# *TerraLegs: Powered Quadcopter Landing Gear for Uneven Terrain*

Aditya Vishwa, Jonas Plichta, Joe Domke, Angelica Lewis
MECENG 102B: Mechatronics Design, Fall 2025

## Opportunity

Quadcopter landings are vulnerable to tip-over, rotor strike, and instability when operating on uneven and unpredictable terrain. Current drones, both hobby and commercial, use fixed landing gear because of the inherent light weight, low cost, and simple integration. However, rigid landing gear requires flat, obstacle-free surfaces and provides zero ability to adapt to real-world conditions. TerraLegs addresses this limitation by providing a powered, self-leveling landing gear system that allows quadcopters to land safely and reliably in complex terrain conditions, reducing operational risk and increasing potential deployment area. The system is fully modular, enabling it to be mounted to any existing frames with just a few bolts and a power lead.

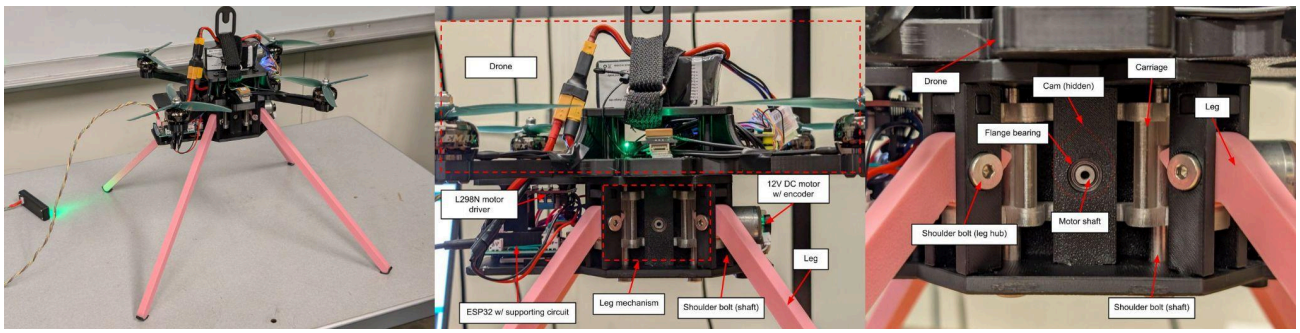
## High-Level Strategy

TerraLegs is a powered landing-gear system that uses a single 12V motor to control 4 legs simultaneously via a worm gear based gearbox that drives a cam and carriage mechanism. Three distinct cam positions, 0°, 90° and 180° correspond to stowed, free-moving, and clamped states respectively. At 0°, the carriage presses levers on the leg hubs to stow the legs near horizontal for flight. At 90°, the carriage is neutral, allowing the legs to passively settle to terrain in landing. At 180°, ratchet style teeth on the carriage mesh with similar teeth on each leg hub, locking legs in fixed positions without continuous power.

An ESP32 Feather V2 runs an event-driven state machine using an internal limit switch for homing and a downward-facing LiDaR sensor to determine several state transitions. Power to the ESP32 is provided by a small battery and to the motor by the main drone battery.

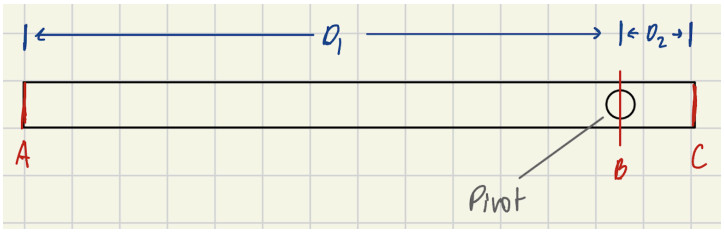
This system was designed to enable quadcopter landings on uneven terrain, intended to accommodate at least ±5 cm terrain variation, complete deployment and locking in <10 s, maintain drone tilt within ±5°, require no holding power when clamped, and remain under 750g. The realized system achieved passive adaptation to ±7.25 cm height variation at 15cm hover, zero holding power when clamped, and had a final mass <650g. The final landing sequence required ~6.1 s to complete, with opportunity to be reduced further, however was unable to stay within the desired tilt limits in all terrain conditions. The discrete locking teeth constrain each leg to 15° increments, resulting in a possible clamping error up to ±7.5° per leg. This quantization effect is greater when clamping nearer to horizontal, where small angular errors produce larger positional changes. Despite this limitation, even-ground landing tests with 15 ± 2 cm never resulted in a final platform tilt greater than 5°.

## Fully Assembled, Integrated Design



# Functional-Critical Decisions and Calculations

These calculations were done using skills from ME C85, introduction to solid mechanics, as well as some empirical engineering assumptions.




**Required Motor Torque:**

The maximum required torque occurs when retracting the legs, as when the legs are locked during landing, most of the force is taken by the ratchet-gear-like mechanism.

Max torque required to retract 2 legs (each cam retracts 2 legs, D1 = 0.15m, m = 0.0238kg):

$$T_{2-legs} = F \times D_1 = m * a * D_1 * sin(90) * 2 = 0.0238 * 9.81 *.15 * 1 * 2 = 0.0350217\ Nm$$

Required force to lift 2 legs (D2 = 0.01m):

$$T_{2-legs} = 0.0350217 = F \times D_2 = F * 0.01 * sin(90) \rightarrow F_{2-legs} = 7.00434N$$

Combined required force 2-carriages must push down with (carriageMass = 0.0125kg):

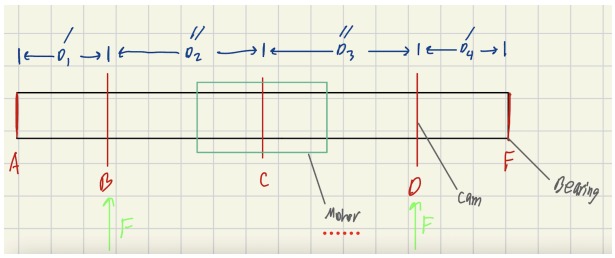
$$2 * F_{2-legs} - carriageMass * 9.81 * 2 = 2 * 7.00434N - 0.0125 * 9.81 * 2 = 13.76343N$$

Conservative estimate of total required motor torque (camRadius = 0.0155m):

$$T_{motor} = F \times camRadius = 13.76343 * 0.0155 * sin(90) \approx 0.21\ N*m$$

**Bearing Load and Plate Stress:**

Due to cam reaction forces being symmetric about the motor shaft, the system was modeled as a simply supported beam, resulting in a max bearing load of $13.76343/2 \approx 6.9N$ per bearing as calculated above.




Calculation of bearing plate stress was done using a pin-in-hole stress approximation and assuming a uniform PLA plate (F = force, D = hole/bearing diameter, T = hole/bearing thickness, 60/360 = contact arc):

$$\sigma_{bearing} \approx F/(D * T * 60/360) = 6.9/(10 * 10^{-3} * 3 * 10^{-3} * 60/360) \approx 0.73\ MPa.$$

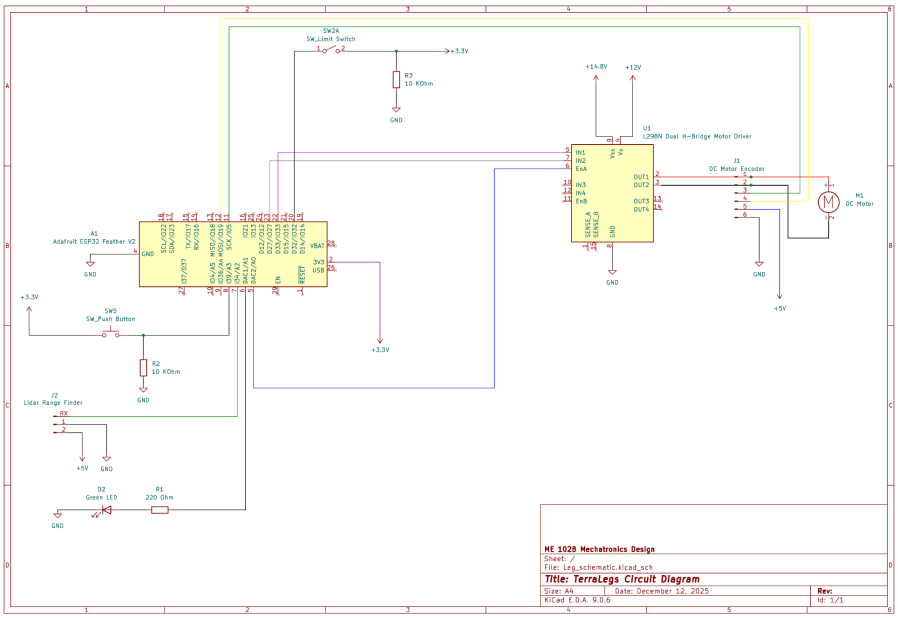
The bearing plate stress was well below the strength of PLA (~ 50 MPa), though we used a uniform composition in our assumptions. Given the high factor of safety there was no concern about material strengths for the expected loadings, supported by no visible damage in any stage of project testing.

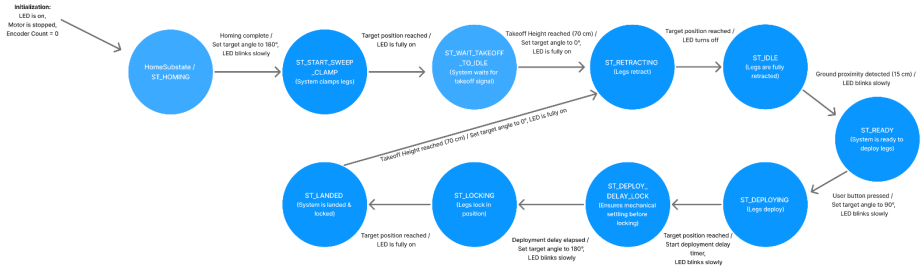
**Motor Decision:**

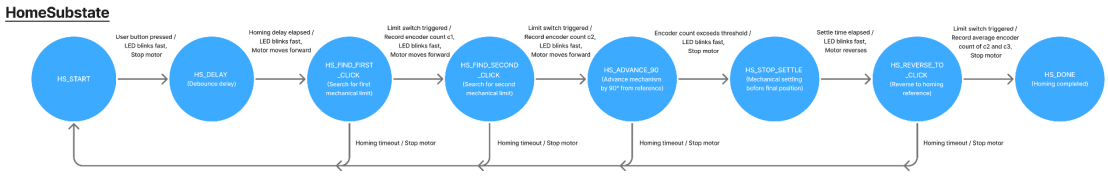
Due to long lead times, the motor was selected before the final design was complete. Early concepts assumed longer legs printed from a denser material, requiring a higher motor torque, and sizing was

intended to be conservative due to dynamic loads and friction not being explicitly modeled in our calculations. With this in mind, we ordered a motor with a stall torque of 2.5 N*m. As a result of the changes in leg design, we had an increased static factor of safety (~12), ensuring reliable operation but leaving room for improvement in speed and weight.

## Circuit and State Transition Diagrams




## Reflection

TerraLegs functioned reliably as an integrated prototype, successfully demonstrating a single-actuator landing gear capable of adapting to adverse terrain. An early decision to prioritize integration allowed for necessary debugging, but due to motor lead-times, our motor was sized on a longer leg design, which was later resized to fit on-hand Bambu P1S build volume. This critical design change reduced torque demand, resulting in an oversized motor limiting potential weight reduction and speed maximization. Future project iterations would include a higher speed motor and machined leg hubs with higher resolution teeth to address quantization error and improve repeatability in higher variability terrain.

# Appendix A: Bill of Materials

| Part name | Vendor | Quantity | Package price | Notes |
|---|---|---|---|---|
| Carriage top | 3D printed | 2 | $0.00 | PLA |
| Carriage left spacer | 3D printed | 2 | $0.00 | PLA |
| Carriage right spacer | 3D printed | 2 | $0.00 | PLA |
| Carriage bottom | 3D printed | 2 | $0.00 | PLA |
| Bottom plate | 3D printed | 1 | $0.00 | PLA |
| Top plate | 3D printed | 1 | $0.00 | PLA |
| Motor mount left | 3D printed | 1 | $0.00 | PLA |
| Motor mount right | 3D printed | 1 | $0.00 | PLA |
| Bearing plate | 3D printed | 2 | $0.00 | PLA |
| Cam | 3D printed | 2 | $0.00 | PLA |
| Flanged bearing | Amazon | 2 | $9.69 | |
| Leg | 3D printed | 4 | $0.00 | PLA |
| Leg mount left | 3D printed | 4 | $0.00 | PLA |
| Leg mount right | 3D printed | 4 | $0.00 | PLA |
| Leg spacer | 3D printed | 4 | $0.00 | PLA |
| 10mm M5 screw - top plate to drone | On hand | 4 | $0.00 | |
| 6mm bore 10mm sleeve bearing - leg hub | Amazon | 4 | $11.89 | |
| 6mm bore 4mm sleeve bearing - carriage | Amazon | 4 | $0.00 | Assorted pack with 10mm sleeve bearings |
| 6mm shoulder bolt - carriage shaft | Amazon | 4 | $8.99 | |
| M5 6mm shaft shoulder screw - Leg hub | Amazon | 4 | $9.49 | |
| M5 hex nut | On hand | 8 | $0.00 | |
| Shaft collar | Amazon | 2 | $8.99 | |
| M6 washer | On hand | 4 | $0.00 | |
| M3 nut | On hand | 4 | $0.00 | |
| M3 screw - carriage | On hand | 4 | $0.00 | |
| M3 screw - motor mount | On hand | 4 | $0.00 | |
| 12V DC motor dual shaft with encoder | Robotshop | 1 | $12.63 | |
| ESP32 Feather V2 | On hand | 1 | $0.00 | |
| L298N driver | On hand | 1 | $0.00 | |
| Lidar rangefinder | On hand | 1 | $0.00 | |
| Green LED | On hand | 1 | $0.00 | |
| Limit switch | On hand | 1 | $0.00 | |
| Micro pushbutton | On hand | 1 | $0.00 | |
| 220 Ohm resistor | On hand | 1 | $0.00 | |
| 10 KOhm resistor | On hand | 4 | $0.00 | |
| 500 mAh Li-Po battery | On hand | 1 | $0.00 | |
| Slide switch | On hand | 1 | $0.00 | |
| Surface mount JST connector | On hand | 10 | $0.00 | |
| | | **Total** | $61.68 | |

# Appendix B: CAD

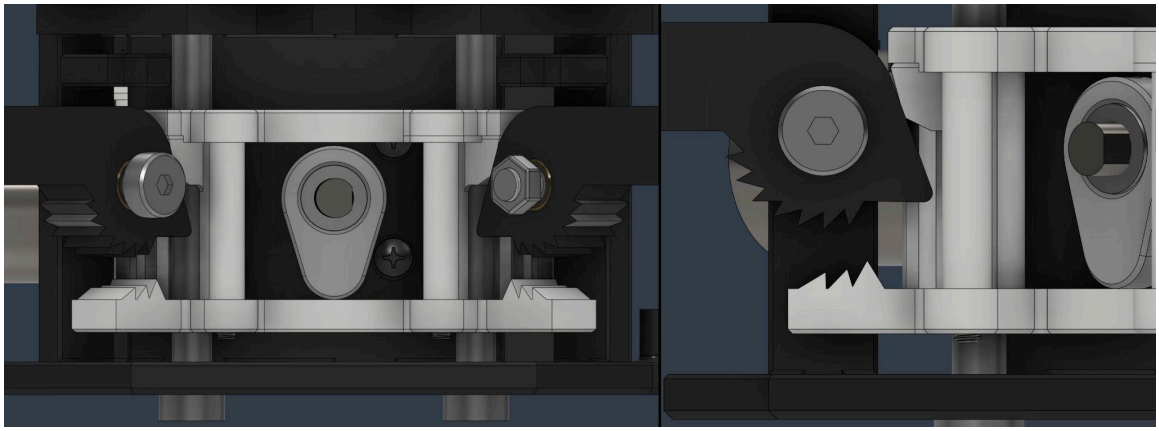


*Figure B1: Cam at 0° position, pressing down on carriage and holding legs in stowed (horizontal) position. Support pieces hidden for visibility.*

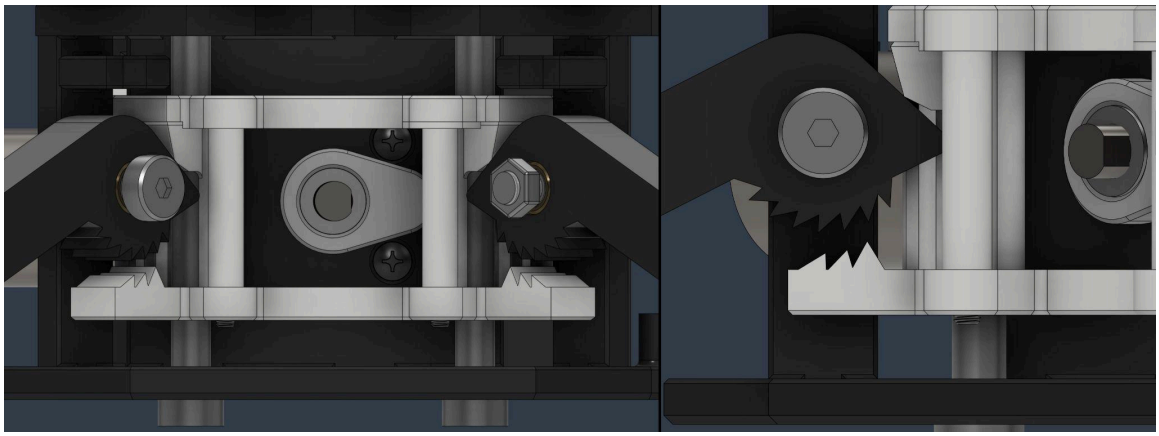


*Figure B2. Cam at 90° position, legs free to move under their own weight. Support pieces hidden for visibility.*

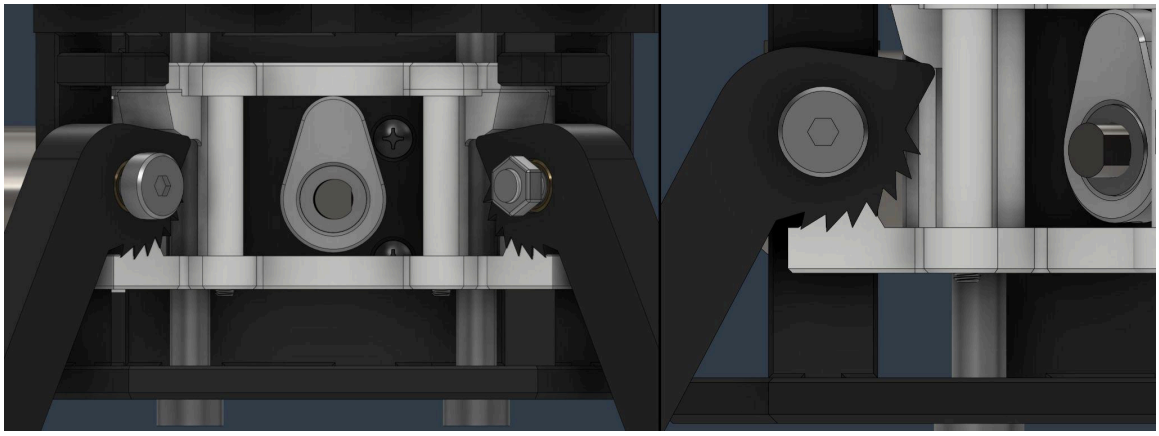


*Figure B3. Cam at 180° position, pressing up on the carriage and locking legs in a clamped position. Support pieces hidden for visibility.*

*Figure B4: Terralegs CAD assembly.*

## Appendix C: Code

```
/// FINAL CODE ////
#include <Arduino.h>
#include <HardwareSerial.h>
#include "esp_timer.h"

// ----------Pin Mapping----------

// Motor driver
constexpr int PIN_MOTOR_ENA = 26; // Blue wire from driver
constexpr int PIN_MOTOR_IN1 = 33; // Purple wire from driver
constexpr int PIN_MOTOR_IN2 = 27; // Grey wire from driver

// Encoder
constexpr int PIN_ENC_A = 5; // Green wire from encoder
constexpr int PIN_ENC_B = 19; // Yellow wire from encoder

// Range Finder
constexpr int PIN_LIDAR_RX = 34; // Green wire from range finder
constexpr int PIN_LIDAR_TX = 13; // White wire from range finder (not used, cut off)
constexpr uint32_t LIDAR_BAUD = 115200;
HardwareSerial LidarSerial(2);

// User input + LED
constexpr int PIN_BTN       = 39; // User controller button input
constexpr int PIN_LED_GREEN = 25; // User controller indicator LED

// Limit switches
constexpr int PIN_LIM1 = 15; // Not used
constexpr int PIN_LIM2 = 32; // Not used
constexpr int PIN_LIM3 = 14; // Used for homing
constexpr int PIN_LIM4 = 37; // Not used

//  Encoder direction fix
constexpr bool ENC_INVERT = true;

// Fixed measured counts/output-rev
constexpr float COUNTS_PER_OUTPUT_REV = 42754.0f; // Encoder count for 6 RPM gear box. 66 RPM gear box
count was 3834.0

// PWM tuning
constexpr int   PWM_MAX      = 255;
constexpr int   PWM_MIN_MOVE = 150;   // no-load
constexpr int   PWM_MIN_LOAD = 175;   // loaded
constexpr float ERR_LOAD_DEG = 25.0f; // above this error assume loaded move
constexpr float DEG_TOL      = 1.0f;  // deadband around target [deg]

//  Height thresholds
constexpr uint16_t HEIGHT_OK_CM      = 15;   // near ground / ready
constexpr uint16_t HEIGHT_TAKEOFF_CM = 70;  // takeoff detection (0.7 m)

//  Homing constants
constexpr int HOMING_PWM = 200; // Homing PWM

// CHANGED timing to background timer to reflect functionality test feedback
// timing previously done with millis()/delay(), now driven from a fixed-rate esp_timer tick
```

```
constexpr uint32_t CTRL_HZ = 100;                // 100 Hz control update
constexpr float    CTRL_DT = 1.0f / (float)CTRL_HZ;
constexpr int64_t  CTRL_PERIOD_US = 1000000LL / CTRL_HZ; // 10,000 us

volatile uint32_t ctrlTicks = 0;
esp_timer_handle_t ctrlTimer = nullptr;

void ctrl_timer_cb(void*) {
  ctrlTicks++;
}

static inline uint32_t nowTick() { return ctrlTicks; }
static inline uint32_t msToTicks(uint32_t ms) { return (ms + 9) / 10; } // 10 ms per tick @ 100 Hz

// ---------------- Encoder ----------------
volatile long encCount = 0;

// Encoder A ISR
void IRAM_ATTR isr_encA() {
  bool a = digitalRead(PIN_ENC_A);
  bool b = digitalRead(PIN_ENC_B);

  long step;

  if (a == b) {
    step = +1;
  } else {
    step = -1;
  }

  if (ENC_INVERT) {
    encCount -= step;
  } else {
    encCount += step;
  }
}

// Encoder B ISR
void IRAM_ATTR isr_encB() {
  bool a = digitalRead(PIN_ENC_A);
  bool b = digitalRead(PIN_ENC_B);

  long step;

  if (a != b) {
    step = +1;
  } else {
    step = -1;
  }

  if (ENC_INVERT) {
    encCount -= step;
  } else {
    encCount += step;
  }
}
```

```
inline float countsToDeg(long c)  { return (c * 360.0f) / COUNTS_PER_OUTPUT_REV; }
inline long  degToCounts(float d) { return lroundf(d * COUNTS_PER_OUTPUT_REV / 360.0f); }

static inline long readEncAtomic() {
  noInterrupts();
  long c = encCount;
  interrupts();
  return c;
}

// ---------------- Lidar ----------------
static const uint8_t HDR = 0x59;
uint16_t tf_dist = 65535;
bool nearGround = false;
bool takeoffHigh = false;

void lidar_poll() {
  static uint8_t buf[9];
  while (LidarSerial.available() >= 9) {
    if (LidarSerial.peek() != HDR) { LidarSerial.read(); continue; }
    LidarSerial.readBytes(buf, 9);
    if (buf[0]==HDR && buf[1]==HDR) {
      uint8_t sum=0; for (int i=0;i<8;i++) sum+=buf[i];
      if ((sum&0xFF)!=buf[8]) continue;
      uint16_t d = (uint16_t)buf[2] | ((uint16_t)buf[3]<<8);
      tf_dist = d;
      nearGround  = (d <= HEIGHT_OK_CM);
      takeoffHigh = (d >= HEIGHT_TAKEOFF_CM);
    }
  }
}

// ---------------- ISR flags ----------------
volatile bool btnEdge  = false; //initialize
volatile bool lim3Edge = false;

void IRAM_ATTR isr_btn()  { btnEdge  = true; } // user button flag
void IRAM_ATTR isr_lim3() { lim3Edge = true; } // homing sequence limit switch flag

// Button debounce
const uint32_t BTN_DEBOUNCE_TICKS = msToTicks(100);
const uint32_t LIM_DEBOUNCE_TICKS = msToTicks(80);

bool checkBtnPressed() {
  static uint32_t last = 0;
  if (!btnEdge) return false;
  btnEdge = false;
  uint32_t t = nowTick();
  if (digitalRead(PIN_BTN) == HIGH && (t - last) > BTN_DEBOUNCE_TICKS) {
    last = t;
    return true;
  }
  return false;
}

// active-LOW limit switch
```

```
bool checkLim3Pressed() {
  static uint32_t last = 0;
  static bool prevPressed = false;

  bool pressed = (digitalRead(PIN_LIM3) == LOW);
  bool newPress = pressed && !prevPressed;
  prevPressed = pressed;

  // Button debounce
  uint32_t t = nowTick();
  if (newPress && (t - last) > LIM_DEBOUNCE_TICKS) {
    last = t;
    return true;
  }
  return false;
}

// ---------------- Motor drivers ----------------
inline void motorStop() {
  digitalWrite(PIN_MOTOR_IN1, LOW);
  digitalWrite(PIN_MOTOR_IN2, LOW);
  analogWrite(PIN_MOTOR_ENA, 0);
}
inline void motorFwd(int pwm) {
  digitalWrite(PIN_MOTOR_IN1, HIGH);
  digitalWrite(PIN_MOTOR_IN2, LOW);
  analogWrite(PIN_MOTOR_ENA, pwm);
}
inline void motorRev(int pwm) {
  digitalWrite(PIN_MOTOR_IN1, LOW);
  digitalWrite(PIN_MOTOR_IN2, HIGH);
  analogWrite(PIN_MOTOR_ENA, pwm);
}

//CONTROL SECTION CHANGED TO REFLECT FUNCTIONALITY TEST FEEDBACK -> now cascaded

// Position loop generates velocity command, velocity loop generates PWM command

float targetDeg = 0.0f;

constexpr float KP_STARTUP_SCALE = 0.5f;  // slower during 0->180 clamp sweep
float kp_scale = 1.0f;  // scaled during startup sweep

long  encPrev = 0;
float velDegPerSec = 0.0f;

constexpr float KP_POS      = 5.0f;
constexpr float VEL_CMD_MAX = 80.0f;

constexpr float KP_VEL      = 2.0f;
constexpr float KI_VEL      = 5.0f;
constexpr float I_VEL_CLAMP = 80.0f;
float velI = 0.0f;

void resetController() {
  encPrev = readEncAtomic();
  velI = 0.0f;
```

```
}

int minPwmForErr(float errDeg) {
  return (fabsf(errDeg) > ERR_LOAD_DEG) ? PWM_MIN_LOAD : PWM_MIN_MOVE;
}

void controlStep_cascaded() {
  long c = readEncAtomic();
  long dc = c - encPrev;
  encPrev = c;

  float pos = countsToDeg(c);
  float ddeg = countsToDeg(dc);
  velDegPerSec = ddeg / CTRL_DT;

  float err = targetDeg - pos;

  if (fabsf(err) <= DEG_TOL) {
    motorStop();
    velI = 0.0f;
    return;
  }

  float velCmd = (KP_POS * kp_scale) * err;
  velCmd = constrain(velCmd, -VEL_CMD_MAX, VEL_CMD_MAX);

  float velErr = velCmd - velDegPerSec;

  velI += velErr * CTRL_DT;
  velI = constrain(velI, -I_VEL_CLAMP, I_VEL_CLAMP);

  float u = KP_VEL * velErr + KI_VEL * velI;

  int pwm = (int)fabsf(u);
  int pwmMin = minPwmForErr(err);

  if (pwm < pwmMin) pwm = pwmMin;
  pwm = constrain(pwm, 0, PWM_MAX);

  if (u > 0) motorFwd(pwm);
  else       motorRev(pwm);
}

// hold check only for state transitions
const uint32_t HOLD_TICKS = msToTicks(250);
uint32_t holdStartTick = 0;

bool atTargetHold() {
  float pos = countsToDeg(encCount);
  float err = targetDeg - pos;
  uint32_t t = nowTick();

  if (fabsf(err) <= DEG_TOL) {
    if (holdStartTick == 0) holdStartTick = t;
    if (t - holdStartTick >= HOLD_TICKS) return true;
  } else {
    holdStartTick = 0;
```

```
  }
  return false;
}

// ---------------- LEDs ----------------
const uint32_t LED_FAST_TICKS = msToTicks(120);
const uint32_t LED_SLOW_TICKS = msToTicks(500);

void ledsBlinkFast() { digitalWrite(PIN_LED_GREEN, ((nowTick()/LED_FAST_TICKS)&1)); }
void ledsBlinkSlow() { digitalWrite(PIN_LED_GREEN, ((nowTick()/LED_SLOW_TICKS)&1)); }
void ledsOn()        { digitalWrite(PIN_LED_GREEN, HIGH); }
void ledsOff()       { digitalWrite(PIN_LED_GREEN, LOW); }

// ---------------- Homing ----------------
enum HomeSubstate {
  HS_START,
  HS_DELAY,
  HS_FIND_FIRST_CLICK,
  HS_FIND_SECOND_CLICK,
  HS_ADVANCE_90,
  HS_STOP_SETTLE,
  HS_REVERSE_TO_CLICK,
  HS_DONE
};
HomeSubstate hs = HS_START;

long c1 = 0, c2 = 0, c3 = 0;
long counts90 = 0;

const uint32_t HOME_DELAY_TICKS     = msToTicks(2500);
const uint32_t HOMING_TIMEOUT_TICKS = msToTicks(12000);
const uint32_t SETTLE_TICKS         = msToTicks(100);

uint32_t home_t0 = 0;
uint32_t hs_t0 = 0;

void homing_step() {
  uint32_t t = nowTick();

  switch (hs) {

    case HS_START:
      ledsOn();
      motorStop();
      home_t0 = 0;
      hs_t0 = 0;
      if (checkBtnPressed()) {
        hs = HS_DELAY;
        Serial.println("[HOMING] Start homing sequence.");
      }
      break;

    case HS_DELAY:
      if (home_t0 == 0) home_t0 = t;
      ledsBlinkFast();
      motorStop();
      if (t - home_t0 >= HOME_DELAY_TICKS) {
```

```
    hs = HS_FIND_FIRST_CLICK;
    hs_t0 = t;
    Serial.println("[HOMING] Spin to first click.");
  }
  break;

case HS_FIND_FIRST_CLICK:
  ledsBlinkFast();
  motorFwd(HOMING_PWM);
  if (checkLim3Pressed()) {
    noInterrupts(); c1 = encCount; interrupts();
    hs = HS_FIND_SECOND_CLICK;
    hs_t0 = t;
    Serial.println("[HOMING] First click. Continue to next click.");
  }
  if (t - hs_t0 > HOMING_TIMEOUT_TICKS) {
    motorStop();
    Serial.println("[HOMING FAIL] Timeout at first click.");
    hs = HS_START;
  }
  break;

case HS_FIND_SECOND_CLICK:
  ledsBlinkFast();
  motorFwd(HOMING_PWM);
  if (checkLim3Pressed()) {
    noInterrupts(); c2 = encCount; interrupts();
    counts90 = degToCounts(90.0f);
    hs = HS_ADVANCE_90;
    hs_t0 = t;
    Serial.println("[HOMING] Second click. Advance +90.");
  }
  if (t - hs_t0 > HOMING_TIMEOUT_TICKS) {
    motorStop();
    Serial.println("[HOMING FAIL] Timeout at second click.");
    hs = HS_START;
  }
  break;

case HS_ADVANCE_90:
  ledsBlinkFast();
  motorFwd(HOMING_PWM);
  if (labs(encCount - c2) >= counts90) {
    motorStop();
    hs = HS_STOP_SETTLE;
    hs_t0 = t;
    Serial.println("[HOMING] +90 reached. Reverse to click.");
  }
  if (t - hs_t0 > HOMING_TIMEOUT_TICKS) {
    motorStop();
    Serial.println("[HOMING FAIL] Timeout during +90.");
    hs = HS_START;
  }
  break;

case HS_STOP_SETTLE:
  ledsBlinkFast();
```

```
      motorStop();
      if (t - hs_t0 >= SETTLE_TICKS) {
        hs = HS_REVERSE_TO_CLICK;
        hs_t0 = t;
      }
      break;

    case HS_REVERSE_TO_CLICK:
      ledsBlinkFast();
      motorRev(HOMING_PWM);
      if (checkLim3Pressed()) {
        noInterrupts(); c3 = encCount; interrupts();

        long c0 = lroundf(0.5f * ((float)c2 + (float)c3));
        noInterrupts(); encCount -= c0; interrupts();

        motorStop();
        hs = HS_DONE;
        Serial.println("[HOMING] Zero set. Done.");
      }
      if (t - hs_t0 > HOMING_TIMEOUT_TICKS) {
        motorStop();
        Serial.println("[HOMING FAIL] Timeout reversing to click.");
        hs = HS_START;
      }
      break;

    case HS_DONE:
      motorStop();
      break;
  }
}

// ---------------- States ----------------
enum State {
  ST_HOMING,
  ST_START_SWEEP_CLAMP,
  ST_WAIT_TAKEOFF_TO_IDLE,
  ST_RETRACTING,
  ST_IDLE,
  ST_READY,
  ST_DEPLOYING,
  ST_DEPLOY_DELAY_LOCK,
  ST_LOCKING,
  ST_LANDED
};
State st = ST_HOMING;

const uint32_t DEPLOY_DELAY_TICKS = msToTicks(100);
uint32_t deployDelay_t0 = 0;

// ---------------- Setup ----------------
void setup() {
  Serial.begin(115200);

  pinMode(PIN_MOTOR_IN1, OUTPUT);
  pinMode(PIN_MOTOR_IN2, OUTPUT);
```

```
  pinMode(PIN_MOTOR_ENA, OUTPUT);
  motorStop();

  pinMode(PIN_ENC_A, INPUT_PULLUP);
  pinMode(PIN_ENC_B, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(PIN_ENC_A), isr_encA, CHANGE);
  attachInterrupt(digitalPinToInterrupt(PIN_ENC_B), isr_encB, CHANGE);

  pinMode(PIN_BTN, INPUT);
  attachInterrupt(digitalPinToInterrupt(PIN_BTN), isr_btn, RISING);

  pinMode(PIN_LIM3, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(PIN_LIM3), isr_lim3, FALLING);

  pinMode(PIN_LED_GREEN, OUTPUT);
  ledsOff();

  LidarSerial.begin(LIDAR_BAUD, SERIAL_8N1, PIN_LIDAR_RX, PIN_LIDAR_TX);

  resetController();

  esp_timer_create_args_t args = {};
  args.callback = &ctrl_timer_cb;
  args.dispatch_method = ESP_TIMER_TASK;
  args.name = "ctrlTick";

  if (esp_timer_create(&args, &ctrlTimer) != ESP_OK) {
    Serial.println("[TIMER FAIL] esp_timer_create failed");
  } else {
    if (esp_timer_start_periodic(ctrlTimer, CTRL_PERIOD_US) != ESP_OK) {
      Serial.println("[TIMER FAIL] esp_timer_start_periodic failed");
    }
  }

  Serial.println("Landing Gear: homing + tick-driven cascaded controller.");
}

// ---------------- Loop ----------------
void loop() {
  lidar_poll();

  switch (st) {

    case ST_HOMING:
      homing_step();
      if (hs == HS_DONE) {
        targetDeg = 180.0f; // clamp legs in down position
        kp_scale = KP_STARTUP_SCALE;  // slow sweep
        resetController();
        st = ST_START_SWEEP_CLAMP;
        Serial.println("[START] Sweeping to 180 clamp.");
      }
      break;

    case ST_START_SWEEP_CLAMP:
      ledsBlinkSlow();
      if (atTargetHold()) {
```

```
      kp_scale = 1.0f;
      st = ST_WAIT_TAKEOFF_TO_IDLE;
      Serial.println("[START] Clamped at 180. Waiting takeoff.");
    }
    break;

  case ST_WAIT_TAKEOFF_TO_IDLE:
    ledsOn();
    if (takeoffHigh) {
      targetDeg = 0.0f;
      kp_scale = 1.0f;
      resetController();
      st = ST_RETRACTING;
      Serial.println("[TAKEOFF] Retracting to 0.");
    }
    break;

  case ST_RETRACTING:
    ledsOn();
    if (atTargetHold()) {
      st = ST_IDLE;
      Serial.println("[IDLE] Legs up at 0.");
    }
    break;

  case ST_IDLE:
    ledsOff();
    if (nearGround) {
      st = ST_READY;
      Serial.println("[READY] Near ground. Waiting button.");
    }
    break;

  case ST_READY:
    ledsBlinkSlow();
    if (checkBtnPressed()) {
      targetDeg = 90.0f;
      resetController();
      st = ST_DEPLOYING;
      Serial.println("[DEPLOY] Going to 90.");
    }
    break;

  case ST_DEPLOYING:
    ledsBlinkSlow();
    if (atTargetHold()) {
      deployDelay_t0 = nowTick();
      st = ST_DEPLOY_DELAY_LOCK;
      Serial.println("[DEPLOY] At 90. Delay then lock.");
    }
    break;

  case ST_DEPLOY_DELAY_LOCK:
    ledsBlinkSlow();
    if (nowTick() - deployDelay_t0 >= DEPLOY_DELAY_TICKS) {
      targetDeg = 180.0f;
      resetController();
```

```
      st = ST_LOCKING;
      Serial.println("[LOCK] Locking to 180.");
    }
    break;

  case ST_LOCKING:
    ledsBlinkSlow();
    if (atTargetHold()) {
      st = ST_LANDED;
      Serial.println("[LANDED] Locked at 180. Waiting takeoff.");
    }
    break;

  case ST_LANDED:
    ledsOn();
    if (takeoffHigh) {
      targetDeg = 0.0f;
      resetController();
      st = ST_RETRACTING;
      Serial.println("[TAKEOFF] Retracting to 0.");
    }
    break;
}

// controller always on after homing
static uint32_t ticksServiced = 0;
uint32_t tNow = nowTick();

if (st != ST_HOMING) {
  while (ticksServiced != tNow) {
    ticksServiced++;
    controlStep_cascaded();
  }
} else {
  ticksServiced = tNow;
}

// status print
const uint32_t PRINT_EVERY_TICKS = msToTicks(500);
static uint32_t tLast = 0;

if (tNow - tLast > PRINT_EVERY_TICKS) {
  tLast = tNow;
  float pos_raw = countsToDeg(encCount);

  Serial.print("state=");
  switch(st){
    case ST_HOMING: Serial.print("HOMING"); break;
    case ST_START_SWEEP_CLAMP: Serial.print("START_SWEEP_CLAMP"); break;
    case ST_WAIT_TAKEOFF_TO_IDLE: Serial.print("WAIT_TAKEOFF_TO_IDLE"); break;
    case ST_RETRACTING: Serial.print("RETRACTING"); break;
    case ST_IDLE: Serial.print("IDLE"); break;
    case ST_READY: Serial.print("READY"); break;
    case ST_DEPLOYING: Serial.print("DEPLOYING"); break;
    case ST_DEPLOY_DELAY_LOCK: Serial.print("DEPLOY_DELAY_LOCK"); break;
    case ST_LOCKING: Serial.print("LOCKING"); break;
    case ST_LANDED: Serial.print("LANDED"); break;
```

```
    }
    Serial.print(" | counts=");
    Serial.print(encCount);
    Serial.print(" | pos=");
    Serial.print(pos_raw,1);
    Serial.print("° tgt=");
    Serial.print(targetDeg,1);
    Serial.print("° | near=");
    Serial.print(nearGround);
    Serial.print(" | high=");
    Serial.print(takeoffHigh);
    Serial.print(" | dist=");
    Serial.print(tf_dist);
    Serial.println("cm");
  }
}
```