📘 Analog Inputs & ESP-to-ESP MQTT

File Structure, Hardware Connections, Code Logic & Thonny Instructions

📁 Project File Structure (MicroPython)

Your project is organized by **device responsibility**, which is the **correct and professional approach** in embedded systems.

```
MICROPYTHON
│
├── ESP32 – MQTT Subscriber +Display
│   └──main.py
│
├── ESP8266 – MQTT Publisher
│   └──main.py
│
├── task1_temp.py
├── task2_joystick_raw.py
├── task3_joystick_map.py
├── task4_uart.py
├── task5_json_uart.py
├── task6_temp_units.py
```

This structure reflects **how the system is architected**, not just how the code is written.

🔹 Core Rule in MicroPython (VERY IMPORTANT)

> Any file named main.py is executed automatically when the board boots.

This explains the structure:

- **ESP32** → must use `main.py`
- **ESP8266** → must use `main.py`
- **Raspberry Pi Pico** → does **not** use `main.py` for this lab

This is **intentional**, not accidental.

🟦 ESP32 — MQTT Subscriber + Display

Folder

```
ESP32 – MQTT Subscriber +Display/
└──main.py
```

Responsibility

- Connect to Wi-Fi
- Subscribe to MQTT topic
- Initialize LCD via SPI
- Receive messages
- Display received data

Why `main.py`

- ESP32 must start **listening immediately after power-on**
- No manual execution required
- Mimics real embedded firmware behavior

Oral explanation (ready to say)

> "The ESP32 is a permanent MQTT subscriber. Using main.py ensures the device behaves like real firmware and starts automatically after boot."

🟦 ESP8266 — MQTT Publisher

Folder

```
ESP8266 – MQTT Publisher/
└──main.py
```

Responsibility

- Connect to Wi-Fi
- Create MQTT client
- Publish messages periodically or on button press

Why `main.py`

- Publisher should start sending data **as soon as it is powered**
- No interaction required
- Matches IoT publishing model

Oral explanation

> "The ESP8266 acts as a data source. Using main.py allows it to publish messages automatically after boot."

🟩 Raspberry Pi Pico — Task Scripts (PART A)

Files

```
task1_temp.py
task2_joystick_raw.py
task3_joystick_map.py
task4_uart.py
task5_json_uart.py
task6_temp_units.py
```

Why these are NOT `main.py`

- Each task is **evaluated independently**

- Instructor expects **step-by-step execution**
- Only one experiment runs at a time
- Manual execution is required

This **exactly matches the PDF structure**.

Oral explanation

> "The Pico scripts are not firmware but experiments, so they are executed manually rather than automatically."

🧑‍💻 Thonny — How to Work (IMPORTANT)

For **every task**:

1. Connect the board via USB
2. Open **Thonny**
3. Go to

   ```
   Tools →Options →Interpreter
   ```

4. Select:
   - **MicroPython (Raspberry Pi Pico)** for Pico
   - **MicroPython (ESP8266 / ESP32)** for ESP boards
5. Select the correct **USB / COM port**
6. Open the correct file
7. Click **Run** ▶️

👉 Output appears in the **Shell window**

🔷 PART A — Raspberry Pi Pico (Mandatory PDF Tasks)

✅ Task 1 — Internal Temperature Sensor

🔌 Connections

No external connections

Why:

- RP2040 contains a built-in temperature sensor
- Connected internally to **ADC channel 4**

🧠 Why the code is written this way

- `ADC(4)` selects the internal sensor
- Voltage is converted to °C using the datasheet formula
- `print()` sends data via USB serial to the PC

What this proves

- ADC usage
- Analog → physical conversion
- Pico → PC communication

✅ Task 2 — Joystick Raw Values

🔌 Connections

Joystick	Pico	Purpose
GND	GND	Common reference
VCC	3V3	Power
VRx	GP26 (ADC0)	X-axis
VRy	GP27 (ADC1)	Y-axis

Joystick	Pico	Purpose
SW	GP15	Button

🧠 Why the code is written this way

- Each joystick axis is an analog voltage
- ADC reads position as a number
- Raw values help verify correct wiring

What this proves

- External analog input
- Correct joystick wiring
- ADC resolution behavior

✅ Task 3 — Map Joystick Values (0–10)

🔌 Connections

Same as Task 2

🧠 Why mapping is needed

- Raw ADC values (0–65535) are not user-friendly
- UI elements require small numeric ranges

🧠 Code logic

- Linear mapping converts full ADC range to 0–10
- Used later for MQTT, dashboards, LEDs

✅ Task 4 — UART Communication (Pico ↔ ESP / PC)

🔌 Connections

Pico	Other device	Purpose
GP4 (TX)	RX	Send data

Pico	Other device	Purpose
GP5 (RX)	TX	Receive data
GND	GND	Common ground
3V3	3V3	Logic level

To send data from a Pico to a PC using your specified pins (GP4 TX to PC RX, GP5 RX to PC TX, GND to GND, 3V3 to 3V3), you'll use the Pico's UART for hardware serial, connecting via a USB-to-TTL adapter, then read the data on the PC with a Python script using the pyserial library and Thonny IDE for easy MicroPython/Python development, ensuring 3.3V logic levels are respected.

1. Hardware Connection (Pico to USB-to-TTL Adapter)

- **Pico GP4 (TX)** to **Adapter RX** (Data Out from Pico)
- **Pico GP5 (RX)** to **Adapter TX** (Data In to Pico)
- **Pico GND** to **Adapter GND** (Common Ground)
- **Pico 3V3** to **Adapter VCC/3.3V** (Logic Level)
- Connect the **USB-to-TTL Adapter** to your PC via USB.

2. Pico (MicroPython) Code Use Thonny IDE (from thonny.org) to upload this code to your Pico:

python

```
from machine import UART
import time

# Initialize UART on GP4 (TX) and GP5 (RX) at 115200 baud
uart = UART(0, baudrate=115200)

print("Pico Serial Sender Started!")

while True:
    # Send a simple message every second
    message = "Hello from Pico!\n"
    uart.write(message)
    print(f"Sent: {message.strip()}")
    time.sleep(1)
```

3. PC (Python) Code On your PC, use a Python script with `pyserial` to read the data (install with `pip install pyserial`):

python

```
import serial
import time
```

```
# Find the correct COM port for your USB-to-TTL adapter (e.g., COM3 on Windows,
/dev/ttyUSB0 on Linux)
# Check Device Manager (Windows) or terminal (Linux: ls /dev/tty*)
port = 'COM3' # <--- CHANGE THIS TO YOUR PORT
baud_rate = 115200

try:
    ser = serial.Serial(port, baud_rate, timeout=1)
    print(f"Connected to {port}")
    time.sleep(2) # Wait for connection to establish

    while True:
        if ser.in_waiting > 0:
            line = ser.readline().decode('utf-8').strip()
            print(f"Received: {line}")
except serial.SerialException as e:
    print(f"Error connecting to serial port: {e}")
    print("Please check your port and connection.")
except KeyboardInterrupt:
    print("Exiting...")
finally:
    if 'ser' in locals() and ser.is_open:
        ser.close()
        print("Serial port closed.")
```

Explanation:

- **UART (Universal Asynchronous Receiver/Transmitter):** The protocol used for serial communication, using dedicated TX (Transmit) and RX (Receive) pins.
- **TX to RX, RX to TX:** Data flows in opposite directions across the two devices.
- **3.3V Logic:** The Pico uses 3.3V signals; using a 3.3V USB-to-TTL adapter ensures compatibility, preventing damage.
- **Thonny IDE:** A simple Python IDE for MicroPython, making it easy to upload code to the Pico.
- **pyserial:** The Python library for serial port communication on the PC.

⚠️ TX and RX **must be crossed**

🧠 Why the code is written this way

- UART enables device-to-device communication
- 9600 baud is stable and standard
- Received characters trigger actions (LED toggle)

✅ Task 5 — JSON + Node-RED

🔌 Connections

Same as Task 4

🧠 Why JSON is used

- Structured
- Human-readable
- IoT industry standard

🧠 System behavior

- Pico sends JSON via UART
- Node-RED parses JSON
- Dashboard displays values (gauge, chart, debug)

✅ Task 6 — Temperature in °C / K / °F

🔌 Connections

No new wiring

🧠 Why the code is written this way

- Temperature converted into multiple units
- JSON groups all values in one message
- Enables flexible UI display

✅ Task 7 — Temperature vs Time

🔌 Connections

Same as Task 6

🧠 How it works

- Pico sends temperature periodically
- Node-RED chart plots temperature over time

🔷 PART B — ESP ↔ ESP MQTT (Instructor Extension)

🧠 System Concept

```
ESP8266 → Wi-Fi → MQTT Broker → Wi-Fi → ESP32 →Display
```

There is **no direct wire** between ESP boards.

🟦 ESP8266 — MQTT Publisher

🔌 Connections

- USB for power and programming
- Optional button on GPIO

🧠 Why the code is written this way

- Wi-Fi connection required for MQTT
- Publisher sends messages to broker
- Decouples sender and receiver

Role

- Message source
- Remote control unit

⬛ ESP32 — MQTT Subscriber + Display

🔌 LCD Connections (SPI)

LCD	ESP32
VCC	3V3

LCD	ESP32
GND	GND
MOSI	GPIO 23
SCK	GPIO 18
CS	GPIO 5
DC	GPIO 2
RST	GPIO 4

🧠 Why the code is written this way

- SPI is fast and reliable for displays
- Callback function updates display on message arrival
- Clear separation between communication and UI

🧪 Correct Testing Order

1. Start MQTT broker

```
mosquitto -v
```

2. Flash **ESP32**
3. Flash **ESP8266**
4. Observe LCD updates
5. Verify with:

```
mosquitto_sub -t class/display -v
```

📦 PROJECT NO. 1 — IMPLEMENTATION INSTRUCTIONS

(Using previously developed codes from Exercise 02 + extensions)

1️⃣ Project Objective

The goal of **Project No. 1** is to build an **IoT measurement and visualization system** that:

- reads temperature data from a sensor,
- sends the data in **JSON format**,
- visualizes the data using **Node-RED**,
- allows **user control via buttons**,
- and presents data using **indicators and charts**.

All required functionality is implemented using the **existing scripts** created during **Exercise 02**, without rewriting them.

2️⃣ Hardware Used in the Project

- Raspberry Pi Pico
- PC / Laptop (Node-RED + Thonny)
- USB cable
- (Extension) ESP8266, ESP32, LCD display

> ⚠️ Note:
>
> **Only the Pico + Node-RED are required for Project No. 1.**
>
> ESP8266 / ESP32 are an **extension**, not mandatory.

3️⃣ Software Used

- **MicroPython firmware** (Pico)
- **Thonny IDE**
- **Node-RED**
- **Mosquitto MQTT broker** (only for extension)

4️⃣ Project File Set (Already Implemented)

The project uses the following **existing files**:

Raspberry Pi Pico (core of Project No. 1)

- `task1_temp.py`
- `task2_joystick_raw.py`
- `task3_joystick_map.py`
- `task4_uart.py`
- `task5_json_uart.py`
- `task6_temp_units.py`

ESP Extension (optional)

- `ESP8266/main.py`
- `ESP32/main.py`

5️⃣ Step-by-Step Project Instructions

STEP 1 — Verify Temperature Measurement (Sensor Data)

Used file:

`task1_temp.py`

What to do

1. Connect **Raspberry Pi Pico** to the PC via USB
2. Open **Thonny**

3. Select interpreter:

```
MicroPython (Raspberry Pi Pico)
```

4. Open and run `task1_temp.py`

What happens

- Pico reads its **internal temperature sensor**
- Temperature is printed in °C in the Thonny Shell

Why this step matters

✔️ Confirms the sensor works

✔️ Confirms ADC and MicroPython setup

✔️ Satisfies *"Sensor Data Presentation"* requirement

STEP 2 — Prepare Data for User Interaction (Optional Input)

Used files:

- `task2_joystick_raw.py`
- `task3_joystick_map.py`

What to do

1. Connect joystick to Pico (ADC pins)
2. Run `task2_joystick_raw.py` to verify wiring
3. Run `task3_joystick_map.py` to map values to 0–10

Why this step matters

- Demonstrates **analog input handling**
- Shows how raw data is converted into usable ranges
- Prepares values suitable for UI controls

> ℹ️ This step supports Project No. 1 but is not mandatory if only temperature is used.

STEP 3 — Enable Device Communication (UART Channel)

Used file:

`task4_uart.py`

What to do

1. Connect Pico UART:
   - GP4 → RX
   - GP5 → TX
   - GND → GND
2. Run `task4_uart.py`

What happens

- Pico sends characters over UART
- Pico reacts to incoming commands

Why this step matters

✔️ Establishes a **communication channel**

✔️ Required for Node-RED interaction

✔️ Enables remote control logic

STEP 4 — Send Sensor Data as JSON (Core Project Step)

Used file:

`task5_json_uart.py`

What to do

1. Keep UART wiring from Step 3
2. Open **Node-RED**

3. Create flow:

```
serial-in →json →debug / gauge / chart
```

4. Run `task5_json_uart.py`

What happens

- Pico sends **JSON-formatted data**
- Node-RED parses JSON automatically
- Data appears in the Debug panel

Why this step matters

✔️ Satisfies **JSON communication requirement**

✔️ Connects Pico to Node-RED

✔️ Forms the backbone of Project No. 1

STEP 5 — Build the User Interface (Node-RED)

Used with:

`task5_json_uart.py` or `task6_temp_units.py`

UI elements to create

- **Gauge** → current temperature
- **Chart** → temperature vs time
- **Buttons**:
  - Start measurement
  - Stop measurement
  - Instant measurement

How buttons work

- Buttons send **JSON commands**
- Commands go via UART

- Pico reacts based on received command

Why this step matters

✔️ Satisfies **User Interface requirement**

✔️ Provides **control elements (3 buttons)**

✔️ Enables real-time visualization

STEP 6 — Display Temperature in Multiple Units

Used file:

`task6_temp_units.py`

What to do

1. Run `task6_temp_units.py`
2. Update Node-RED to display:
   - Celsius
   - Kelvin
   - Fahrenheit

Why this step matters

✔️ Demonstrates **data processing**

✔️ Shows advanced JSON usage

✔️ Can replace or extend Task 5

STEP 7 — Display Temperature vs Time (Chart)

Used files:

`task6_temp_units.py` + Node-RED chart

What happens

- Pico sends temperature periodically
- Node-RED plots values over time

Why this step matters

✔️ Satisfies **Data Visualization requirement**

✔️ Mandatory part of Project No. 1

6️⃣ (Optional) Extension — ESP-to-ESP MQTT Display

Used files:

- `ESP8266/main.py`
- `ESP32/main.py`

Purpose

- Demonstrates **wireless data distribution**
- Shows scalable IoT architecture

Important note

> This is an extension, not required for Project No. 1.
>
> It does **not replace Node-RED**.

7️⃣ Final Project Validation Checklist

✔️ Node-RED UI used

✔️ Temperature sensor data displayed

✔️ At least 3 control buttons

✔️ Chart (temperature vs time)

✔️ Gauge / indicator

✔️ JSON-based communication

✔️ Uses previously developed code