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

constexpr uint8_t PWM_LEFT = 0;
constexpr uint8_t IN1 = 1;
constexpr uint8_t IN2 = 2;

constexpr uint8_t IN3 = 3;
constexpr uint8_t IN4 = 4;
constexpr uint8_t PWM_RIGHT = 5;

constexpr uint8_t LED_PIN = 6;
constexpr uint8_t CIRCUIT_PIN = 7;
constexpr uint8_t ESP_NOW_CHANNEL = 1;
constexpr uint8_t PROTOCOL_VERSION = 0x10;

constexpr int CONTROL_MAX = 127;
constexpr uint16_t FAILSAFE_MS = 300;
constexpr uint16_t PWM_FREQUENCY_HZ = 1200;
constexpr uint8_t PWM_RESOLUTION_BITS = 8;
constexpr int MOTOR_PWM_LIMIT = 220;
constexpr int MOTOR_MIN_EFFECTIVE_PWM = 85;
constexpr bool EXTERNAL_LED_ACTIVE_LOW = false;

// Pacchetto ricevuto dal joystick (codice 2) — senza flags
struct __attribute__((packed)) ControlPacket {
 int8_t x;
 int8_t y;
 uint8_t seq;
};

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

// Pacchetto inviato alla terza scheda con lo stato del circuito
struct __attribute__((packed)) StatusPacket {
 uint8_t flags;
 uint8_t seq;
};

static_assert(sizeof(StatusPacket) == 2, "Unexpected packet size");

constexpr uint8_t FLAG_CIRCUIT_CLOSED = 0x01;

// MAC della terza scheda — da aggiornare con il MAC reale
```

```
uint8_t thirdBoardMAC[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

constexpr uint16_t STATUS_SEND_INTERVAL_MS = 100;

portMUX_TYPE packetMux = portMUX_INITIALIZER_UNLOCKED;

volatile bool packetPending = false;
volatile uint32_t lastPacketTime = 0;
ControlPacket pendingPacket = {};

int lastLeftSpeed = 32767;
int lastRightSpeed = 32767;
int lastLedState = -1;
bool sequenceInitialized = false;
uint8_t lastSequence = 0;

uint8_t statusSequence = 0;
uint32_t lastStatusSendMs = 0;

bool isNewerSequence(uint8_t incoming, uint8_t previous) {
 const uint8_t delta = incoming - previous;
 return delta != 0 && delta < 128;
}

int shapeMotorCommand(int command) {
 if (command == 0) {
   return 0;
 }

 const int magnitude = abs(command);
 const int shapedMagnitude = MOTOR_MIN_EFFECTIVE_PWM +
                             ((magnitude - 1) * (MOTOR_PWM_LIMIT -
MOTOR_MIN_EFFECTIVE_PWM)) / (CONTROL_MAX - 1);

 return command > 0 ? shapedMagnitude : -shapedMagnitude;
}

void writeIndicator(bool circuitClosed) {
 const int externalLedState = (circuitClosed ^ EXTERNAL_LED_ACTIVE_LOW) ? HIGH :
LOW;
 if (externalLedState != lastLedState) {
   digitalWrite(LED_PIN, externalLedState);
   lastLedState = externalLedState;
 }
```

```
#if defined(RGB_BUILTIN)
 if (circuitClosed) {
   rgbLedWrite(RGB_BUILTIN, 0, 32, 0);
 } else {
   rgbLedWrite(RGB_BUILTIN, 0, 0, 0);
 }
#endif
}

void writeMotor(uint8_t pin1, uint8_t pin2, uint8_t pwmPin, int speed, int
&lastSpeed) {
 if (speed == lastSpeed) {
   return;
 }

 lastSpeed = speed;

 if (speed > 0) {
   digitalWrite(pin1, HIGH);
   digitalWrite(pin2, LOW);
 } else if (speed < 0) {
   digitalWrite(pin1, LOW);
   digitalWrite(pin2, HIGH);
 } else {
   digitalWrite(pin1, LOW);
   digitalWrite(pin2, LOW);
 }

 ledcWrite(pwmPin, abs(speed));
}

void applyOutputs(int8_t x, int8_t y, bool circuitClosed) {
 int leftCommand = static_cast<int>(y) + static_cast<int>(x);
 int rightCommand = static_cast<int>(y) - static_cast<int>(x);

 const int maxCommandMagnitude = max(abs(leftCommand), abs(rightCommand));
 if (maxCommandMagnitude > CONTROL_MAX) {
   leftCommand = (leftCommand * CONTROL_MAX) / maxCommandMagnitude;
   rightCommand = (rightCommand * CONTROL_MAX) / maxCommandMagnitude;
 }

 const int leftSpeed = shapeMotorCommand(leftCommand);
 const int rightSpeed = shapeMotorCommand(rightCommand);

 writeMotor(IN1, IN2, PWM_LEFT, leftSpeed, lastLeftSpeed);
```

```
 writeMotor(IN3, IN4, PWM_RIGHT, rightSpeed, lastRightSpeed);
 writeIndicator(circuitClosed);
}

void stopOutputs() {
 applyOutputs(0, 0, false);
}

void onReceive(const esp_now_recv_info_t *info, const uint8_t *incomingData, int
len) {
 (void)info;

 if (len != sizeof(ControlPacket)) {
   return;
 }

 ControlPacket packet = {};
 memcpy(&packet, incomingData, sizeof(packet));

 // Nessun campo flags: il controllo di versione è rimosso in accordo con il codice
2
 portENTER_CRITICAL(&packetMux);
 pendingPacket = packet;
 packetPending = true;
 lastPacketTime = millis();
 portEXIT_CRITICAL(&packetMux);
}

void sendCircuitStatus(bool circuitClosed) {
 StatusPacket statusPacket = {};
 statusPacket.flags = (PROTOCOL_VERSION & 0xF0) | (circuitClosed ?
FLAG_CIRCUIT_CLOSED : 0);
 statusPacket.seq = statusSequence++;

 esp_now_send(thirdBoardMAC, reinterpret_cast<const uint8_t *>(&statusPacket),
sizeof(statusPacket));
}

void setup() {
 pinMode(IN1, OUTPUT);
 pinMode(IN2, OUTPUT);
 pinMode(IN3, OUTPUT);
 pinMode(IN4, OUTPUT);
 pinMode(PWM_LEFT, OUTPUT);
 pinMode(PWM_RIGHT, OUTPUT);
```

```
 pinMode(LED_PIN, OUTPUT);
 pinMode(CIRCUIT_PIN, INPUT_PULLUP);

 uint8_t forcedMAC[] = {0xB8, 0xF8, 0x62, 0x2D, 0x5B, 0x44};

 WiFi.mode(WIFI_STA);
 WiFi.disconnect();

 if (esp_wifi_set_mac(WIFI_IF_STA, forcedMAC) != ESP_OK) {
   while (true) {
     delay(1000);
   }
 }

 if (esp_wifi_set_ps(WIFI_PS_NONE) != ESP_OK) {
   while (true) {
     delay(1000);
   }
 }

 if (esp_wifi_set_channel(ESP_NOW_CHANNEL, WIFI_SECOND_CHAN_NONE) != ESP_OK) {
   while (true) {
     delay(1000);
   }
 }

 if (!ledcAttach(PWM_LEFT, PWM_FREQUENCY_HZ, PWM_RESOLUTION_BITS) ||
!ledcAttach(PWM_RIGHT, PWM_FREQUENCY_HZ, PWM_RESOLUTION_BITS)) {
   while (true) {
     delay(1000);
   }
 }

 if (esp_now_init() != ESP_OK) {
   while (true) {
     delay(1000);
   }
 }

 esp_now_register_recv_cb(onReceive);

 esp_now_peer_info_t peerInfo = {};
 memcpy(peerInfo.peer_addr, thirdBoardMAC, sizeof(thirdBoardMAC));
 peerInfo.channel = ESP_NOW_CHANNEL;
 peerInfo.encrypt = false;
```

```
 peerInfo.ifidx = WIFI_IF_STA;

 esp_now_add_peer(&peerInfo);

 stopOutputs();
}

void loop() {
 ControlPacket packet = {};
 bool hasPacket = false;
 uint32_t packetAgeReference = 0;

 portENTER_CRITICAL(&packetMux);
 hasPacket = packetPending;
 if (hasPacket) {
   packet = pendingPacket;
   packetPending = false;
 }
 packetAgeReference = lastPacketTime;
 portEXIT_CRITICAL(&packetMux);

 // Lo stato del circuito viene letto localmente: LOW = circuito chiuso
(INPUT_PULLUP)
 const bool circuitClosed = (digitalRead(CIRCUIT_PIN) == LOW);

 if (hasPacket) {
   const bool acceptPacket = !sequenceInitialized || isNewerSequence(packet.seq,
lastSequence);

   if (acceptPacket) {
     sequenceInitialized = true;
     lastSequence = packet.seq;
     applyOutputs(packet.x, packet.y, circuitClosed);
   }
 }

 if (millis() - packetAgeReference > FAILSAFE_MS) {
   stopOutputs();
 }

 // Invio periodico dello stato del circuito alla terza scheda
 const uint32_t now = millis();
 if (now - lastStatusSendMs >= STATUS_SEND_INTERVAL_MS) {
   lastStatusSendMs = now;
   sendCircuitStatus(circuitClosed);
```

```
 }

 delay(1);
}
```