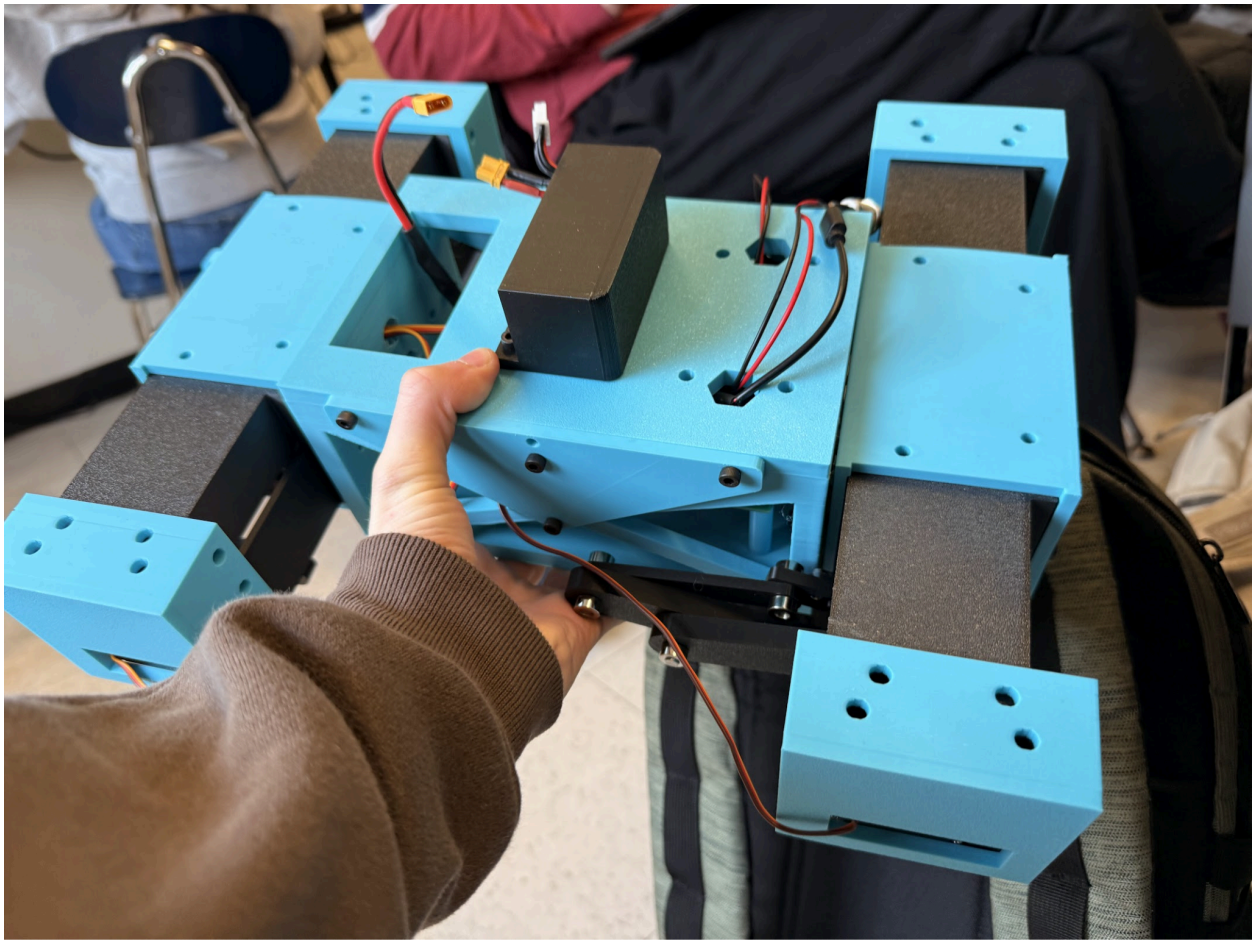
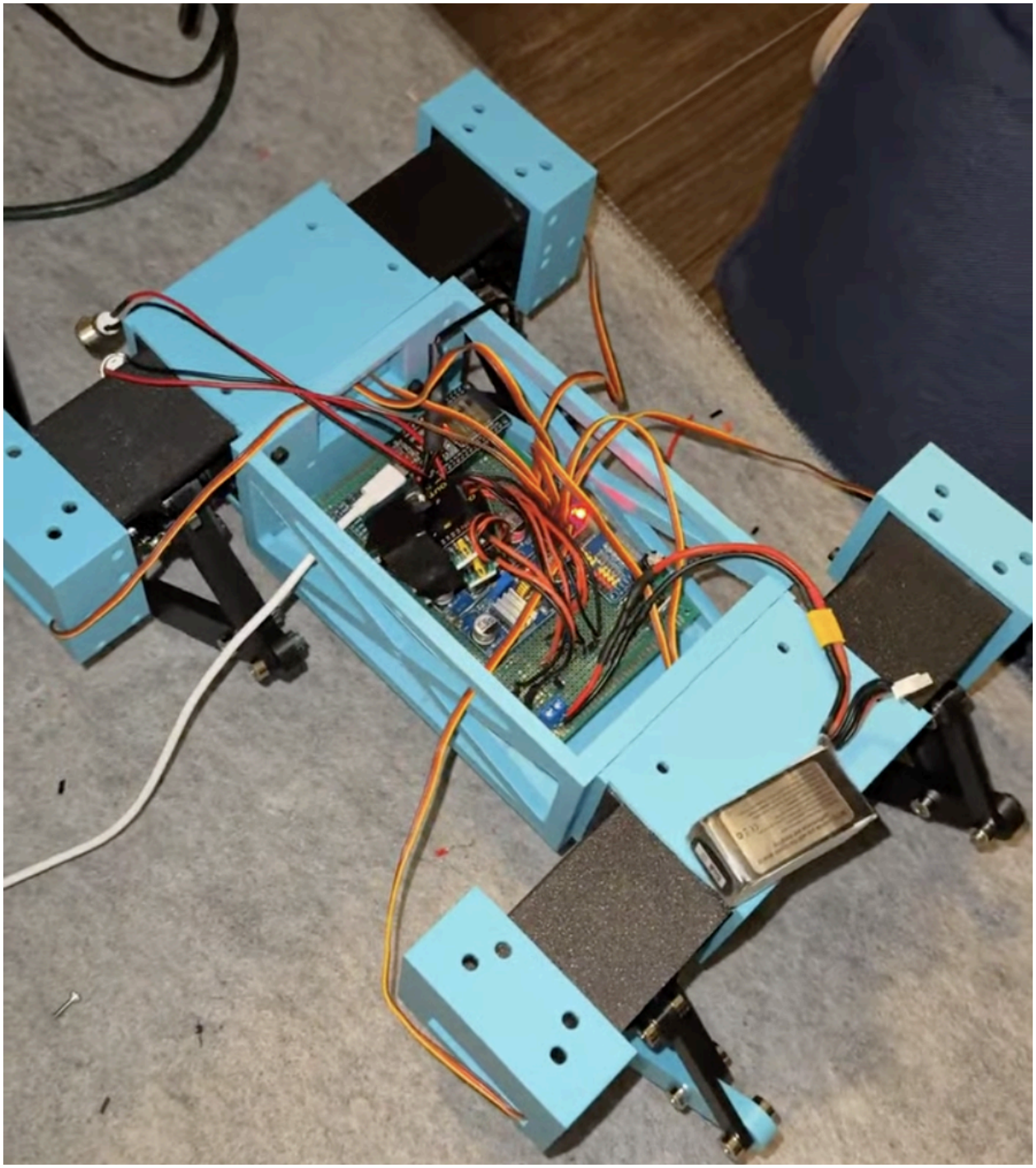
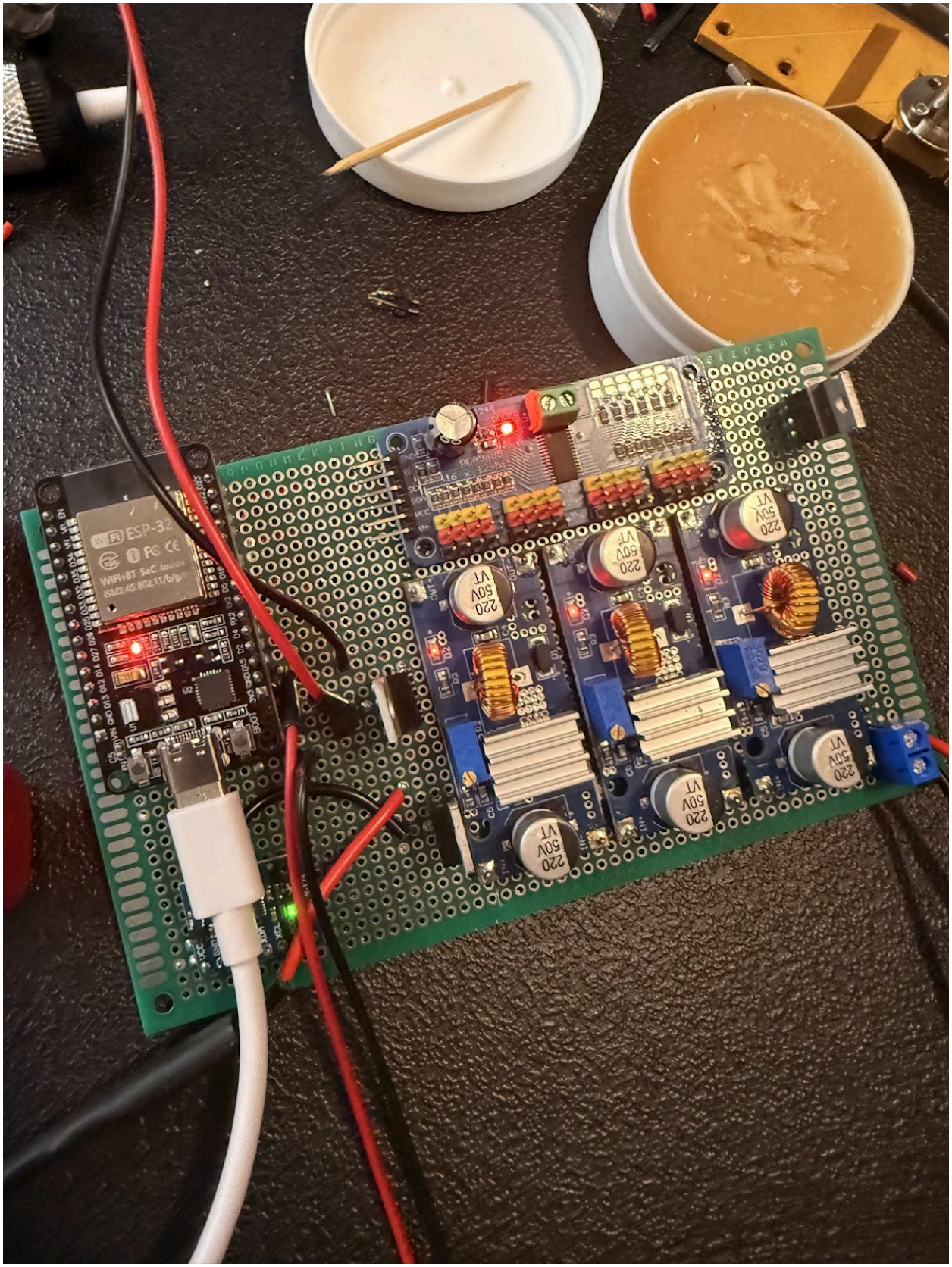
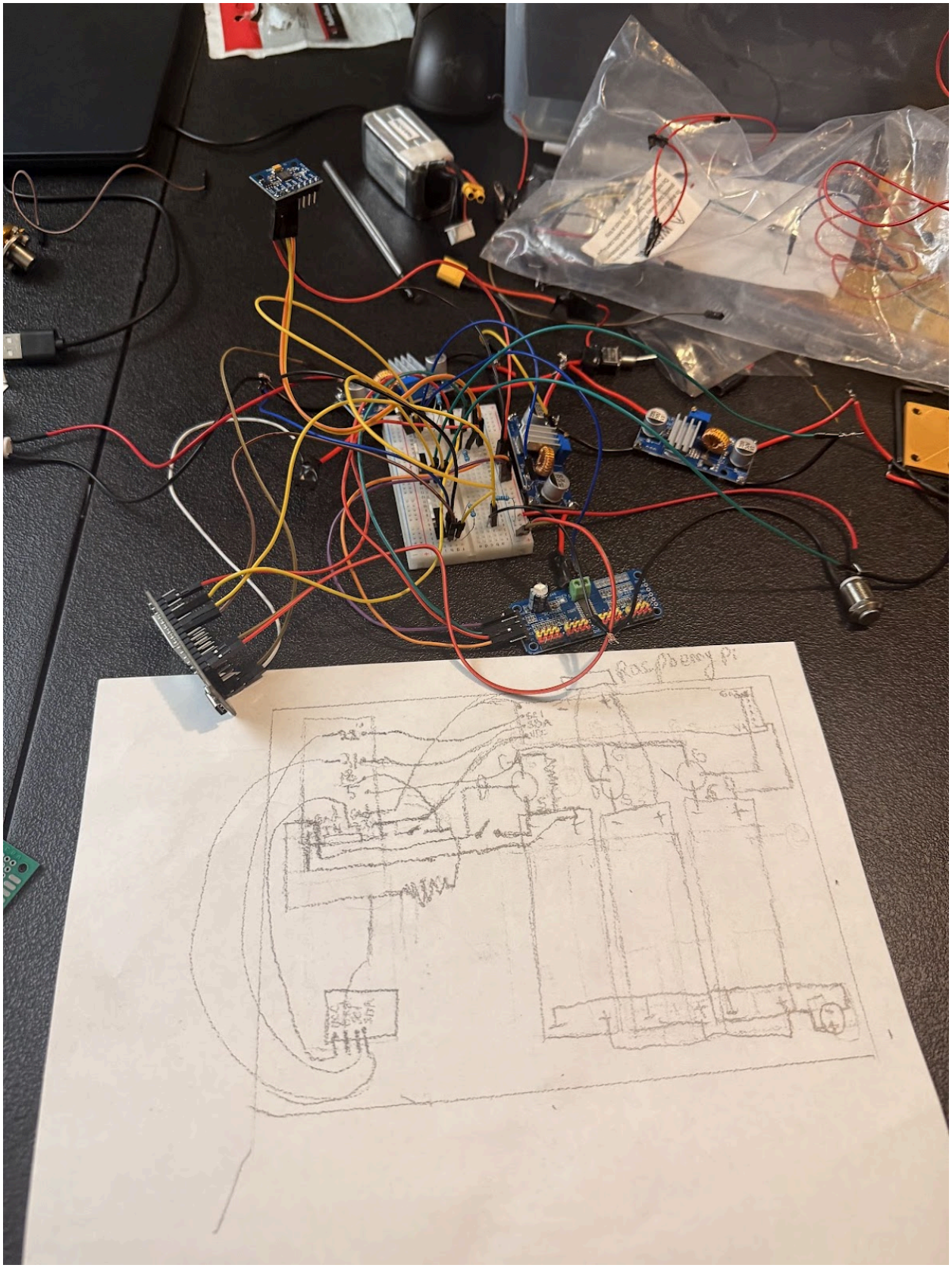




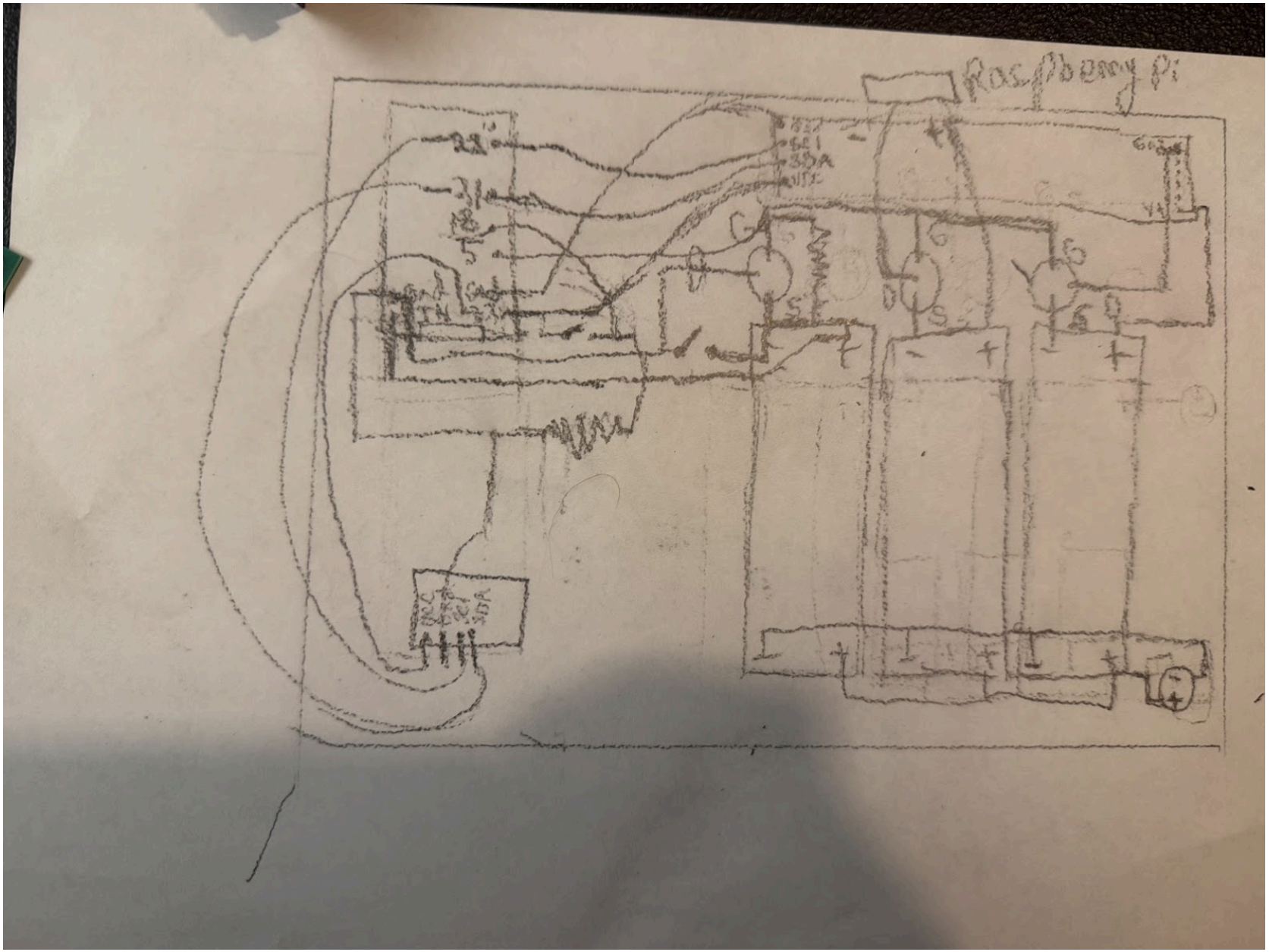
First prototype board

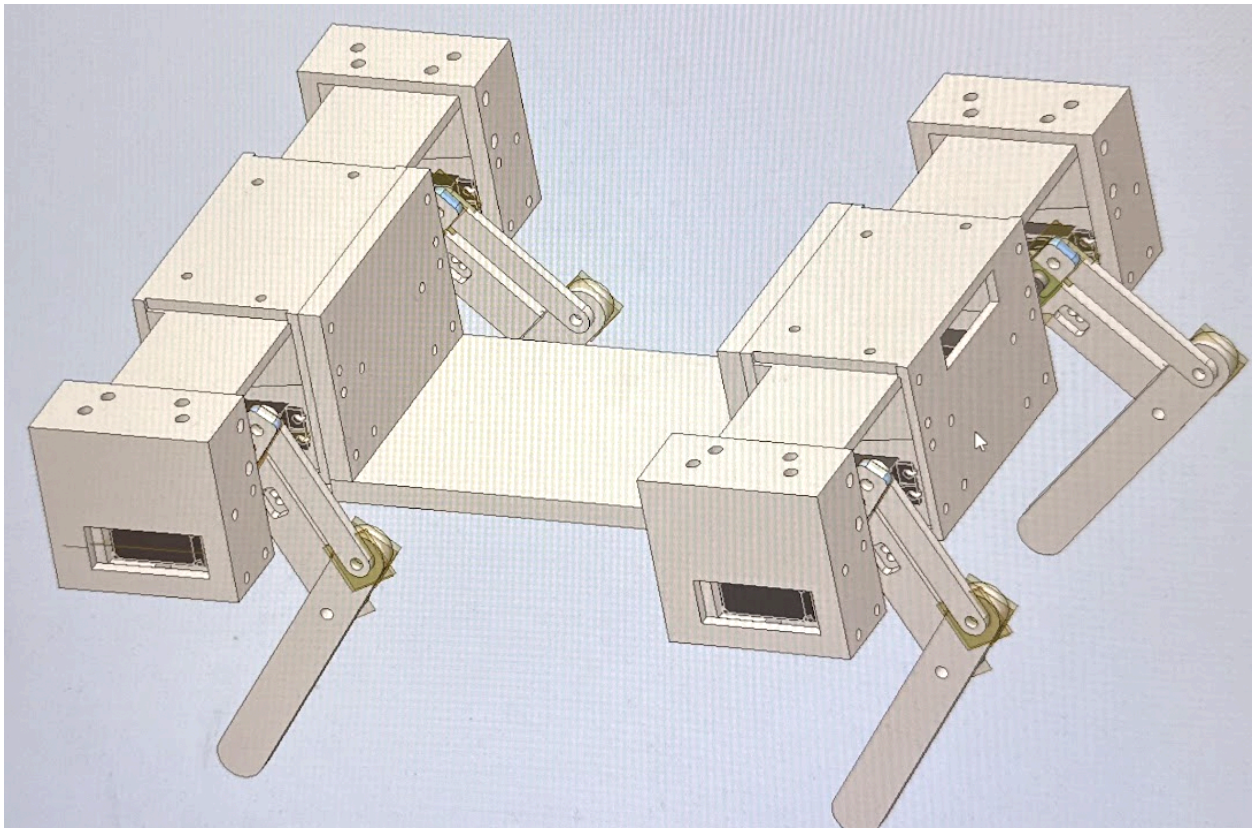

ESP-32
WiFi+BT SoC
220
50V
VT

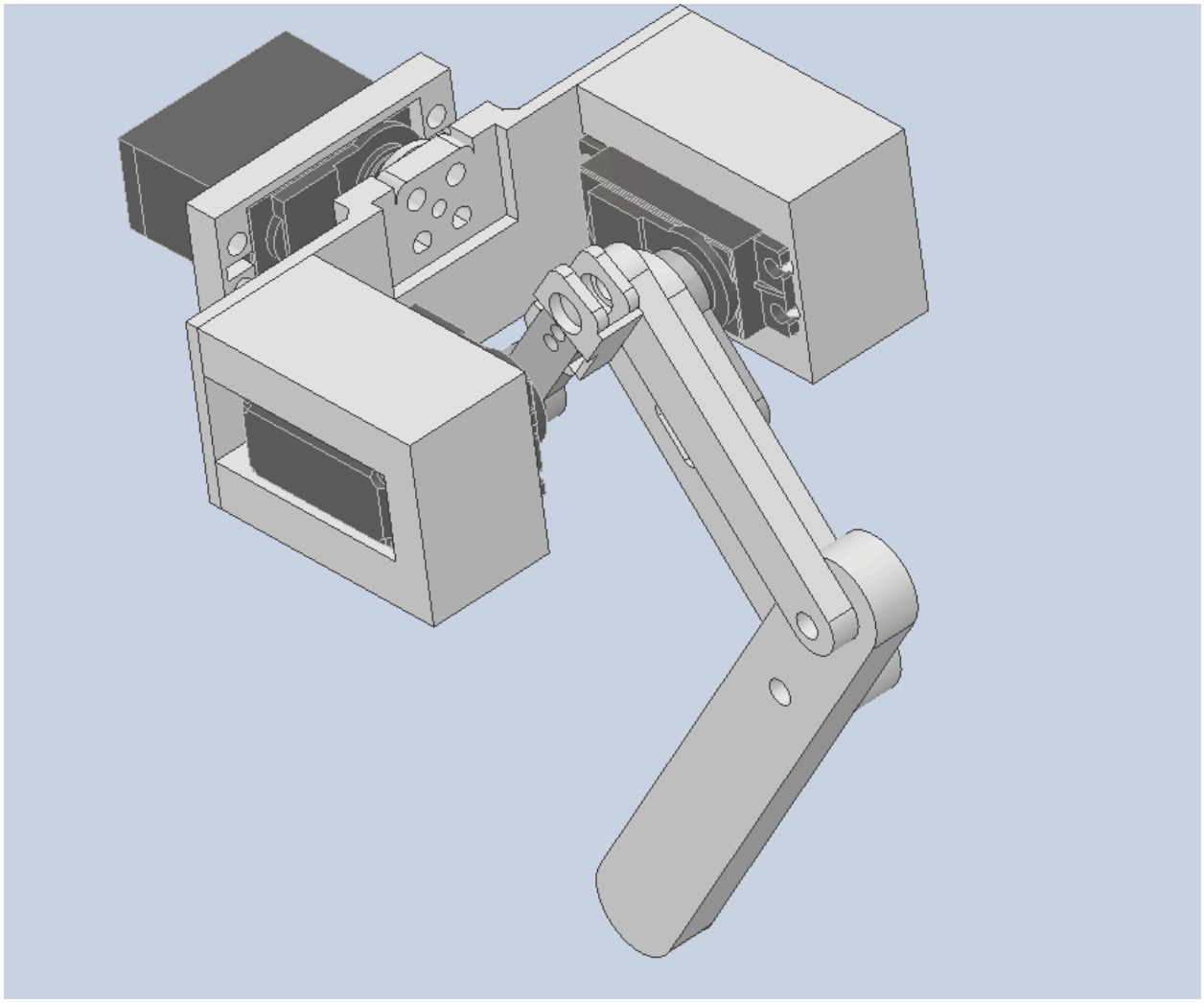
Breadboarding test board

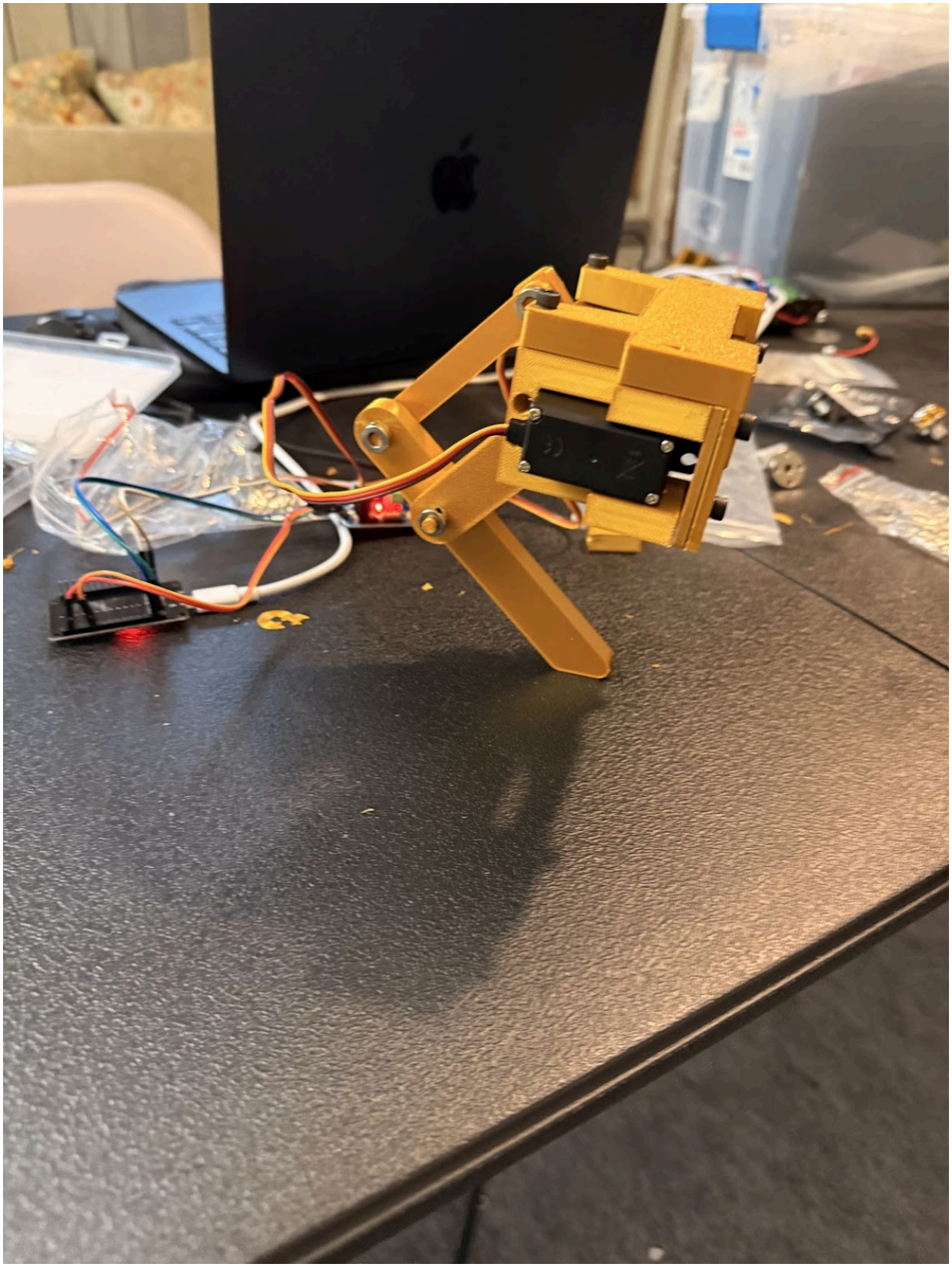


Circut diagram


Raspberry Pi
SCL
SDA







```
// ===== DOG CODE =====
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>
#include <MPU6050.h>
#include <math.h>
#include <WiFi.h>
#include <esp_now.h>

// ===== HARDWARE =====
Adafruit_PWMServoDriver pca = Adafruit_PWMServoDriver();
MPU6050 mpu;

#define MOSFET_PIN       5
#define SHUTDOWN_BUTTON 19
#define SERVOMIN 150
#define SERVOMAX 535
#define TEST_CHANNEL 15
#define amplitude 20

// ===== BUTTON =====
const unsigned long LONG_PRESS_MS = 1500;
unsigned long buttonPressStart = 0;
bool buttonWasPressed = false;
bool walkToggle = false;

// ===== ESP-NOW =====
typedef struct {
  float speed;
  float turnBias;
  bool  active;
} ControlPacket;

ControlPacket rxPacket;
bool newPacket = false;

void onReceive(const esp_now_recv_info_t* info, const uint8_t* data, int len) {
  if (len == sizeof(ControlPacket)) {
    memcpy(&rxPacket, data, sizeof(ControlPacket));
    newPacket = true;
  }
}
```

```
// ===== GAIT =====
float n[8] = {46, 34, 40, 40, 36, 39, 31, 48};
float t = 0.0;
bool walking      = false;
bool balancing    = false;
bool remoteActive = false;
float gaitSpeed   = 0.6;
float turnBias    = 0.0;

unsigned long lastWalkTime    = 0;
unsigned long lastBalanceTime = 0;
const unsigned long STEP_MS   = 50;

// ===== FUNCTIONS =====
void writeServo(int channel, float angle) {
 angle = constrain(angle, 0, 180);
 int pulse = map((int)angle, 0, 180, SERVOMIN, SERVOMAX);
 pca.setPWM(channel, 0, pulse);
}

void slowMoveToNeutral() {
 const float TARGET         = 70.0;
 const int   STEPS          = 60;
 const int   STEP_DELAY_MS = 20;
 float startAngles[8];
 for (int i = 0; i < 8; i++) startAngles[i] = n[i];
 for (int s = 1; s <= STEPS; s++) {
   float frac = (float)s / STEPS;
   for (int i = 0; i < 8; i++)
     writeServo(i, startAngles[i] + (TARGET - startAngles[i]) * frac);
   delay(STEP_DELAY_MS);
 }
}

void walkCycle() {
 // Original simple circle gait, speed controls direction and magnitude
 const float MIN_SPEED = 0.15;
 float spd = gaitSpeed;
 if (abs(spd) > 0.01) {
   float sign = spd > 0 ? 1.0 : -1.0;
   spd = sign * constrain(abs(spd), MIN_SPEED, 0.8);
 }
```

```
  // FR and BL diagonal pair A
  writeServo(0, n[0] + cos(t) * amplitude);
  writeServo(1, n[1] + sin(t) * amplitude);

  // FL and BR diagonal pair B (PI offset)
  writeServo(2, n[2] + cos(t + PI) * amplitude);
  writeServo(3, n[3] + sin(t + PI) * amplitude);

  writeServo(4, n[4] + cos(t + PI) * amplitude);
  writeServo(5, n[5] + sin(t + PI) * amplitude);

  writeServo(6, n[6] + cos(t) * amplitude);
  writeServo(7, n[7] + sin(t) * amplitude);

  t = fmod(t + spd, TWO_PI);
}

void turnCycle() {
  const float TURN_SPEED  = 0.4;
  const float TURN_AMP    = amplitude;
  float spd = TURN_SPEED * turnBias;

  // turn right (turnBias +): only LEFT side moves forward, right side stays neutral
  // turn left  (turnBias -): only RIGHT side moves forward, left side stays neutral

  if (turnBias > 0) {
    // LEFT side moves (FL ch2,3 and BL ch6,7)
    writeServo(2, n[2] + cos(t + PI) * TURN_AMP);
    writeServo(3, n[3] + sin(t + PI) * TURN_AMP);
    writeServo(6, n[6] + cos(t) * TURN_AMP);
    writeServo(7, n[7] + sin(t) * TURN_AMP);
    // right side stays at neutral
    writeServo(0, n[0]);
    writeServo(1, n[1]);
    writeServo(4, n[4]);
    writeServo(5, n[5]);
  } else {
    // RIGHT side moves (FR ch0,1 and BR ch4,5)
    writeServo(0, n[0] + cos(t) * TURN_AMP);
    writeServo(1, n[1] + sin(t) * TURN_AMP);
    writeServo(4, n[4] + cos(t + PI) * TURN_AMP);
```

```
    writeServo(5, n[5] + sin(t + PI) * TURN_AMP);
    // left side stays at neutral
    writeServo(2, n[2]);
    writeServo(3, n[3]);
    writeServo(6, n[6]);
    writeServo(7, n[7]);
  }

  t = fmod(t + spd, TWO_PI);
}
float getTilt() {
  int16_t ax, ay, az, gx, gy, gz;
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
  return atan2((float)ay, (float)az) * 180.0 / PI;
}

void applyBalance() {
  const float DEAD_ZONE = 10.0;
  const float STEP_SIZE = 3.0;
  float tilt = -getTilt();
  if (abs(tilt) < DEAD_ZONE) return;
  float beyond         = abs(tilt) - DEAD_ZONE;
  float snapped        = floor(beyond / STEP_SIZE) * STEP_SIZE;
  float effectiveTilt = copysign(snapped, tilt);
  float adjust         = constrain(effectiveTilt * 0.5, -20, 20);
  for (int i = 0; i < 8; i++) {
    if (i % 2 == 0) writeServo(i, n[i] - adjust);
    else            writeServo(i, n[i] + adjust);
  }
}

void shutdownPi() {
  Serial.println("Cutting power...");
  slowMoveToNeutral();
  delay(1500);
  digitalWrite(MOSFET_PIN, LOW);
  delay(100);
  while (true);
}

// ===== SETUP =====
void setup() {
```

```
  Serial.begin(115200);
  Wire.begin(21, 22);
  delay(1000);

  pca.begin();
  pca.setPWMFreq(50);
  mpu.initialize();

  pinMode(SHUTDOWN_BUTTON, INPUT);
  pinMode(MOSFET_PIN, OUTPUT);
  digitalWrite(MOSFET_PIN, HIGH);

  for (int i = 0; i < 8; i++) writeServo(i, n[i]);
  writeServo(TEST_CHANNEL, 90);

  WiFi.mode(WIFI_STA);
  delay(100);
  Serial.print("Dog MAC: ");
  Serial.println(WiFi.macAddress());

  if (esp_now_init() != ESP_OK) {
    Serial.println("ESP-NOW init failed");
    return;
  }
  esp_now_register_recv_cb(onReceive);
}

// ===== LOOP =====
void loop() {
  digitalWrite(MOSFET_PIN, HIGH);

  // ===== BUTTON =====
  bool buttonDown = digitalRead(SHUTDOWN_BUTTON) == HIGH;
  if (buttonDown && !buttonWasPressed) {
    buttonPressStart = millis();
    buttonWasPressed = true;
  }
  if (!buttonDown && buttonWasPressed) {
    unsigned long held = millis() - buttonPressStart;
    buttonWasPressed = false;
    if (held >= LONG_PRESS_MS) {
      shutdownPi();
```

```
    } else {
      walkToggle = !walkToggle;
      if (walkToggle) {
        walking = true; balancing = false; t = 0.0;
      } else {
        walking = false; balancing = false;
        for (int i = 0; i < 8; i++) writeServo(i, n[i]);
      }
    }
  }

  // ===== ESP-NOW PACKET =====
  if (newPacket) {
    newPacket = false;
    if (rxPacket.active) {
      gaitSpeed     = rxPacket.speed;
      turnBias      = rxPacket.turnBias;
      walking       = true;
      balancing     = false;
      remoteActive = true;
    } else {
      walking       = false;
      remoteActive = false;
      gaitSpeed     = 0.6;
      turnBias      = 0.0;
      for (int i = 0; i < 8; i++) writeServo(i, n[i]);
    }
  }

  // ===== SERIAL COMMANDS =====
  String command = "";
  if (Serial.available()) command = Serial.readStringUntil('\n');
  if (command.length() > 0) {
    command.trim();
    command.toLowerCase();
    if       (command == "walk")         { walking = true;  balancing = false; t = 0.0; }
    else if (command == "stop")         { walking = false; balancing = false; for (int
i=0;i<8;i++) writeServo(i,n[i]); }
    else if (command == "balance")      { balancing = true;  walking = false; }
    else if (command == "stopbalance") { balancing = false; }
    else if (command == "neutral")      { walking = false; balancing = false;
slowMoveToNeutral(); }
```

```
  else if (command == "shutdown")     { shutdownPi(); }
  else if (command.startsWith("t"))   { writeServo(TEST_CHANNEL,
command.substring(1).toInt()); }
 }

 // ===== GAIT =====
 unsigned long now = millis();
 if (walking && now - lastWalkTime >= STEP_MS) {
   if (remoteActive && abs(turnBias) > 0.05) {
     turnCycle();
   } else {
     walkCycle();
   }
   lastWalkTime = now;
 }

 if (balancing && now - lastBalanceTime >= 20) {
   applyBalance();
   lastBalanceTime = now;
 }
}
```