUntil = 0, lastUpdate = 0, maxLoopMs = 0, lastStatus = 0, lastFrameMs = 0, framesRx = 0, lastHealthCheck = 0;
float speedPct = 0;           // speed commanded right now, easing toward the table value. 0 = not started yet
int distanceCm = -1;
bool canReady = false;

int speedForDistance(int cm)
{
  return constrain(MIN_PCT + PCT_PER_CM * (cm - MIN_CM), MIN_PCT, 100);
}

int pwmForSpeed(float pct)
```

```
int pwmForSpeed(float pct)
{
  if (pct <= 0) return 0;
  for (unsigned i = 1; i < sizeof(CURVE) / sizeof(CURVE[0]); i++)
  {
    if (pct <= CURVE[i][0])
    {
      float f = (pct - CURVE[i - 1][0]) / (CURVE[i][0] - CURVE[i - 1][0]);
      return (int)(CURVE[i - 1][1] + f * (CURVE[i][1] - CURVE[i - 1][1]) + 0.5);
    }
  }
  return 255;
}

// One bit-banged MCP2515 SPI transaction: sends b[0..n-1], each byte is replaced by the byte that came back
void mcpXfer(byte *b, byte n)
{
  digitalWrite(MCP_CS, LOW);
  delayMicroseconds(2);
  for (byte k = 0; k < n; k++)
  {
    byte in = 0;
    for (int i = 7; i >= 0; i--)
    {
      digitalWrite(PIN_MOSI, (b[k] >> i) & 1);
      delayMicroseconds(2);
      digitalWrite(PIN_SCK, HIGH);
      delayMicroseconds(2);
      in = (in << 1) | (digitalRead(PIN_MISO) ? 1 : 0);
      digitalWrite(PIN_SCK, LOW);
    }
    b[k] = in;
  }
  delayMicroseconds(2);
  digitalWrite(MCP_CS, HIGH);
  delayMicroseconds(2);
}

byte mcpRead(byte addr)             { byte b[] = {0x03, addr, 0};     mcpXfer(b, 3); return b[2]; }   // READ
void mcpWrite(byte addr, byte value) { byte b[] = {0x02, addr, value}; mcpXfer(b, 3); }               // WRITE
```

```

bool mcpBegin()
{
  byte rst[] = {0xC0};                              // RESET
  mcpXfer(rst, 1);
  delay(10);
  if ((mcpRead(0x0E) & 0xE0) != 0x80) return false;   // CANSTAT not in config mode: chip not answering

  // One WRITE burst (0x28-0x2C are consecutive): CNF3, CNF2, CNF1, CANINTE = 0 (we poll), CANINTF = 0 (clear flags)
  byte cfg[] = {0x02, 0x28, 0x86, 0xF0, 0x00, 0x00, 0x00};
  mcpXfer(cfg, 7);
  mcpWrite(0x60, 0x64);                             // RXB0CTRL: accept every frame, roll over into RXB1
  mcpWrite(0x70, 0x60);                             // RXB1CTRL: accept every frame
  mcpWrite(0x0F, 0x07);                             // CANCTRL: request normal mode (reset value 0x87 with REQOP = 000)
  for (int i = 0; i < 50; i++, delay(2))
    if ((mcpRead(0x0E) & 0xE0) == 0x00) return true;
  return false;
}

// Reads one waiting frame into f[14]: f[1..5] = SIDH SIDL EID8 EID0 DLC, f[6..13] = data. false = nothing waiting
bool mcpReceive(byte *f)
{
  byte flags = mcpRead(0x2C);                       // CANINTF
  if (!(flags & 0x03)) return false;
  memset(f, 0, 14);
  f[0] = (flags & 0x01) ? 0x90 : 0x94;              // READ RX BUFFER 0 or 1 (clears its RXnIF)
  mcpXfer(f, 14);
  return true;
}

void setup()
{
  Bridge.begin();
  Monitor.begin();                  // App Lab Serial Monitor (required before Monitor.print)
  pinMode(M_left, OUTPUT);
  pinMode(E_left, OUTPUT);
  pinMode(encoder0pinA, INPUT_PULLUP);
  digitalWrite(M_left, LOW);          // IN1 LOW + IN2 HIGH = forward
  digitalWrite(E_left, HIGH);
  analogWrite(PWM, 0);
```

Figure 2: Completed RX side code, part 1: declarations, helper functions, and start of setup()

```
      analogWrite(PWM, 0);

      attachInterrupt(digitalPinToInterrupt(encoder0pinA), [] { encoderValue++; }, FALLING);

      pinMode(MCP_CS, OUTPUT);
      digitalWrite(MCP_CS, HIGH);
      pinMode(PIN_SCK, OUTPUT);
      digitalWrite(PIN_SCK, LOW);
      pinMode(PIN_MOSI, OUTPUT);
      pinMode(PIN_MISO, INPUT_PULLUP);    // pull-up: a disconnected SO reads 0xFF, not random
      canReady = mcpBegin();

      lastUpdate = lastStatus = millis();
    }

    void loop()
    {
      unsigned long now = millis();
      unsigned long dt = now - lastUpdate;
      lastUpdate = now;
      if (dt > maxLoopMs) maxLoopMs = dt;

      byte f[14];
      for (int i = 0; i < 2 && canReady && mcpReceive(f); i++)
      {
        unsigned int id = ((unsigned int)f[1] << 3) | (f[2] >> 5);
        if (!(f[2] & 0x08) && id == DIST_ID && (f[5] & 0x0F) > DIST_BYTE)   // standard frame, our ID, long enough
        {
          distanceCm = f[6 + DIST_BYTE];
          lastFrameMs = now;
          framesRx++;
        }
      }

      bool fresh = framesRx && (now - lastFrameMs < CAN_LOST_MS);
      int tablePct = 0, pwmOut = 0;       // both stay 0 until the first distance frame arrives
      if (framesRx)
      {
        tablePct = fresh ? speedForDistance(distanceCm) : MIN_PCT;
        if (speedPct == 0)                // first start: straight to the table speed, with a kick
        {
```

```
        {
          speedPct = tablePct;
          kickUntil = now + START_KICK_MS;
        }
        speedPct += (tablePct - speedPct) * dt / (SMOOTH_MS + dt);                    // ease toward the table value
        if (speedPct > tablePct - 0.5 && speedPct < tablePct + 0.5) speedPct = tablePct;  // land exactly (100 % = PWM 255)
        pwmOut = pwmForSpeed(speedPct);
      }
      analogWrite(PWM, (pwmOut > 0 && (long)(now - kickUntil) < 0) ? 255 : pwmOut);   // 255 during the start kick

      if (now - lastHealthCheck >= 1000)
      {
        lastHealthCheck = now;
        if (!canReady || (mcpRead(0x0E) & 0xE0) != 0x00) canReady = mcpBegin();
      }

      if (now - lastStatus >= STATUS_MS)
      {
        unsigned long span = now - lastStatus;
        lastStatus = now;
        noInterrupts();
        unsigned long pulses = encoderValue;
        encoderValue = 0;
        interrupts();
        int rpm = (int)(pulses * 60000UL / ((unsigned long)ENCODEROUTPUT * span));
        float v = 2 * 3.14159265 * 0.03 * rpm / 60;

        String s = "dist=";
        if (fresh) s += String(distanceCm) + "cm";
        else s += "-- (no CAN data: crawling)";
        s += String(" table=") + tablePct + "% now=" + (int)(speedPct + 0.5) + "% pwm=" + pwmOut + " rpm=" + rpm +
             " v=" + String(v, 2) + "m/s frames=" + (long)framesRx + " loop=" + (long)maxLoopMs + "ms";
        if (framesRx == 0) s += " WAITING for the first distance frame";
        if (!canReady) s += " CAN-OFFLINE";
        Bridge.notify("telemetry", s);

        // Serial Monitor block in the lab format, built as one String and sent with ONE Monitor call:
        // each Monitor call can be a blocking round trip to the Linux side, and several per report stall the loop
        Monitor.print(String("rpm =") + rpm + "\r\nLinear Velocity (V)=" + String(v, 2) + "m/s\r\nPWM =" + pwmOut +
                      "\r\ndis =" + (fresh ? String(distanceCm) : String("--")) + "\r\n------------------------------\r\n");
        maxLoopMs = 0;
```



```
        noInterrupts();
        unsigned long pulses = encoderValue;
        encoderValue = 0;
        interrupts();
        int rpm = (int)(pulses * 60000UL / ((unsigned long)ENCODEROUTPUT * span));
        float v = 2 * 3.14159265 * 0.03 * rpm / 60;

        String s = "dist=";
        if (fresh) s += String(distanceCm) + "cm";
        else s += "-- (no CAN data: crawling)";
        s += String(" table=") + tablePct + "% now=" + (int)(speedPct + 0.5) + "% pwm=" + pwmOut + " rpm=" + rpm +
             " v=" + String(v, 2) + "m/s frames=" + (long)framesRx + " loop=" + (long)maxLoopMs + "ms";
        if (framesRx == 0) s += " WAITING for the first distance frame";
        if (!canReady) s += " CAN-OFFLINE";
        Bridge.notify("telemetry", s);

        // Serial Monitor block in the lab format, built as one String and sent with ONE Monitor call:
        // each Monitor call can be a blocking round trip to the Linux side, and several per report stall the loop
        Monitor.print(String("rpm =") + rpm + "\r\nLinear Velocity (V)=" + String(v, 2) + "m/s\r\nPWM =" + pwmOut +
                      "\r\ndis =" + (fresh ? String(distanceCm) : String("--")) + "\r\n----------------------------\r\n");
        maxLoopMs = 0;
      }

      delay(5);
    }
```

Figure 3: Completed RX side code, part 2: end of setup() (encoder interrupt) and loop()

2. Photos of the Physical Structure

2.1 Whole Physical Structure (Top View)

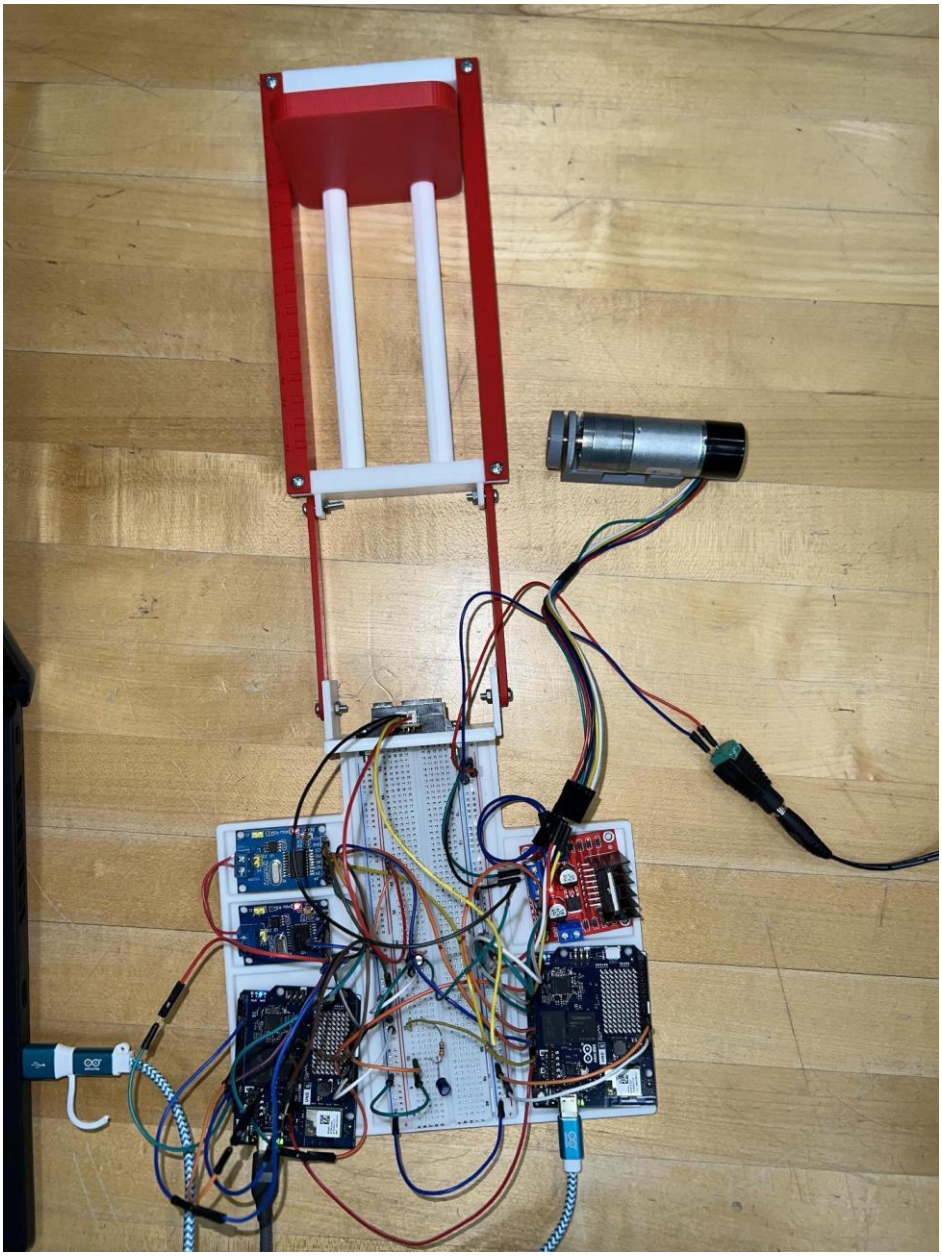


Figure 4: Whole physical structure, top view

2.2 Sending (TX) Side

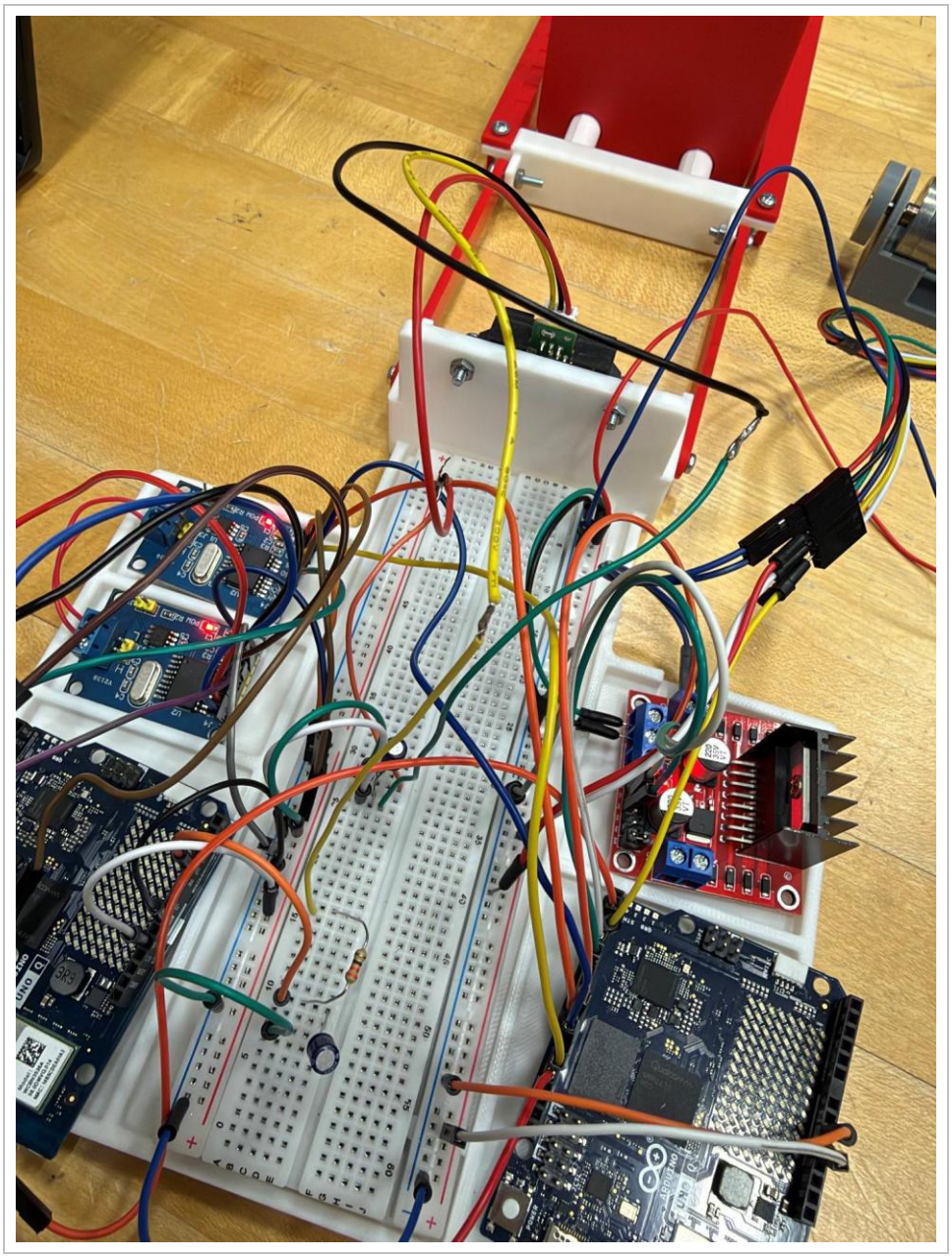


Figure 5: Sending (TX) side: IR sensor (GP2Y0A21), 10 µF bypass capacitor, RC low pass filter (50 Hz cutoff), Arduino Uno #1, and CAN module #1 (MCP2515)

2.3 Receiving (RX) Side

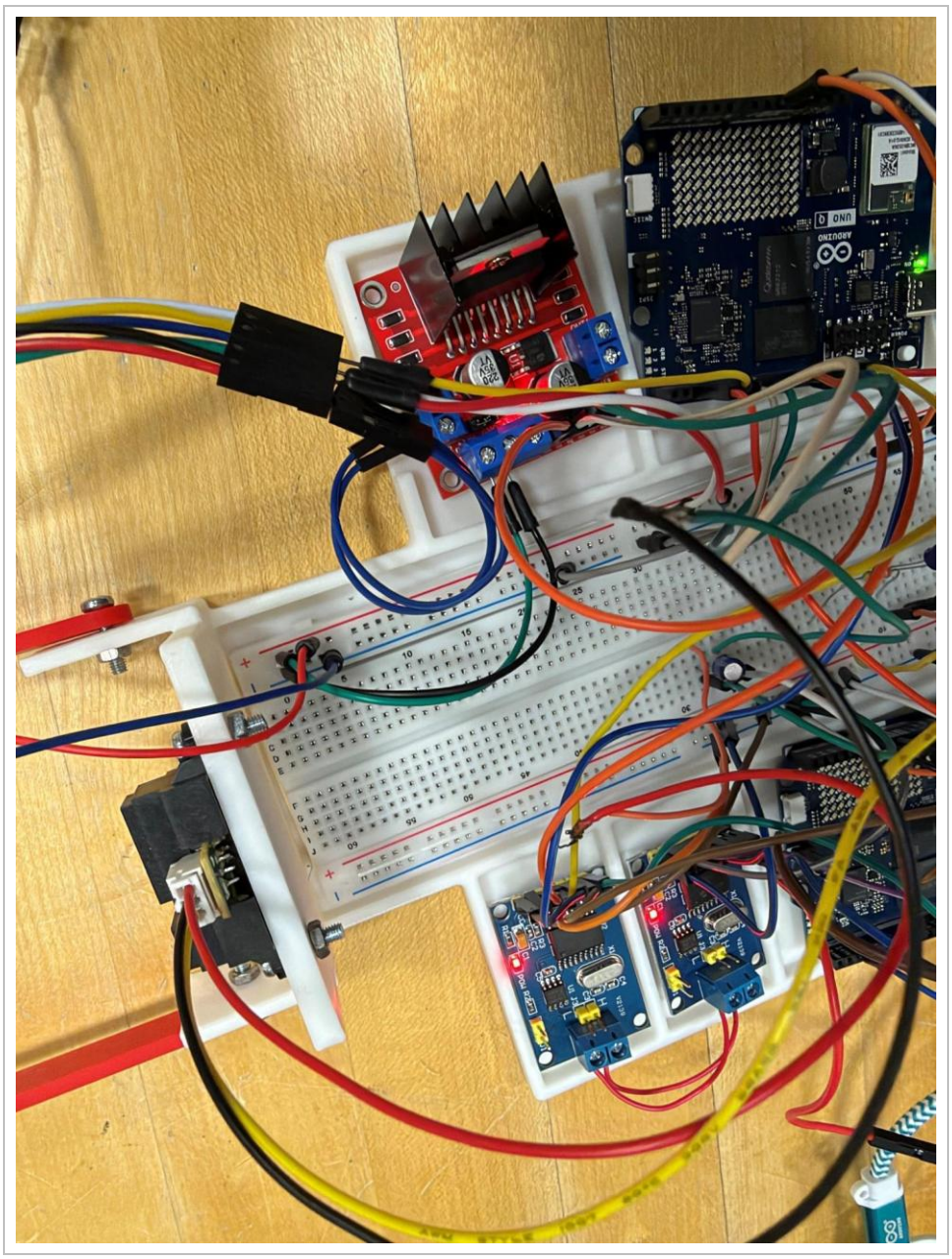


Figure 6: Receiving (RX) side: CAN module #2 (MCP2515), Arduino Uno #2, and L298N DC motor driver

2.4 12 V Geared DC Motor with Encoder

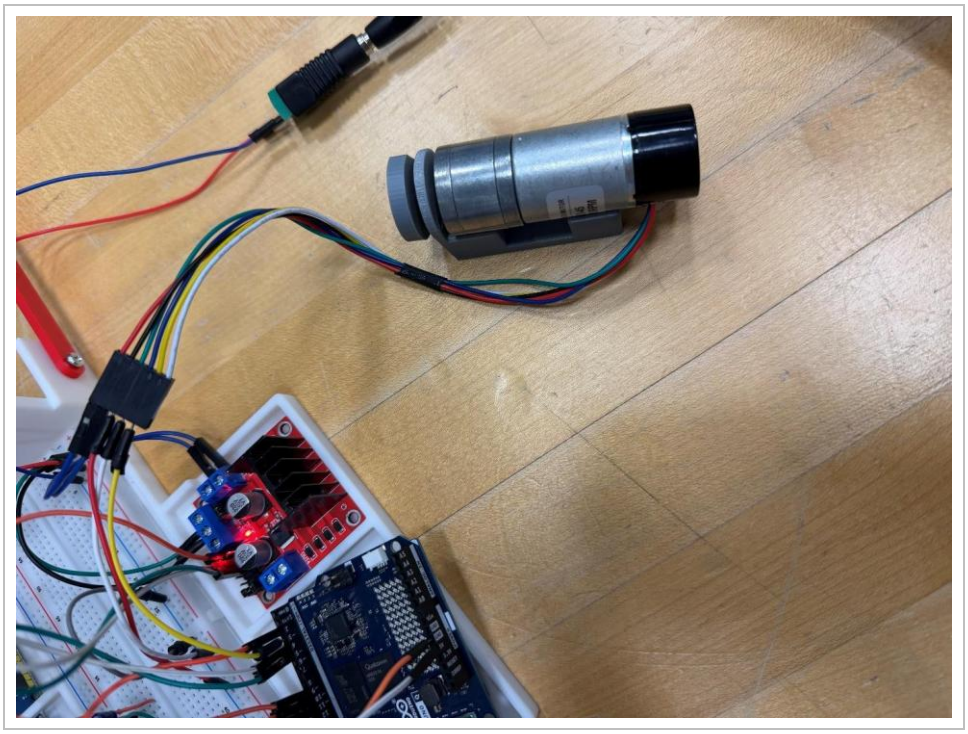


Figure 7: 12 V geared DC motor with quadrature Hall effect encoder

2.5 Serial Monitor Output

Serial monitor output from the RX node at four different obstacle distances, showing the received distance (dis), PWM value, motor speed (rpm), and linear velocity.

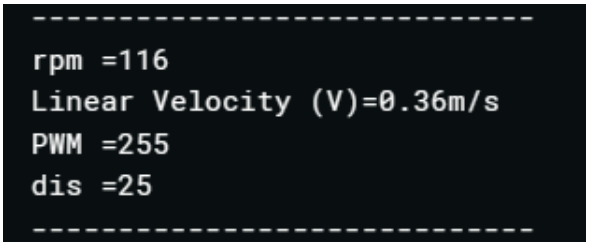


Figure 8: Serial monitor, reading 1

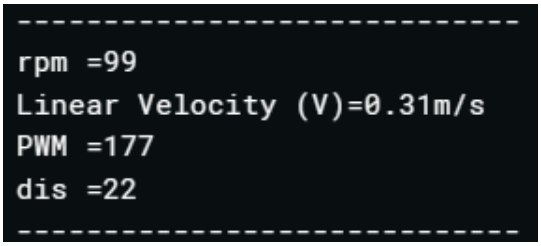


Figure 9: Serial monitor, reading 2

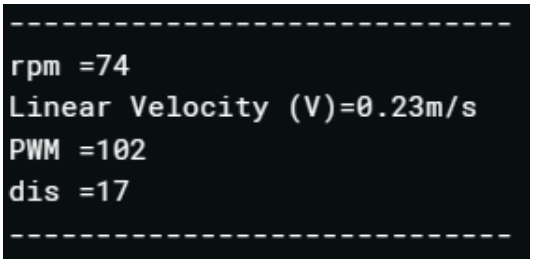


Figure 10: Serial monitor, reading 3

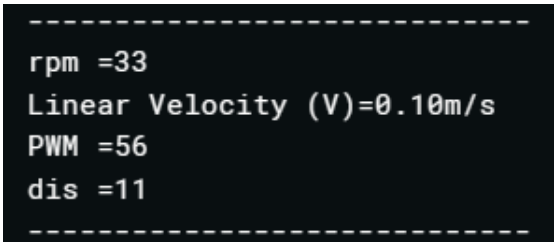


Figure 11: Serial monitor, reading 4

Table 1: Summary of the serial monitor readings in Figures 8 to 11

Reading	Distance, dis (cm)	PWM (0 to 255)	Speed (rpm)	Linear velocity (m/s)
Figure 8	25	255	116	0.36
Figure 9	22	177	99	0.31
Figure 10	17	102	74	0.23
Figure 11	11	56	33	0.10

3. Demonstration Video

The MP4 demonstration video captures the physical structure, the DC motor, and the serial monitor at the same time while the obstacle distance changes, showing the motor speed being controlled by the IR sensor over the CAN bus.

YouTube link (unlisted)	https://youtube.com/shorts/CVVfN91MFhM?feature=share
Video format	MP4

Watch Demonstration Video