AGH University of Krakow

Design Lab

Project of a drone controlled using a mobile application

Report prepared by:

Michał Ciągała

Paweł Michalcewicz

Huber Pałka

Supervisor: **dr Inż. Jacek Stępień**

Contents

1. Introduction

The topic of this project, carried out during the Design Lab course, was a drone built using the SpeedyBee F405 flight controller and ESP32-WROOM microcontroller for wireless communication with a mobile application for the Android system. The primary objective was to learn systematic group work on a single task. The project included configuring the SpeedyBee platform using its dedicated application, programming the network controller from scratch in C++ using Arduino IDE, and developing a mobile application for Android in Kotlin using Android Studio. Additionally, the project required assembling all components and placing them on the drone's frame.

The complete project documentation is available in the GitHub repository: https://github.com/Ciagal/DesignLab---Ciagala-Michalcewicz-Palka.

2. Description of Technologies and Tools

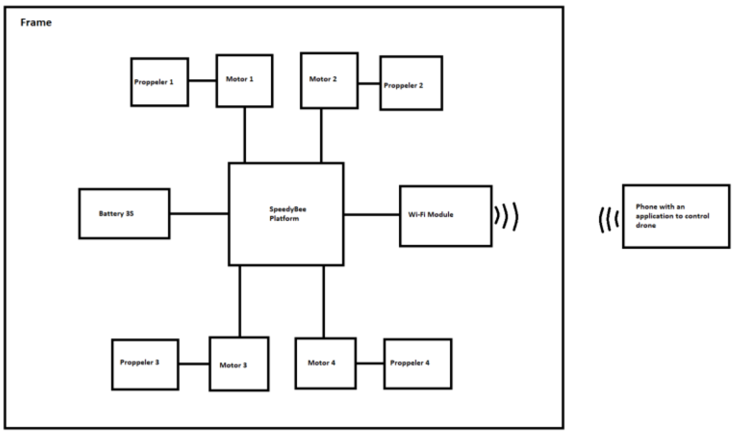
The drone project primarily relied on the SpeedyBee F405 controller, a platform commonly used in racing drone development. Its built-in flight control and motor control functions provide significant advantages, addressing two of the most challenging aspects of such projects. Typically, SpeedyBee is paired with an RC receiver for control via a dedicated transmitter. However, in this project, Wi-Fi communication via the ESP32-WROOM microcontroller and a UART protocol connecting the two microcontrollers was used instead. This approach enabled control of the drone using a custom-developed mobile application.

To configure the SpeedyBee controller, the BetaFlight flight controller firmware was installed and configured via a dedicated app to suit the project’s needs. For the Wi-Fi receiver, the ESP32-WROOM module was programmed in C++ using Arduino IDE. The code utilized the WiFi.h and WiFiServer.h libraries for Wi-Fi communication and HardwareSerial.h for UART-based data transmission.

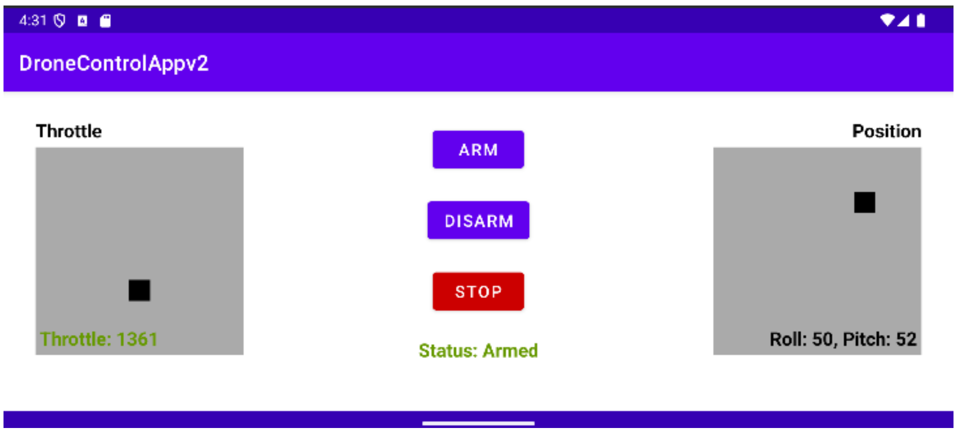
The mobile application was developed using Android Studio due to its popularity among Android app developers, ranging from beginners to advanced. Android Studio provides templates that ensure compatibility with Android devices, allowing developers to focus on defining app functionality and design.

Mechanical components were also essential for the project. Flash Hobby BE1806 2300kV brushless motors were chosen for their appropriate parameters and availability. The motors were paired with 5.1-inch Nepal N1 propellers. A Dualsky 1800mAh 25C 11.1V Li-Poly battery was selected based on the voltage requirements of the motors and controllers. All components were mounted on the QAV250 frame, chosen for its wide availability and low cost. Detailed technical specifications for all components can be found in the repository.

3. Electric Scheme



4. Application layout



Left analog pad is responsible for throttle input and right throttle pad is responsible for the movement of the drone such as pitch, yaw and angle. Arm button is for arming the drone, without it, inputs of both analog pads are turned off. Disarm button is used for disarming the drone and lastly the stop button is used for turning off the drone in case of emergency.

5. Work Methodology

The project began with selecting and documenting components, followed by ordering them. During the waiting period, a concept for the mobile application was developed. After receiving the components, they were assembled and soldered onto the drone frame, and preliminary configuration of the SpeedyBee controller was carried out.

Once the physical assembly was complete, software development began. Each team member was assigned one of four key tasks:

- Configuring the SpeedyBee controller
- Flight Controller configuration
- Programming the Wi-Fi receiver

- Programming the mobile application

4. Implementation Description

4.1. Assembly

After collecting all the components, assembly of the drone frame began, including measuring the correct length for the brushless motor wires, after adjusting them to the main structure. The wires provided by the manufacturer were significantly too long, so it was decided to shorten them for practical and aesthetic reasons. Next, the brushless motor wires were soldered to the ESC board, and the power cable connecting the battery to the system was attached. An electrolytic capacitor was also soldered in the same locations, which improves the overall stability of the system.

4.2. SpeedyBee Controller Configuration

Using the tool available on the website https://esc-configurator.com, the configuration started with motor tests. After successfully completing the tests, it was decided to change the ESC firmware from the default BLHeli S to Bluejay for two reasons. The first reason was the ability to use the melody editor feature, which allows a selected melody to play when powering up, a feature not available in BLHeli S. The second reason was the need to use the DShot protocol instead of the standard PWM. Appropriate changes were also made in the SpeedyBee configuration app settings. After completing these steps, the ESC configuration was successfully completed.

4.3. Flight Controller Configuration

Setup of the flight controller was conducted through a dedicated BetaFlight app. First, the controller was calibrated using a built in system. After that was done, the correct port was selected in the Ports section. Also, in the same section the data protocol was selected to be SBUS. Next in modes section, data range for both arming and angle control was implemented. Arming range allows the drone to be armed recieving a specific signal. Angle control allows the drone to return to level after user stops puting in any inputs.

4.4. Wi-Fi Receiver Programming

The Wi-Fi receiver for the drone project was implemented using the ESP32 microcontroller, programmed with Arduino IDE. The receiver operates in Access Point (AP) mode, creating a network named "DronESP" with the password "12345678". The ESP32 also hosts two servers: an HTTP server on port 80 and a RAW server on port 81. The RAW server allows real-time communication via tools like PuTTY.

5.4.1 Communication with the Flight Controller

The ESP32 communicates with the drone's flight controller using the SBUS protocol via the UART2 interface. The TX pin of UART2 is assigned to GPIO pin 17, while the RX pin is not used. The UART2 is configured with a baud rate of 100000 and SERIAL_8E2 mode, as demonstrated below:

```
SerialDrone.begin(100000, SERIAL_8E2, SBUS_RX_PIN, SBUS_TX_PIN, true);
```

5.4.2 SBUS Frame Creation

A key function of the receiver is to translate channel values from the 1000–2000 range to the SBUS protocol range of 880–2112. This is accomplished using the mapToSBUS() function:

```
uint16_t mapToSBUS(uint16_t value) {
    return map(value, 1000, 2000, 880, 2112);
}
```

The createSBUSFrame() function assembles an SBUS frame containing the channel values and flags for failsafe or frame loss, as shown here:

```
void createSBUSFrame(const uint16_t* ch, bool failsafe, bool frameLost) {
    memset(sbusFrame, 0, SBUS_FRAME_SIZE);
    sbusFrame[0] = 0x0F; // start byte

    uint8_t bitIndex = 0;
    for (int i = 0; i < 16; i++) {
        uint16_t sbusValue = mapToSBUS(ch[i]);
        for (int bit = 0; bit < 11; bit++) {
            if (sbusValue & (1 << bit)) {
                sbusFrame[1 + (bitIndex / 8)] |= (1 << (bitIndex % 8));
            }
            bitIndex++;
        }
    }

    if (failsafe)  sbusFrame[23] |= (1 << 3);
    if (frameLost) sbusFrame[23] |= (1 << 2);

    sbusFrame[24] = 0x00;
}
```

5.4.3 Command Handling

The receiver processes client commands sent in the format "CHx=yyyy", where x represents the channel number, and yyyy is the desired value. The handleClientCommand() function parses these commands and updates the channel values:

```
void handleClientCommand(const char* command, uint16_t* chArray) {
    if (strncmp(command, "CH", 2) == 0) {
        const char* eq = strchr(command, '=');
        if (eq) {
            int channel = atoi(command + 2);
            int value   = atoi(eq + 1);
            if (channel >= 1 && channel <= 16) {
                chArray[channel - 1] = constrain(value, 1000, 2000);
            }
        }
    }
}
```

5.4.4 HTTP and RAW Communication

The HTTP server (port 80) handles commands by receiving requests, parsing the commands, and sending SBUS frames to the flight controller. The handleHTTP() function manages these interactions:

```
void handleHTTP(WiFiClient client) {
    // Receive HTTP request
    String fullRequest;
    unsigned long startTime = millis();
    while (millis() - startTime < 1000) {
        while (client.available()) {
            char c = client.read();
            fullRequest += c;
            startTime = millis();
        }
        delay(5);
    }

    // Process commands
    int startIndex = 0;
    while (true) {
        int newLinePos = fullRequest.indexOf('\n', startIndex);
        if (newLinePos == -1) {
            break;
        }
        String line = fullRequest.substring(startIndex, newLinePos);
        line.trim();
        startIndex = newLinePos + 1;

        char cmdBuff[128];
        line.toCharArray(cmdBuff, sizeof(cmdBuff));
        handleClientCommand(cmdBuff, channels);
    }

    // Respond to the client
    client.println("HTTP/1.1 200 OK");
    client.println("Content-Type: text/plain");
    client.println("Connection: close");
    client.println();
    client.println("OK: command parsed");
    client.flush();
    client.stop();
}
```

The RAW server (port 81) allows real-time command input and response, suitable for debugging. It maintains a session where commands can be sent and acknowledged continuously.

5.4.5 Conclusion

This implementation supports one-way client-to-drone communication, as two-way communication was not required. The use of dual servers provides flexibility, allowing integration with mobile apps (HTTP) and real-time debugging (RAW). The system reliably constructs and transmits SBUS frames at ~14 ms intervals, ensuring seamless communication with the flight controller.

4.5. Mobile Application Programming

The mobile application for controlling the drone was developed using an Android-based template. Its primary purpose is to send SBUS channel values to the ESP32 receiver over a Wi-Fi connection. The app interface contains two virtual controls: a joystick for roll, pitch, and yaw adjustments, and a throttle control for managing the drone's vertical movement. Additionally, there is an arming button to enable or disable the controls.

4.5.1 Joystick Control:

- Responsible for controlling the roll and pitch of the drone.
- User input is processed in real time, with SBUS-compatible channel values being sent to the ESP32 receiver. The following code snippet highlights how the joystick updates roll and pitch:

```
rollValue = ((newX / maxX) * 200 - 100).toInt()

pitchValue = -((newY / maxY) * 200 - 100).toInt()


tvJoystickValues.text = "Roll: $rollValue, Pitch: $pitchValue"

sendAllChannels()
```

4.5.2 Throttle Control:

- Allows the user to control the vertical ascent and descent of the drone.
- The throttle value ranges between 1000 (minimum) and 2000 (maximum), converted based on touch input, as shown below:

```
throttleValue = (1000 +

                ((throttleControl.height - newY) / throttleControl.height) * 1000).toInt()

throttleValue = throttleValue.coerceIn(1000, 2000)



tvThrottle.text = "Throttle: $throttleValue"

sendAllChannels()
```

4.5.3 Arming and Disarming:

- The arming button ensures that the drone cannot be controlled unless armed. On arming, the throttle is set to a safe minimum, and the arm channel value is updated to 2000:

```
btnArm.setOnClickListener {

    isArmed = true
    armChannelValue = 2000

    throttleValue = 1000

    tvArmDisarmStatus.text = "Status: Armed"

tvArmDisarmStatus.setTextColor(getColor(android.R.color.holo_gr
een_dark))

    sendAllChannels()

}
```

- Disarming resets all values to their safe defaults, preventing unintentional movement:

```
btnDisarm.setOnClickListener {

    isArmed = false

    armChannelValue = 1000

    throttleValue = 1000

    rollValue = 0

    pitchValue = 0

    yawValue = 1500

    tvArmDisarmStatus.text = "Status: Disarmed"

tvArmDisarmStatus.setTextColor(getColor(android.R.color.holo_re
d_dark))

    sendAllChannels()

}
```

4.5.4 Communication Protocol

The application communicates with the ESP32 receiver using the HTTP protocol. Commands are sent in the SBUS format as plain text to the ESP32's HTTP server at **http://192.168.4.1/**. The key function for this task is sendAllChannels(), which composes and sends the channel data as follows:

```
val command = """
```

```
    CH1=$rollSbus

    CH2=$pitchSbus

    CH3=$throttleValue

    CH4=$yawValue

    CH5=$armChannelValue

""".trimIndent() + "\n"

sendCommandToESP32(command)
```

The sendCommandToESP32() method handles the network request using the OkHttp library. Errors and responses are displayed to the user:

```
private fun sendCommandToESP32(command: String) {

    val url = "http://192.168.4.1/"

    val requestBody = command.toRequestBody()

    val request = Request.Builder()

        .url(url)

        .post(requestBody)

        .build()


    client.newCall(request).enqueue(object : Callback {

        override fun onFailure(call: Call, e: IOException) {

            runOnUiThread {

                Toast.makeText(

                    this@MainActivity,

                    "Error: ${e.message}",

                    Toast.LENGTH_SHORT

                ).show()
            }

        }


        override fun onResponse(call: Call, response: Response)
{

            val responseBody = response.body?.string() ?: ""

            runOnUiThread {

                Toast.makeText(

                    this@MainActivity,

                    "Command
Sent:\n$command\nResponse:\n$responseBody",

                    Toast.LENGTH_SHORT

                ).show()

            }

        }

    })

}
```

4.5.5 User Requirements

The app requires users to connect their mobile devices to the ESP32's Access Point (SSID: "DronESP", Password: "12345678"). Without this connection, commands cannot be sent, and control will be unavailable.

4.5.6 Conclusion

The mobile application provides a user-friendly interface for controlling the drone. Its robust implementation ensures secure and responsive communication with the ESP32 receiver, allowing precise control of the drone's movement and operational states.

5. Results

Unfortunately, despite extensive efforts, the project's primary goal was not achieved. Tests using a standard RC receiver confirmed that the issue was software-related, as the drone successfully took off using a conventional setup.

The primary challenge was the lack of documentation for the SpeedyBee F405 controller. While the BetaFlight app correctly interpreted the data, the controller itself could not process it, preventing the drone from arming. Attempts to find a solution on forums were unsuccessful, as using SpeedyBee with a Wi-Fi receiver is a niche approach. Various adjustments, including modifying frame formats and transmission intervals, did not resolve the issue.

However, other objectives were met, including configuring the flight controller and assembling the drone's mechanical structure. The mobile application also performed as expected, although comprehensive testing was limited due to receiver issues.

6. Conclusion

The primary educational objective of the Design Lab course—to practice systematic group work—was achieved. Each team member fulfilled their responsibilities on time and supported others in solving project challenges.

Although the ultimate goal was not met, the project succeeded as an educational exercise. Participants gained valuable experience in teamwork, microcontroller programming, and electronics.

7. Application code

ESP32-WROOM Wi-Fi reciever code:

```
/***************************************************************

 * ESP32 Drone SBUS Controller (Dual Server)

 *

 *  - AP "DronESP" (pw "12345678").

 *  - serverHTTP (port 80): stary model HTTP (dla aplikacji).

 *  - serverRaw  (port 81): ciągła sesja RAW (dla PuTTY).

 *

 *  - SBUS -> FC (TX=17, RX=-1).

 ***************************************************************/


#include <WiFi.h>

#include <WiFiServer.h>

#include <HardwareSerial.h>
```

```
#include <string.h>
#include <stdlib.h>

// --- Ustawienia Wi-Fi ---
const char* ssid     = "DronESP";
const char* password = "12345678";

// Definiujemy DWA serwery globalnie:
// 1) serverHTTP (port 80) - Twój stary serwer HTTP
// 2) serverRaw  (port 81) - nowy serwer do testów PuTTY (RAW)
WiFiServer serverHTTP(80);
WiFiServer serverRaw(81);

// --- UART & SBUS ---
#define SBUS_RX_PIN -1
#define SBUS_TX_PIN 17

HardwareSerial SerialDrone(2);  // UART2 na ESP32

#define SBUS_FRAME_SIZE 25
uint8_t sbusFrame[SBUS_FRAME_SIZE];

// Tablica 16 kanałów [1000..2000]
uint16_t channels[16] = {
  1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000,
  1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000
};

// ------------------------------------------------------------
// map [1000..2000] -> SBUS [880..2112]
uint16_t mapToSBUS(uint16_t value) {
  return map(value, 1000, 2000, 880, 2112);
```

```
}

// -------------------------------------------------------------
// Buduje ramkę SBUS (16 kanałów)
void createSBUSFrame(const uint16_t* ch, bool failsafe, bool frameLost) {
  memset(sbusFrame, 0, SBUS_FRAME_SIZE);
  sbusFrame[0] = 0x0F; // start byte

  uint8_t bitIndex = 0;
  for (int i = 0; i < 16; i++) {
    uint16_t sbusValue = mapToSBUS(ch[i]);
    for (int bit = 0; bit < 11; bit++) {
      if (sbusValue & (1 << bit)) {
        sbusFrame[1 + (bitIndex / 8)] |= (1 << (bitIndex % 8));
      }
      bitIndex++;
    }
  }

  // Byte 23: flags
  if (failsafe)  sbusFrame[23] |= (1 << 3);
  if (frameLost) sbusFrame[23] |= (1 << 2);

  sbusFrame[24] = 0x00;
}

// -------------------------------------------------------------
// Przetwarzanie komendy "CHx=yyyy"
void handleClientCommand(const char* command, uint16_t* chArray) {
  if (strncmp(command, "CH", 2) == 0) {
    const char* eq = strchr(command, '=');
    if (eq) {
      int channel = atoi(command + 2);
      int value   = atoi(eq + 1);
      if (channel >= 1 && channel <= 16) {
        chArray[channel - 1] = constrain(value, 1000, 2000);
      }
    }
  }
}

// ---------------------------------------------------------------
// Obsługa Pojedynczego Żądania HTTP (Port 80)
void handleHTTP(WiFiClient client) {
  // Odczyt żądania
  String fullRequest;
  unsigned long startTime = millis();
  while (millis() - startTime < 1000) {
    while (client.available()) {
      char c = client.read();
      fullRequest += c;
      startTime = millis();
    }
    delay(5);
  }

  Serial.println("HTTP Request:\n-----------");
  Serial.println(fullRequest);
  Serial.println("-----------");

  // Rozbij na linie
  int startIndex = 0;
  while (true) {
    int newLinePos = fullRequest.indexOf('\n', startIndex);
    if (newLinePos == -1) {
      break;
    }
    String line = fullRequest.substring(startIndex, newLinePos);
    line.trim();
    startIndex = newLinePos + 1;

    // Konwersja na C-string
    char cmdBuff[128];
    line.toCharArray(cmdBuff, sizeof(cmdBuff));

    handleClientCommand(cmdBuff, channels);
  }

  // Zbuduj ramkę SBUS
  createSBUSFrame(channels, false, false);
  SerialDrone.write(sbusFrame, SBUS_FRAME_SIZE);

  // Debug
  Serial.print("SBUS Frame: ");
  for (int i = 0; i < SBUS_FRAME_SIZE; i++) {
    Serial.printf("%02X ", sbusFrame[i]);
  }
  Serial.println();

  // Odpowiedź HTTP i rozłączenie
  client.println("HTTP/1.1 200 OK");
  client.println("Content-Type: text/plain");
  client.println("Connection: close");
  client.println();
  client.println("OK: command parsed");
  client.flush();
  client.stop();
  Serial.println("HTTP client disconnected.");
}

// ---------------------------------------------------------------
// Obsługa Wieloliniowej Sesji (Port 81) - do PuTTY Raw
void handleRaw(WiFiClient client) {
  Serial.println("Raw client connected (port 81).");
  client.println("Welcome to SBUS Raw console! Type 'CH3=1500' etc.\n");

  while (client.connected()) {
    // Odczyt
    if (client.available()) {
      String line = client.readStringUntil('\n');
      line.trim();
      Serial.printf("[RAW] line: %s\n", line.c_str());

      // Obsługa "CHx=yyyy"
      handleClientCommand(line.c_str(), channels);

      // Zbuduj ramkę SBUS
      createSBUSFrame(channels, false, false);
      SerialDrone.write(sbusFrame, SBUS_FRAME_SIZE);

      // Odpowiedz w sesji
      client.println("OK: command parsed");
    }

    // co ~14 ms wysyłamy SBUS (podczas aktywnej sesji)
    static uint32_t lastRawSend = 0;
    if (millis() - lastRawSend > 14) {
      createSBUSFrame(channels, false, false);
      SerialDrone.write(sbusFrame, SBUS_FRAME_SIZE);
      lastRawSend = millis();
    }

    delay(1);
  }

  client.stop();
  Serial.println("Raw client disconnected (port 81).");
}

// ---------------------------------------------------------------
// Setup
// ---------------------------------------------------------------
void setup() {
  Serial.begin(115200);
  Serial.println("ESP32 SBUS Controller - Start (Dual server)");

  // UART2 for SBUS
  //  Param 'true' lub 'false' zależnie od inwersji w FC
  SerialDrone.begin(100000, SERIAL_8E2, SBUS_RX_PIN, SBUS_TX_PIN, true);

  // Wi-Fi AP
  WiFi.softAP(ssid, password);
  Serial.print("Access Point IP: ");
  Serial.println(WiFi.softAPIP());

  // Start server HTTP (80)
  serverHTTP.begin();
  Serial.println("HTTP server started on port 80.");
  // Start server RAW (81)
  serverRaw.begin();
  Serial.println("Raw server started on port 81.");
}

// ---------------------------------------------------------------
// Main loop
// ---------------------------------------------------------------
void loop() {
  // 1) Obsługa HTTP (port 80)
  WiFiClient clientHTTP = serverHTTP.available();
  if (clientHTTP) {
    handleHTTP(clientHTTP);
  }

  // 2) Obsługa RAW (port 81)
  WiFiClient clientR = serverRaw.available();
  if (clientR) {
    handleRaw(clientR);
  }

  // 3) Gdy brak klientów, wciąż wysyłamy SBUS co ~14ms
  static uint32_t lastSendTime2 = 0;
  if (millis() - lastSendTime2 > 14) {
    createSBUSFrame(channels, false, false);
    SerialDrone.write(sbusFrame, SBUS_FRAME_SIZE);
    lastSendTime2 = millis();
  }
}
```

MOBILE APPLICATION CODE

```
package com.example.dronecontrolappv2


import android.os.Bundle

import android.view.MotionEvent

import android.view.View

import android.widget.Button

import android.widget.RelativeLayout

import android.widget.TextView

import android.widget.Toast

import androidx.appcompat.app.AppCompatActivity

import okhttp3.*

import okhttp3.RequestBody.Companion.toRequestBody

import java.io.IOException


class MainActivity : AppCompatActivity() {
    private val client = OkHttpClient()

    private var isArmed = false

    // SBUS kanały
    private var rollValue = 0
    private var pitchValue = 0
    private var throttleValue = 1000  // Domyślnie minimalna przepustnica =
1000
    private var yawValue = 1500
    private var armChannelValue = 1000

    // Throttle – pointer
    private var throttlePointerId = -1
    private var initialY = 0f

    // Joystick – pointer
    private var joystickPointerId = -1
    private var initialXJoystick = 0f
    private var initialYJoystick = 0f

    private var lastSentTime = 0L

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val btnArm: Button = findViewById(R.id.btnArm)
        val btnDisarm: Button = findViewById(R.id.btnDisarm)
        val btnStop: Button = findViewById(R.id.btnStop)
        val            throttleControl:           RelativeLayout           =
findViewById(R.id.throttleControl)
        val centerDot: View = findViewById(R.id.centerDot)

        val            joystickControl:           RelativeLayout           =
findViewById(R.id.joystickControl)
        val joystickDot: View = findViewById(R.id.joystickDot)

        val tvThrottle: TextView = findViewById(R.id.tvGasValue)
        val tvJoystickValues: TextView = findViewById(R.id.tvJoystickValues)
        val             tvArmDisarmStatus:             TextView            =
findViewById(R.id.tvArmDisarmStatus)

        // ---------------------------------
        // Obsługa przycisków Arm / Disarm / Stop
        // ---------------------------------

        // ARMWANIE
        btnArm.setOnClickListener {
            isArmed = true
            armChannelValue = 2000
            // Ustawiamy minimalny gaz:
            throttleValue = 1000
            tvArmDisarmStatus.text = "Status: Armed"

tvArmDisarmStatus.setTextColor(getColor(android.R.color.holo_green_dark))
            sendAllChannels()
        }

        // ROZBROJENIE
        btnDisarm.setOnClickListener {
            isArmed = false
            armChannelValue = 1000
            throttleValue = 1000
            rollValue = 0
            pitchValue = 0
            yawValue = 1500
            tvArmDisarmStatus.text = "Status: Disarmed"

tvArmDisarmStatus.setTextColor(getColor(android.R.color.holo_red_dark))
            sendAllChannels()
        }

        // STOP
        btnStop.setOnClickListener {
            isArmed = false
            armChannelValue = 1000
            throttleValue = 1000
            rollValue = 0
            pitchValue = 0
            yawValue = 1500
            sendAllChannels()
            Toast.makeText(this,             "Emergency             STOP!",
Toast.LENGTH_SHORT).show()
        }

        // ----------------------------
        // THROTTLE – onTouch
        // ----------------------------
        centerDot.setOnTouchListener { v, event ->
            if (!isArmed) return@setOnTouchListener true

            v.parent.requestDisallowInterceptTouchEvent(true)

            when (event.actionMasked) {
                MotionEvent.ACTION_DOWN -> {
                    throttlePointerId =
event.getPointerId(event.actionIndex)
                    initialY = event.getY(event.actionIndex) - v.y
                }
                MotionEvent.ACTION_POINTER_DOWN -> {
                    if (throttlePointerId == -1) {
                        throttlePointerId =
event.getPointerId(event.actionIndex)
                        initialY = event.getY(event.actionIndex) - v.y
                    }
                }
                MotionEvent.ACTION_MOVE -> {
                    val pointerIndex =
event.findPointerIndex(throttlePointerId)
                    if (pointerIndex != -1 && throttlePointerId != -1) {
                        val newY = (event.getY(pointerIndex) - initialY)
                            .coerceIn(0f, throttleControl.height -
v.height.toFloat())

                        v.y = newY
                        // Mapa [0..(height-dot)] => [1000..2000]
                        throttleValue = (1000 +
                                ((throttleControl.height - newY) /
throttleControl.height) * 1000).toInt()

                        // W razie potrzeby daj tutaj coerceIn(990, 2000),
żeby mieć zapas
                        throttleValue = throttleValue.coerceIn(1000, 2000)

                        tvThrottle.text = "Throttle: $throttleValue"
                        sendAllChannels()
                    }
                }
                MotionEvent.ACTION_UP, MotionEvent.ACTION_POINTER_UP -> {
                    val upPointerId = event.getPointerId(event.actionIndex)
                    if (upPointerId == throttlePointerId) {
                        throttlePointerId = -1
                    }
                }
            }
            true
        }

        // ----------------------------
        // JOYSTICK (Roll/Pitch) - onTouch
        // ----------------------------
        joystickDot.setOnTouchListener { v, event ->
            if (!isArmed) return@setOnTouchListener true

            v.parent.requestDisallowInterceptTouchEvent(true)

            when (event.actionMasked) {
                MotionEvent.ACTION_DOWN -> {
                    joystickPointerId                                     =
event.getPointerId(event.actionIndex)
                    initialXJoystick = event.getX(event.actionIndex) - v.x
                    initialYJoystick = event.getY(event.actionIndex) - v.y
                }
                MotionEvent.ACTION_POINTER_DOWN -> {
                    if (joystickPointerId == -1) {
                        joystickPointerId                                 =
event.getPointerId(event.actionIndex)
                        initialXJoystick = event.getX(event.actionIndex) -
v.x
                        initialYJoystick = event.getY(event.actionIndex) -
v.y
                    }
                }
                MotionEvent.ACTION_MOVE -> {
                    val                    pointerIndex                    =
event.findPointerIndex(joystickPointerId)
                    if (pointerIndex != -1 && joystickPointerId != -1) {
                        val maxX = joystickControl.width - v.width
                        val maxY = joystickControl.height - v.height

                        val    newX    =    (event.getX(pointerIndex)    -
initialXJoystick)
                            .coerceIn(0f, maxX.toFloat())
                        val    newY    =    (event.getY(pointerIndex)    -
initialYJoystick)
                            .coerceIn(0f, maxY.toFloat())

                        v.x = newX
                        v.y = newY

                        rollValue = ((newX / maxX) * 200 - 100).toInt()
                        pitchValue = -((newY / maxY) * 200 - 100).toInt()

                        tvJoystickValues.text  =  "Roll:  $rollValue,  Pitch:
$pitchValue"
                        sendAllChannels()
                    }
                }
                MotionEvent.ACTION_UP, MotionEvent.ACTION_POINTER_UP -> {
                    val upPointerId = event.getPointerId(event.actionIndex)
                    if (upPointerId == joystickPointerId) {
                        joystickPointerId = -1
                    }
                }
            }
```

```
        true
    }
}

private fun sendAllChannels() {
    // Debounce co 100 ms
    if (System.currentTimeMillis() - lastSentTime > 100) {
        val rollSbus = 1500 + rollValue * 5
        val pitchSbus = 1500 + pitchValue * 5

        val command = """
            CH1=$rollSbus
            CH2=$pitchSbus
            CH3=$throttleValue
            CH4=$yawValue
            CH5=$armChannelValue
        """.trimIndent() + "\n"

        sendCommandToESP32(command)
        lastSentTime = System.currentTimeMillis()
    }
}

private fun sendCommandToESP32(command: String) {
    val url = "http://192.168.4.1/"
    val requestBody = command.toRequestBody()
    val request = Request.Builder()
        .url(url)
        .post(requestBody)
        .build()

    client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException) {
                runOnUiThread {
                    Toast.makeText(
                        this@MainActivity,
                        "Error: ${e.message}",
                        Toast.LENGTH_SHORT
                    ).show()
                }
            }

            override fun onResponse(call: Call, response: Response) {
                val responseBody = response.body?.string() ?: ""
                runOnUiThread {
                    Toast.makeText(
                        this@MainActivity,
                        "Command
Sent:\n$command\nResponse:\n$responseBody",
                        Toast.LENGTH_SHORT
                    ).show()
                }
            }
        })
    }
}
```