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

constexpr uint8_t PWM_LEFT = 4;
constexpr uint8_t IN1 = 5;
constexpr uint8_t IN2 = 0;

constexpr uint8_t IN3 = 1;
constexpr uint8_t IN4 = 3;
constexpr uint8_t PWM_RIGHT = 21;

constexpr uint8_t LED_PIN = 6;
constexpr uint8_t ESP_NOW_CHANNEL = 1;
constexpr uint8_t PROTOCOL_VERSION = 0x10;
constexpr uint8_t FLAG_CIRCUIT_CLOSED = 0x01;

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;

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

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

struct __attribute__((packed)) StatusPacket {
 uint8_t flags;
 uint8_t seq;
};

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

portMUX_TYPE controlMux = portMUX_INITIALIZER_UNLOCKED;
portMUX_TYPE statusMux = portMUX_INITIALIZER_UNLOCKED;

volatile bool controlPacketPending = false;
volatile uint32_t lastControlPacketTime = 0;
```

```
ControlPacket pendingControlPacket = {};

volatile bool statusPacketPending = false;
StatusPacket pendingStatusPacket = {};

int lastLeftSpeed = 32767;
int lastRightSpeed = 32767;
int lastLedState = -1;
bool controlSequenceInitialized = false;
uint8_t lastControlSequence = 0;
bool statusSequenceInitialized = false;
uint8_t lastStatusSequence = 0;

bool circuitClosed = false;

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 active) {
 const int externalLedState = (active ^ EXTERNAL_LED_ACTIVE_LOW) ? HIGH : LOW;
 if (externalLedState != lastLedState) {
   digitalWrite(LED_PIN, externalLedState);
   lastLedState = externalLedState;
 }

#if defined(RGB_BUILTIN)
 if (active) {
   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 active) {
 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(active);
}

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

   portENTER_CRITICAL(&controlMux);
   pendingControlPacket = packet;
   controlPacketPending = true;
   lastControlPacketTime = millis();
   portEXIT_CRITICAL(&controlMux);

 } else if (len == sizeof(StatusPacket)) {
   StatusPacket packet = {};
   memcpy(&packet, incomingData, sizeof(packet));

   if ((packet.flags & 0xF0) != PROTOCOL_VERSION) {
     return;
   }

   portENTER_CRITICAL(&statusMux);
   pendingStatusPacket = packet;
   statusPacketPending = true;
   portEXIT_CRITICAL(&statusMux);
 }
}

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);

 // MAC diverso dalla scheda 1 — da aggiornare anche nel trasmettitore joystick di
questa scheda
 uint8_t forcedMAC[] = {0xB8, 0xF8, 0x62, 0x2D, 0x5B, 0x45};
```

```
 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);
 stopOutputs();
}

void loop() {
 ControlPacket controlPacket = {};
 bool hasControlPacket = false;
 uint32_t packetAgeReference = 0;

 portENTER_CRITICAL(&controlMux);
 hasControlPacket = controlPacketPending;
```

```
 if (hasControlPacket) {
   controlPacket = pendingControlPacket;
   controlPacketPending = false;
 }
 packetAgeReference = lastControlPacketTime;
 portEXIT_CRITICAL(&controlMux);

 portENTER_CRITICAL(&statusMux);
 if (statusPacketPending) {
   const StatusPacket statusPacket = pendingStatusPacket;
   statusPacketPending = false;
   portEXIT_CRITICAL(&statusMux);

   const bool acceptStatus = !statusSequenceInitialized ||
isNewerSequence(statusPacket.seq, lastStatusSequence);
   if (acceptStatus) {
     statusSequenceInitialized = true;
     lastStatusSequence = statusPacket.seq;
     circuitClosed = (statusPacket.flags & FLAG_CIRCUIT_CLOSED) != 0;
   }
 } else {
   portEXIT_CRITICAL(&statusMux);
 }

 if (hasControlPacket) {
   const bool acceptPacket = !controlSequenceInitialized ||
isNewerSequence(controlPacket.seq, lastControlSequence);

   if (acceptPacket) {
     controlSequenceInitialized = true;
     lastControlSequence = controlPacket.seq;
     applyOutputs(controlPacket.x, controlPacket.y, circuitClosed);
   }
 }

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

 delay(1);
}
```