```
#include <WiFi.h>
#include <esp_now.h>
#include <esp_wifi.h>

constexpr uint8_t JOY_X = 2;
constexpr uint8_t JOY_Y = 3;

constexpr uint8_t ESP_NOW_CHANNEL = 1;
constexpr uint8_t PROTOCOL_VERSION = 0x10;

constexpr uint8_t CONTROL_MAX = 127;
constexpr uint8_t JOYSTICK_SAMPLES = 4;
constexpr uint8_t CALIBRATION_SAMPLES = 32;
constexpr uint8_t FILTER_DIVISOR = 4;
constexpr int JOYSTICK_DEADZONE = 70;

constexpr uint16_t ACTIVE_SEND_INTERVAL_MS = 25;
constexpr uint16_t IDLE_SEND_INTERVAL_MS = 100;
constexpr uint16_t SEND_TIMEOUT_MS = 100;

uint8_t receiverMAC[] = {0xB8, 0xF8, 0x62, 0x2D, 0x5B, 0x44}; //scheda 1: 44,
scheda 2: 45

struct __attribute__((packed)) ControlPacket {
 int8_t x;
 int8_t y;
 uint8_t seq;
};

static_assert(sizeof(ControlPacket) == 3, "Unexpected packet size");

ControlPacket desiredPacket = {};
ControlPacket lastQueuedPacket = {};

volatile bool sendInFlight = false;
volatile bool sendFailed = false;

uint32_t lastSendAttemptMs = 0;
int centerX = 2048;
int centerY = 2048;
int32_t filteredX = 2048;
int32_t filteredY = 2048;
uint8_t nextSequence = 0;

uint16_t readAveragedAnalog(uint8_t pin) {
```

```
 uint32_t total = 0;

 for (uint8_t i = 0; i < JOYSTICK_SAMPLES; ++i) {
   total += analogRead(pin);
 }

 return total / JOYSTICK_SAMPLES;
}

int calibrateCenter(uint8_t pin) {
 uint32_t total = 0;

 for (uint8_t i = 0; i < CALIBRATION_SAMPLES; ++i) {
   total += analogRead(pin);
   delay(2);
 }

 return total / CALIBRATION_SAMPLES;
}

int8_t scaleAxis(uint16_t rawValue, int32_t &filteredValue, int center) {
 filteredValue += (static_cast<int32_t>(rawValue) - filteredValue) /
FILTER_DIVISOR;

 const int centered = static_cast<int>(filteredValue) - center;
 const int magnitude = abs(centered);

 if (magnitude <= JOYSTICK_DEADZONE) {
   return 0;
 }

 const int availableRange = centered >= 0 ? (4095 - center) : center;
 const int usableRange = max(availableRange - JOYSTICK_DEADZONE, 1);
 int scaled = ((magnitude - JOYSTICK_DEADZONE) * CONTROL_MAX + usableRange / 2) /
usableRange;
 scaled = min(scaled, static_cast<int>(CONTROL_MAX));

 return centered >= 0 ? scaled : -scaled;
}

void onDataSent(const esp_now_send_info_t *txInfo, esp_now_send_status_t status) {
 (void)txInfo;
 sendInFlight = false;
 sendFailed = (status != ESP_NOW_SEND_SUCCESS);
}
```

```
void configureRadio() {
 WiFi.mode(WIFI_STA);
 WiFi.disconnect();

 if (esp_wifi_set_ps(WIFI_PS_NONE) != ESP_OK) {
   Serial.println("Errore power save Wi-Fi");
   while (true) {
     delay(1000);
   }
 }

 if (esp_wifi_set_channel(ESP_NOW_CHANNEL, WIFI_SECOND_CHAN_NONE) != ESP_OK) {
   Serial.println("Errore canale Wi-Fi");
   while (true) {
     delay(1000);
   }
 }

 if (esp_now_init() != ESP_OK) {
   Serial.println("Errore inizializzazione ESP-NOW");
   while (true) {
     delay(1000);
   }
 }

 esp_now_register_send_cb(onDataSent);

 esp_now_peer_info_t peerInfo = {};
 memcpy(peerInfo.peer_addr, receiverMAC, sizeof(receiverMAC));
 peerInfo.channel = ESP_NOW_CHANNEL;
 peerInfo.encrypt = false;
 peerInfo.ifidx = WIFI_IF_STA;

 if (esp_now_add_peer(&peerInfo) != ESP_OK) {
   Serial.println("Errore aggiunta peer");
   while (true) {
     delay(1000);
   }
 }
}

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

```
 analogReadResolution(12);
 analogSetPinAttenuation(JOY_X, ADC_11db);
 analogSetPinAttenuation(JOY_Y, ADC_11db);

 configureRadio();

 delay(50);
 centerX = calibrateCenter(JOY_X);
 centerY = calibrateCenter(JOY_Y);
 filteredX = centerX;
 filteredY = centerY;
}

void loop() {
 const uint32_t now = millis();

 if (sendInFlight && (now - lastSendAttemptMs > SEND_TIMEOUT_MS)) {
   sendInFlight = false;
   sendFailed = true;
 }

 desiredPacket.x = scaleAxis(readAveragedAnalog(JOY_X), filteredX, centerX);
 desiredPacket.y = scaleAxis(readAveragedAnalog(JOY_Y), filteredY, centerY);

 const bool moving = desiredPacket.x != 0 || desiredPacket.y != 0;
 const uint16_t targetInterval = (moving || sendFailed) ? ACTIVE_SEND_INTERVAL_MS :
IDLE_SEND_INTERVAL_MS;
 const bool intervalElapsed = (now - lastSendAttemptMs) >= targetInterval;

 if (!sendInFlight && intervalElapsed) {
   ControlPacket packetToSend = desiredPacket;
   packetToSend.seq = nextSequence;

   lastSendAttemptMs = now;

   const esp_err_t sendResult = esp_now_send(receiverMAC, reinterpret_cast<const
uint8_t *>(&packetToSend), sizeof(packetToSend));

   if (sendResult == ESP_OK) {
     sendInFlight = true;
     sendFailed = false;
     lastQueuedPacket = packetToSend;
     ++nextSequence;
   } else {
     sendFailed = true;
```

```
    }
  }

  delay(1);
}
```