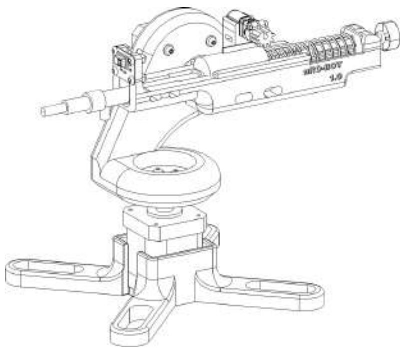
By: Menooa Avrand, Eyan Documet, James Gotesky, Rafael Petrosian



1. Opportunity

"For firefighters or power companies who are operating in hazardous and/or remote fire conditions, its an autonomous cobot that extinguishes small fires via manual or autonomous operation; it can also be used to execute a preventive application of retardant."

2. High-Level Strategy and Functional Outcomes

πRo-BOT is a 2.5DoF robotic water gun intended for autonomous detection and extinguishing of flames, namely in remote and/or dangerous environments. **It operates under the following strategy:**

1. The turret enters a passive patrol mode, autonomously sweeping its field of view using an IR camera to monitor for thermal anomalies.
2. The IR camera, interfaced via an ESP32, detects elevated thermal signatures indicative of potential fire sources and computes their trajectories in real time.
3. Upon detection of a thermal threat, the system engages fire retardant mechanisms, actuating suppression until the thermal signature subsides below a safe threshold.

A manual override capability is provided for human operators to directly control detection or suppression systems as needed–this is controlled remotely by a wireless controller. Furthermore, a preventative regime allows scheduled applications in high-risk zones, where an agency can have πRo apply retardant in a pre-described pattern.

In our original proposal, we stated the following goals:

1. Accurate and precise firing of a stream of water up to a distance of 6 meters
2. Fully functional automatic and manual regimes
3. Automated detection of sources of heat, positioning, and firing until extinguished.
4. Seamless transition to manual input, allowing for natural control.

Due to limitations in the resolution of the Thermal camera we selected, our autonomous mode is only able to accomplish automatic extinguishing at roughly 1/2 our goal. However, in manual mode, the range we set out to achieve is possible with a skilled operator.

3. Integrated Device Overview

Figure 1: Our fully-assembled device, with components labeled.

The device has two joints and a transmission: joint A ("waist"/yaw), joint B ("wrist"/pitch), and transmission C (firing state), giving πRo-BOT 2.5 degrees of freedom.

4. Function-Critical Decisions and Calculations

4.1. Wrist and Waist Motors: NEMA17

When designing our robotic transmission, our group spent significant time debating using Servo Motors or Stepper motors. Ultimately we opted to use **NEMA17 stepper motors** for the following reasons:

- Stepper motors offer the flexibility of continuous rotation, which is not possible with servo motors.
- The compliance of stepper motors creates an inherent safety-factor when dealing with human operators.

We verified our choice under the following hand-calculations:

- Holding torque (specs): $\tau_{max} = 55\,\mathrm{N\,cm}$
- Axial load rading (specs): $P_{max} = 10\,\mathrm{N}$
- Max. applied torque (CAD): $\tau = 41.15\,\mathrm{mm} \cdot 21\,\mathrm{N} \approx 8.64\,\mathrm{N\,cm} \leq \tau_{max}$
- Max. axial load (CAD): $P = 0.667\,\mathrm{kg} \cdot 9.81\,\mathrm{m/s^2} \approx 6.54\,\mathrm{N} \leq P_{max}$

Since both our maximum axial load and torque are significantly less then what's rated on the 42*48 NEMA17's spec-sheet (Stepper Motor Online), we know it's safe to use these motors for our application. Motor speed was not considered, as our device focuses more on precision that rapidity.

4.2. Firing Transmission: Polulu Motor

For our firing mechanism, we had to chose a motor that could deliver the very high torque the original FunWee gun delivered via its gear reduction. We also wanted to implement a design with a much smaller volume, meaning we had to use a direct-drive solution. For this case, the **Polulu 298:1 Gear-Reduction Motor** was an obvious choice.

- A gear reduction is built-in to the motor in an extremely consise volume, allowing very simple design and integration to FunWee's Firing Assembly.
- Sans encoder, the Polulu motor was notably easy to integrate into our circuitry; using a MOSFET as a "driver," we're able to power the smaller motor using the same input as our Wrist and Waist.

The rack and pinion on the off-the-shelf water gun was operated using a 3.7V toy motor, with a likely maximum radial load rating of $20.00\,\mathrm{N\,mm}$. Although we did not characterize our spring stiffness in order to calculate the radial load, compared to an average 3.7V toy motor, our motor is capable of creating the necessary drive to move the rack and pinion. Our 298:1 has a maximum load (at ~60 RPM) of $196.2\,\mathrm{N\,mm}$ (Polulu).

4.3. Material Choice: PLA and Metal

We selected PLA for structural parts to enable rapid, low-cost 3D printing during iteration. Primitive strength testing (trying to break some test pieces with our bare hands) showed suprising reliability when printed at approx. 15% 3D honeycomb infill.

We used metal flanged shaft couplers to maintain metal-metal contacts between our NEMA17s and our robotic segments.

5. Finalized Diagrams

5.1. Circuit Diagrams

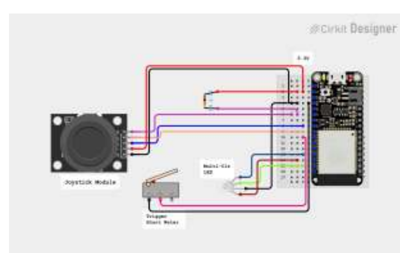


Figure 2: Wiring diagram for the remote controller.

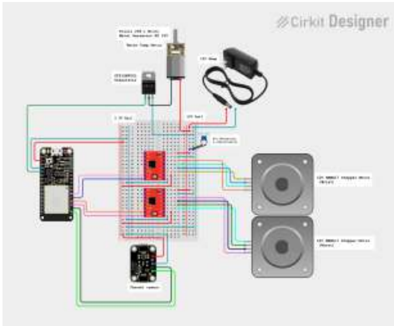


Figure 3: Wiring diagram for the water gun.

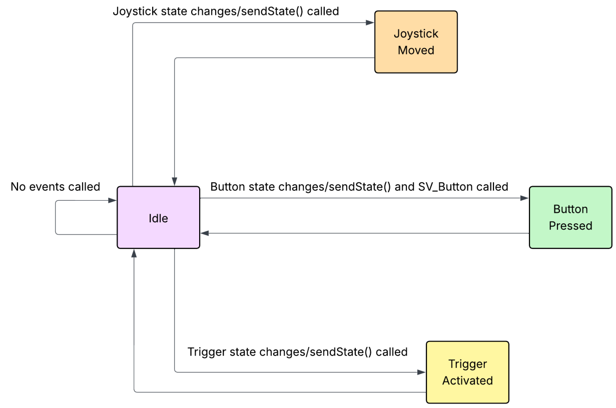


Figure 5: State-transition diagram of the remote control.

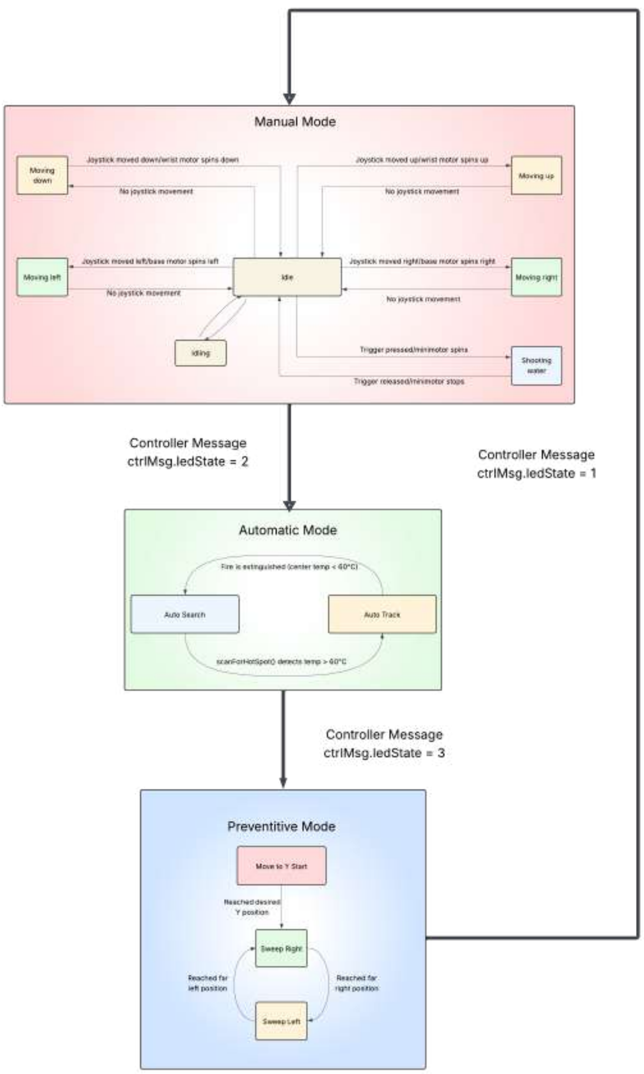


Figure 4: State-transition diagram of the turret.

6. Reflections and Recommendations

If we had an opportunity to continue development (or start over), it would be interesting to try implementing a WIFI-based solution for the remote control aspect of our project. In practice, we'd like the πRo-BOT to be operated remotely from anywhere in the world, as our project goals state it's intended for remote and/or dangerous areas.

For future students, we strongly recommend using off-the-shelf components where possible to avoid "re-inventing the wheel"–this was a significant boon for our team. Dividing labor where possible is also helpful, as long as team-wide syncs are frequent.

Reflecting on our project outcomes, our team is highly satisfied with the final implementation of πRo-BOT. We successfully achieved all core objectives in a robust and sleek system while meeting all major course requirements.

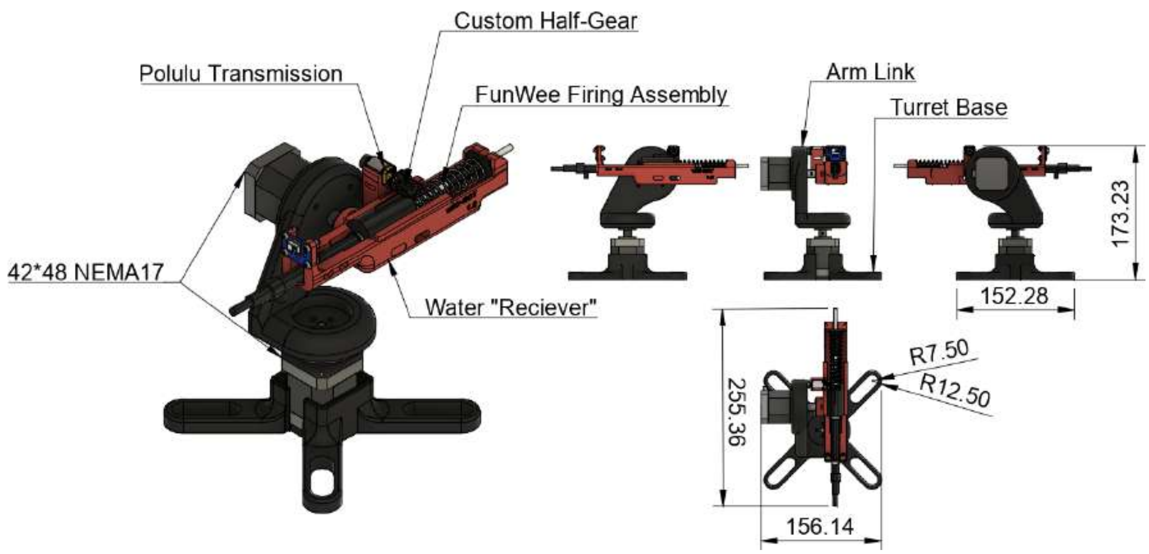
Appendix A: Bill of Materials

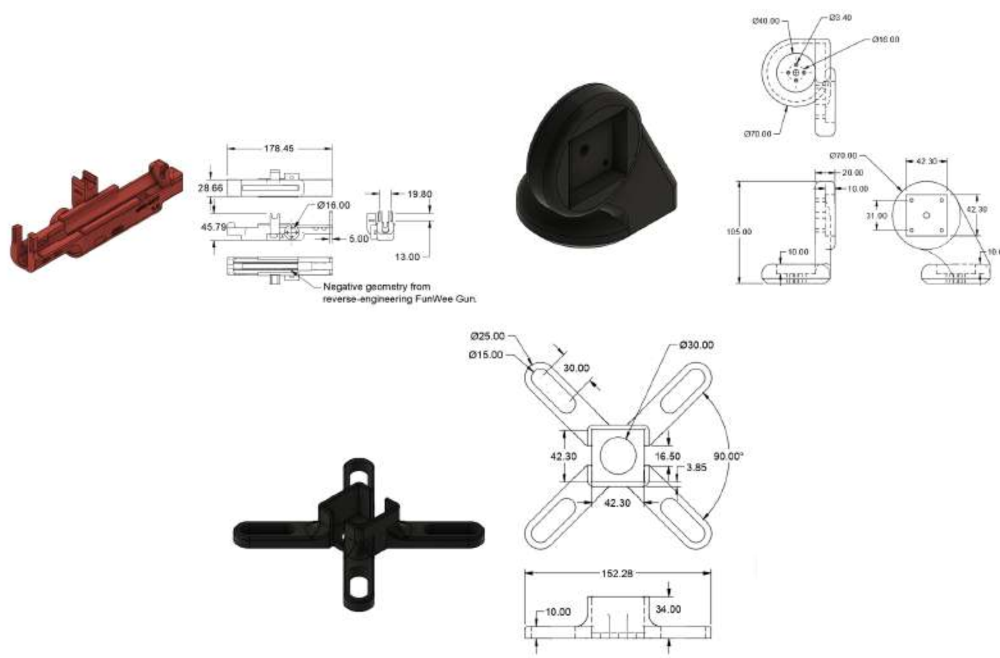
Item Name	Source	Quantity	Unit Price	Total Cost
Nema 17 Stepper Motor	Amazon	2	\$10.99	\$21.98
A4988 Stepper Motor Driver (2pc)	Amazon	2	\$3.80	\$7.60
MLX90640 24x32 Thermal Cam	Adafruit	1	\$74.95	\$74.95
Adafruit ESP32	MicroKit	2	\$0.00	\$0.00
Electric Water Gun	Amazon	1	\$35.99	\$35.99
5mm Coupling Shafts	Amazon	1	\$7.99	\$7.99
Power Supply Adapter	Amazon	1	\$13.99	\$13.99
MOSFET Transistor	ME102B Lab Room	1	\$0.00	\$0.00
Capacitor (220μF)	ME102B Lab Room	1	\$0.00	\$0.00
PLA Filament (2pak)	Amazon	1	\$22.98	\$22.98
281:1 Micro Metal Gearmotor HP 6V	Amazon	1	\$8.91	\$8.91
Dual Axis Joystick Module	Already Owned	1	\$0.00	\$0.00
Limit Switch	Amazon	1	\$5.99	\$5.99
M3 Plain Washers	Ace Hardware	8	\$0.10	\$0.80
M3x0.5 Hex-Head Nuts	Ace Hardware	8	\$0.35	\$2.80
M3x0.5x16 Hex Bolts	Ace Hardware	4	\$0.85	\$3.40
M3x0.5x30 Hex Bolts	Ace Hardware	4	\$0.85	\$3.40
M3x0.5x10 Hex Bolts	Ace Hardware	4	\$0.85	\$3.40
M3x0.5x4 Hex Bolts	Ace Hardware	4	\$0.85	\$3.40
M2x0.4x12 Hex Bolt	Ace Hardware	1	\$0.10	\$0.10
M2x0.4 Hex-Head Nut	Ace Hardware	1	\$0.10	\$0.10
Wiring (Various)	ME102B Lab Room	n/a	\$0.00	\$0.00
Polyethylene Tubing	Amazon	1	\$9.58	\$9.58
1 Gallon Water Jug	Already Owned	1	\$0.00	\$0.00
Zip Ties (Various)	Already Owned	n/a	\$0.00	\$0.00
USB Type-C Cable	Already Owned	2	\$0.00	\$0.00
Breadboard (Various Sizes)	Already Owned	2	\$0.00	\$0.00
PCB Prototyping Board	Already Owned	1	\$0.00	\$0.00
Cable Sleeving	Already Owned	n/a	\$0.00	\$0.00
Waterproof Junction Box	Amazon	1	\$9.99	\$9.99
RGB LED	MicroKit	n/a	\$0.00	\$0.00
Electrical Tape	Already Owned	n/a	\$0.00	\$0.00
Total:		58	n/a	\$237.35

Appendix B: CAD Design & Digital Twinning

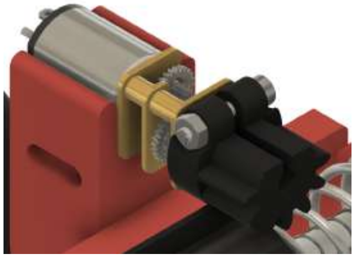
Our entire assembly was designed and executed in Fusion360. Material properties (mass, density, length, etc.) were derived from quantities as-described by our virtual twin.

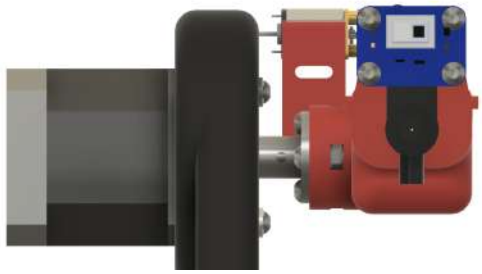
B.1. Overall Turret Layout and Major Sub-Components





B.2. Transmission Close-Ups

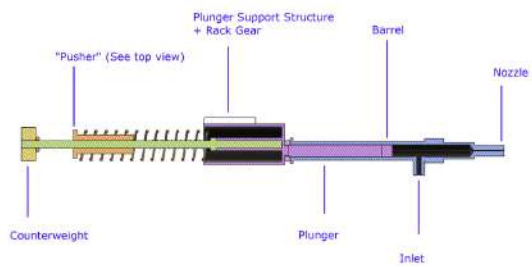




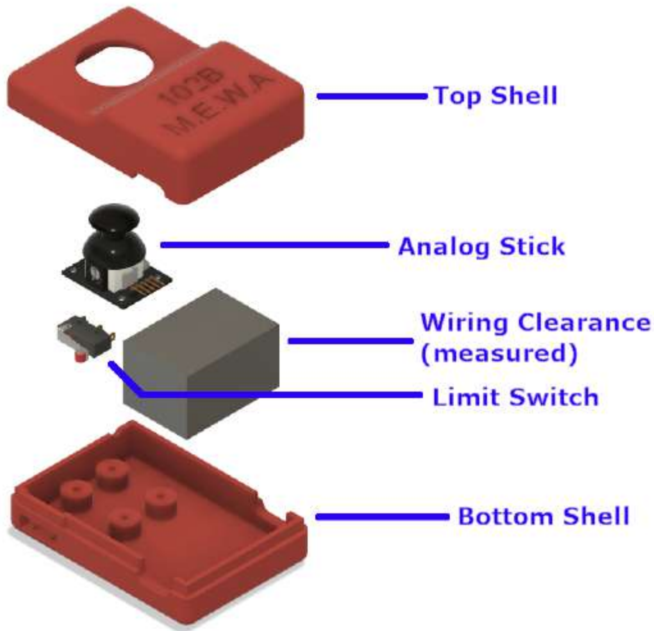
From left to right, the firing transmission, wrist joint, and waist joint are displayed, each serving to control the firing mechanism, pitch, and yaw, respectively.

B.3. FunWee Water Gun Internal Firing Assembly

The water-gun assembly was reverse engineered and cannabilized from an off-the-shelf toy water gun. This is the digital twin we produced from that assembly.



B.4. Remote Control



B.5. Attribution

Our CAD model incorporates reference geometries sourced from GrabCAD. The following attributions are made to the original authors of the reference geometries used:

- Ramirez Escobar, Aldahir. *AMG8833*. 27 Apr. 2020. GrabCAD. https://grabcad.com/library/amg8833-2.
- Hentschke, Steve. *N20 Mini Micro Metal Gear Motor with Encoder*. 27 Oct. 2020. GrabCAD. https://grabcad.com/library/n20-mini-micro-metal-gear-motor-with-encoder-1.
- Baylav, Barış. *Keyes Joystick HW-504*. 21 May 2020. GrabCAD. https://grabcad.com/library/keyes-joystick-hw-504-1.
- Krymoff, Pavel. *Microswitch MSW-13-17*. 16 Mar. 2023. GrabCAD. https://grabcad.com/library/microswitch-msw-13-17-1.

Appendix C: Code Implementation

C.1. Remote Control – Event-Driven Implementation

```
#include <esp_now.h>
#include <WiFi.h>
#include <Arduino.h>
#include <esp_timer.h>   // ESP timer API for debounce

// ------- Pin definitions -------
const int JOYSTICK_SWITCH_PIN = 25;  // Button pin
const int JOYSTICK_X_PIN      = 36;
const int JOYSTICK_Y_PIN      = 39;
const int LIMIT_SWITCH_PIN    = 4;

const int LED_R_PIN = 7;
const int LED_G_PIN = 8;
const int LED_B_PIN = 21;

int ledState = 1;  // Initial LED state

// ------- Debounce Variables -------
volatile bool buttonIsPressed = false;
volatile bool DEBOUNCEflag = false;
const int debounceDelay = 200;   // Debounce cooldown in ms
esp_timer_handle_t debounceTimer = NULL;  // Timer handle

// ------- Joystick calibration values -------
const int midX = 1840;
const int midY = 1800;
const int range = 2000;

// ------- ControllerMessage structure -------
typedef struct __attribute__((packed)) {
  int  joyX;
  int  joyY;
  bool trigger;
  int  ledState;
} ControllerMessage;

// ------- Turret MAC Address (for sending controller data) -------
uint8_t turretAddress[] = {0x0C, 0x8B, 0x95, 0x96, 0xB8, 0x64};

// ------- Function to update the RGB LED -------
void updateLED() {
  digitalWrite(LED_R_PIN, ledState == 1 ? HIGH : LOW);
  digitalWrite(LED_G_PIN, ledState == 2 ? HIGH : LOW);
  digitalWrite(LED_B_PIN, ledState == 3 ? HIGH : LOW);
}

// ------- Timer Callback -- Called when debounce period expires -------
// The esp_timer callback clears the debounce flag so that new button presses are allowed.
```

```
void IRAM_ATTR onTime(void* arg) {
  DEBOUNCEflag = false;   // End cooldown period
}

// ------- Button ISR using esp_timer for Debounce -------
// This ISR fires when the button press (FALLING edge) is detected.
// If not in a debounce period, it sets the button event flag and starts the debounce timer.
void IRAM_ATTR isr() {
  if (!DEBOUNCEflag) {
    buttonIsPressed = true;      // Flag a valid press
    DEBOUNCEflag = true;         // Begin debounce cooldown
    // Start the one-shot timer: debounceDelay in ms (converted to microseconds)
    esp_timer_start_once(debounceTimer, debounceDelay * 1000);
  }
}

// ------- Event Checker Function -------
// Returns true if a button press was detected and the debounce period has expired.
bool CheckForButtonPress() {
  return buttonIsPressed && !DEBOUNCEflag;
}

// ------- Service Function for Button Event -------
// Handles the button press event by toggling the LED state and resetting the flag.
void ButtonResponse() {
  buttonIsPressed = false;              // Clear the press flag
  ledState = (ledState % 3) + 1;          // Cycle through LED states 1-3
  updateLED();
  Serial.println("Button Press Handled: LED toggled.");
}

// ------- Global variables to store previous state for change detection -------
int prevJoyX = 0;
int prevJoyY = 0;
int prevLED = 0;
int prevTrigger = -1;   // Initialized to a value that won't equal digitalRead

// ----- Thermal Camera Reception Variables -----
#define TOTAL_PIXELS 768
#define CHUNK_SIZE   60
#define TOTAL_CHUNKS ((TOTAL_PIXELS + CHUNK_SIZE - 1) / CHUNK_SIZE)

float frame[TOTAL_PIXELS];
bool receivedChunks[TOTAL_CHUNKS] = {0};
uint8_t receivedCount = 0;

#pragma pack(push, 1)
typedef struct {
  uint8_t chunkIndex;
  uint8_t totalChunks;
  float chunkData[CHUNK_SIZE];
} FrameChunk;
#pragma pack(pop)
```

```

// ------- ESP-NOW Receive Callback for Thermal Camera Data -------
void OnDataRecv(const esp_now_recv_info_t *info, const uint8_t *data, int len) {
  if (len != sizeof(FrameChunk)) return; // Ignore messages that don't match the expected frame chunk size

  FrameChunk chunk;
  memcpy(&chunk, data, sizeof(FrameChunk));

  uint16_t offset = chunk.chunkIndex * CHUNK_SIZE;
  uint8_t actualSize = min(CHUNK_SIZE, TOTAL_PIXELS - offset);
  memcpy(&frame[offset], chunk.chunkData, actualSize * sizeof(float));

  if (!receivedChunks[chunk.chunkIndex]) {
    receivedChunks[chunk.chunkIndex] = true;
    receivedCount++;
  }

  if (receivedCount == TOTAL_CHUNKS) {
    Serial.println("FRAME_START");
    for (int i = 0; i < TOTAL_PIXELS; i++) {
      Serial.print(frame[i], 1);
      Serial.print(" ");
      if ((i + 1) % 32 == 0) Serial.println();
    }
    Serial.println("FRAME_END");
    memset(receivedChunks, 0, sizeof(receivedChunks));
    receivedCount = 0;
  }
}

void setup() {
  Serial.begin(115200);

  // Setup pin modes:
  pinMode(JOYSTICK_SWITCH_PIN, INPUT_PULLUP);
  pinMode(JOYSTICK_X_PIN, INPUT);
  pinMode(JOYSTICK_Y_PIN, INPUT);
  pinMode(LIMIT_SWITCH_PIN, INPUT_PULLUP);

  pinMode(LED_R_PIN, OUTPUT);
  pinMode(LED_G_PIN, OUTPUT);
  pinMode(LED_B_PIN, OUTPUT);

  updateLED();

  // Initialize WiFi and ESP-NOW:
  WiFi.mode(WIFI_STA);
  Serial.println("Controller ESP32 Ready");

  if (esp_now_init() != ESP_OK) {
    Serial.println(" ESP-NOW Init Failed");
    return;
```

```
  }

  // Add peer for turret (for sending controller messages)
  esp_now_peer_info_t peerInfo = {};
  memcpy(peerInfo.peer_addr, turretAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;
  if (esp_now_add_peer(&peerInfo) != ESP_OK) {
    Serial.println(" Failed to add turret peer");
    return;
  }

  // Register receive callback for thermal camera frame chunks
  esp_now_register_recv_cb(OnDataRecv);

  // Create the ESP timer for debounce functionality.
  esp_timer_create_args_t debounceTimerArgs = {
    .callback = onTime,
    .arg = NULL,
    .dispatch_method = ESP_TIMER_TASK,  // Execute the callback in the timer task context
    .name = "debounceTimer"
  };
  esp_timer_create(&debounceTimerArgs, &debounceTimer);

  // Attach the external interrupt to the joystick button pin.
  // With INPUT_PULLUP, a button press brings the pin from HIGH to LOW (FALLING edge).
  attachInterrupt(digitalPinToInterrupt(JOYSTICK_SWITCH_PIN), isr, FALLING);
}

void loop() {
  // Process button events using the event-driven approach.
  if (CheckForButtonPress()) {
    ButtonResponse();
  }

  // Process joystick readings:
  int rawX = analogRead(JOYSTICK_X_PIN) - midX;
  int rawY = analogRead(JOYSTICK_Y_PIN) - midY;

  // Map the raw values to a range of -1000 to +1000.
  int joyX = map(rawX, -1840, 2255, -1000, 1000);
  int joyY = map(rawY, -1800, 2295, -1000, 1000);

  // Apply deadzone filtering.
  const int deadzone = 150;
  if (abs(joyX) < deadzone) joyX = 0;
  if (abs(joyY) < deadzone) joyY = 0;

  // Read the limit switch (trigger) state.
  int triggerState = digitalRead(LIMIT_SWITCH_PIN);

  // Pack the joystick and button info into the message structure.
  ControllerMessage msg;
```

```
  msg.joyX = joyX;
  msg.joyY = joyY;
  msg.trigger = (triggerState == LOW);
  msg.ledState = ledState;

  // Send the controller message via ESP-NOW.
  esp_err_t result = esp_now_send(turretAddress, (uint8_t *)&msg, sizeof(msg));
  if (result != ESP_OK) {
    Serial.printf(" Failed to send message: %d\n", result);
  }

  delay(50);
}
```

C.2. Water Turret – Event-Driven Implementation

```
#include <esp_now.h>
#include <WiFi.h>
#include <Wire.h>
#include <Adafruit_MLX90640.h>
#include <math.h>

#define TOTAL_PIXELS 768
#define CHUNK_SIZE 60
#define TOTAL_CHUNKS ((TOTAL_PIXELS + CHUNK_SIZE - 1) / CHUNK_SIZE)

// Motor control pin definitions
#define BIN_1 4       // PWM control for motor (clockwise)
#define BIN_2 5       // Direction control, kept LOW for clockwise rotation
#define LED_PIN 13    // LED indicator pin
#define ENCODER_A 12  // Encoder channel A
#define ENCODER_B 27  // Encoder channel B

// Encoder and motor control
volatile int cumulative_ticks = 0;
volatile bool revolution_complete = false;
const int encoder_target = 105;
const int pwmFreq = 1000;
const int pwmResolution = 8;

// Trigger control tracking
bool prevTriggerState = false;  // Stores last known trigger state

void IRAM_ATTR encoderISR() {
  cumulative_ticks++;
  if (cumulative_ticks >= encoder_target) {
    revolution_complete = true;
  }
}

// Convert percent to 8-bit duty cycle
```

```
int duty_u8(int percentage) {
  return int(percentage / 100.0 * ((1 << pwmResolution) - 1));
}

Adafruit_MLX90640 mlx;
float frame[TOTAL_PIXELS];

// ----- Controller (Joystick) Message Definition -----
#pragma pack(push, 1)
typedef struct __attribute__((packed)) {
  int joyX;
  int joyY;
  bool trigger;
  int ledState;
} ControllerMessage;
#pragma pack(pop)

ControllerMessage ctrlMsg = {};  // Global instance of ControllerMessage

// ----- Target Address for Sending Thermal Data -----
// In this configuration the turret sends its thermal data to the controller.
// (Replace with the actual MAC if needed.)
uint8_t controllerAddress[] = { 0xE8, 0x9F, 0x6D, 0x2F, 0x91, 0x60 };

// ----- Variables for ESP-NOW Send Status -----
volatile bool canSend = true, lastSendSuccessful = true;

// ----- Frame Chunk Definition -----
#pragma pack(push, 1)
typedef struct {
  uint8_t chunkIndex;
  uint8_t totalChunks;
  float chunkData[CHUNK_SIZE];
} FrameChunk;
#pragma pack(pop)

// ----- ESP-NOW Send Callback -----
// Called after a thermal frame chunk is sent.
void OnDataSent(const uint8_t *mac, esp_now_send_status_t status) {
  canSend = true;
  lastSendSuccessful = (status == ESP_NOW_SEND_SUCCESS);
}

// ----- ESP-NOW Receive Callback -----
// This callback is triggered when any ESP-NOW data is received.
// It now uses the corrected signature to match: (const esp_now_recv_info_t*, const uint8_t*, int).
void OnDataRecv(const esp_now_recv_info_t *info, const uint8_t *data, int len) {
  if (len == sizeof(ControllerMessage)) {
    memcpy(&ctrlMsg, data, sizeof(ControllerMessage));
    Serial.printf("Joystick Command Received - X: %d, Y: %d, Trigger: %d, LED: %d\n",
                  ctrlMsg.joyX, ctrlMsg.joyY, ctrlMsg.trigger, ctrlMsg.ledState);
  }
```

```
}

// ----- Function to Send Thermal Camera Frame Chunks -----
// The frame is split into chunks and each chunk is sent via ESP-NOW.
void sendFrameChunks() {
  for (uint8_t i = 0; i < TOTAL_CHUNKS; i++) {
    FrameChunk *chunk = (FrameChunk *)malloc(sizeof(FrameChunk));
    if (!chunk) return;

    chunk->chunkIndex = i;
    chunk->totalChunks = TOTAL_CHUNKS;
    uint16_t offset = i * CHUNK_SIZE;
    uint8_t actualSize = min(CHUNK_SIZE, TOTAL_PIXELS - offset);
    memcpy(chunk->chunkData, &frame[offset], actualSize * sizeof(float));

    canSend = false;
    esp_now_send(controllerAddress, (uint8_t *)chunk, sizeof(FrameChunk));

    // Wait until sending completes or timeout.
    uint32_t timeout = millis() + 100;
    while (!canSend && millis() < timeout) delay(1);

    free(chunk);
    delay(5);  // pacing between chunks
  }
}

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_STA);

  // Initialize ESP-NOW and register callbacks.
  if (esp_now_init() != ESP_OK) {
    Serial.println(" ESP-NOW Init Failed");
    return;
  }
  esp_now_register_send_cb(OnDataSent);
  esp_now_register_recv_cb(OnDataRecv);

  // Setup peer to transmit thermal data to the controller.
  esp_now_peer_info_t peerInfo = {};
  memcpy(peerInfo.peer_addr, controllerAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;
  if (esp_now_add_peer(&peerInfo) != ESP_OK) {
    Serial.println(" Failed to add controller peer");
    return;
  }

  // Initialize I2C (Wire) for MLX90640.
  Wire.begin(22, 20);  // Adjust pins as required
  Wire.setClock(400000);
  if (!mlx.begin(0x33, &Wire)) {
```

```
    Serial.println(" MLX90640 not found");
    while (1) delay(10);
  }
  mlx.setMode(MLX90640_CHESS);
  mlx.setRefreshRate(MLX90640_8_HZ);

  Serial.println(" Turret Ready");

  //Small Motor Spin
  pinMode(BIN_1, OUTPUT);
  pinMode(BIN_2, OUTPUT);
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(BIN_2, LOW);
  if (!ledcAttach(BIN_1, pwmFreq, pwmResolution)) {
    Serial.println("Error: ledcAttach failed to configure the LEDC pin!");
  }
  pinMode(ENCODER_A, INPUT);
  pinMode(ENCODER_B, INPUT);
}

void loop() {
  // Read a frame from the thermal camera.
  if (mlx.getFrame(frame) == 0) {
    sendFrameChunks();
  } else {
    Serial.println(" Frame read failed");
  }
  delay(150);  // Control the rate of frame transmission

  //Motor Spin
  // Detect rising edge of trigger (0 -> 1)
  if (ctrlMsg.trigger && !prevTriggerState) {
    Serial.println("Trigger received. Starting motor...");
    cumulative_ticks = 0;
    revolution_complete = false;
    digitalWrite(LED_PIN, LOW);
    attachInterrupt(digitalPinToInterrupt(ENCODER_A), encoderISR, RISING);
    ledcWrite(BIN_1, duty_u8(20));
  }
  if (revolution_complete) {
    ledcWrite(BIN_1, 0);
    detachInterrupt(digitalPinToInterrupt(ENCODER_A));
    digitalWrite(LED_PIN, HIGH);
    //Serial.print("Revolution complete! Total ticks: ");
    //Serial.println(cumulative_ticks);
  }

  prevTriggerState = ctrlMsg.trigger;
  delay(10);  // Small delay to prevent CPU hog
}
```