

Lab 1: Arduino - Common Part (Group Work Summary)

Ubiquitous Computing

Group: Patrick Denz, Tomke Velten, Dennis Roemmich, Jan Slama, Tiemor Amjad

Date: 06.11.2025

1. Introduction and Objectives

This report summarises the joint efforts of the group in completing the core exercises (0-4) of Lab 1. The primary objective of this laboratory exercise was to introduce the **Arduino Nano RP2040 Connect** development board and its integrated sensors as a low-cost, high-performance platform for Ubiquitous Computing applications.

Specifically, the group aimed to:

1. Establish a working development environment for the RP2040 chip.
2. Implement control logic for the on-board **RGB LED**.
3. Interface with and read data from the integrated **Inertial Measurement Unit (IMU)**, including its temperature sensor.
4. Utilise the **PDM Microphone** for audio sensing.
5. Develop a complete application—the **Posture Detector**—using sensor fusion techniques (Madgwick Filter) to process IMU data.

2. Development Environment and Initial Setup

2.1. Toolchain and Hardware

The group used the **Arduino IDE** and installed the required **Arduino Mbed OS Nano Boards** package to enable compilation and upload to the Arduino Nano RP2040 Connect. Initial setup involved identifying the correct serial port and including necessary libraries, specifically `<WiFiNINA.h>` (required for the platform and accessing hardware like the RGB LED) and `<Arduino_LSM6DSOX.h>` (for the IMU).

2.2. Exercise 0: Blink (Verification)

The standard Blink sketch was successfully loaded onto the board. This confirmed the correct driver installation, board selection, and communication with the host PC. The successful blinking of the built-in LED validated the fundamental control path of the microcontroller.

3. Implementation and Results of Core Exercises (1-4)

3.1. Exercise 1: Internal RGB Control

The objective was to make the on-board RGB LED smoothly transition between Red, Green, and Blue over an interval of 0.5 seconds per segment.

Implementation: The group implemented a custom function, `fade(NinaPin from, NinaPin to)`, to handle the colour blending. This function iterated from a value of 0 to 255. A key piece of logic was the use of `analogWrite()`:

- The **Arduino Nano RP2040's RGB pins are inverted** (a value of 255 is OFF, and 0 is ON).
- The function used `analogWrite(to, 255 - rval)` and `analogWrite(from, rval)` to ensure a smooth cross-fade. As one colour fades *in* (value decreases from 255 to 0), the other colour fades *out* (value increases from 0 to 255).
- A `delay(2)` inside the loop resulted in a smooth 0.5 second 255 * 2 ms transition time between the primary colours (Red -> Green -> Blue).

Result: The RGB LED successfully displayed a smooth, continuous colour cycle, confirming the proper utilisation of Pulse Width Modulation (PWM) for analogue-like colour control.

3.2. Exercise 2: Temperature Sensor

This exercise focused on reading the ambient temperature using the sensor embedded within the IMU module and using the result to control the RGB LED color.

Implementation:

1. **Setup:** The IMU was initialised using `IMU.begin()`.
2. **Sensing:** The `IMU.readTemperature(temperature_deg)` function was called to read the temperature in Celsius.
3. **Visualisation:** A helper function, `set_led(int t)`, implemented the required threshold logic:
 - $t < 20^{\circ}\text{C}$ **Blue LED** (Too Cold)
 - $20^{\circ}\text{C} < t < 32^{\circ}\text{C}$: **Green LED** (Optimal)
 - $t > 32^{\circ}\text{C}$: **Red LED** (Too Hot)
4. **Efficiency:** The code checked if the new temperature differed from the `old_val` before calling `set_led`, which prevents unnecessary pin writes. The temperature was reported via the Serial Port every 2 seconds.
- 5.

Result: The device accurately displayed the IMU temperature on the Serial Monitor and correctly assigned the RGB LED colour according to the defined thermal zones.

3.3. Exercise 3: Microphone Interaction

The goal was to integrate the PDM (Pulse Density Modulation) microphone, read audio samples, and demonstrate signal processing and reaction.

Implementation:

1. **Libraries:** The dedicated `<PDM.h>` library was included for the microphone interface.
2. **Callback Function:** The non-blocking `PDM.onReceive(onPDMdata)` function was used. The `onPDMdata` function runs in an Interrupt Service Routine (ISR) context, efficiently reading available audio data into a `sampleBuffer`.

3. **Signal Plotting:** The main `loop()` waited for the `samplesRead` flag. When new samples were available, they were printed to the Serial Port, allowing visualisation on the **Serial Plotter** for real-time analysis of the audio signal.
4. **Visual Alarm:** A basic noise threshold detection was implemented: if any individual sample value was outside the range of -10,000, 10,000, the code triggered a logic flip (`LED_SWITCH`) to turn the **Blue LED** ON or OFF for 1 second, demonstrating the device's ability to react to sudden, loud audio spikes.

Result: The Serial Plotter showed the microphone's output signal, and loud noises successfully triggered the Blue LED, confirming data flow from the PDM sensor.

3.4. Exercise 4: Posture Detector

This was the most complex exercise, combining multiple sensors (Accelerometer, Gyroscope) and a sophisticated filtering algorithm (Madgwick) to determine the orientation and detect incorrect posture.

Implementation:

1. **Sensor Fusion:** The `<MadgwickAHRS.h>` library was used to fuse the raw accelerometer and gyroscope data, yielding stable orientation estimates (Roll, Pitch, Heading).
2. **Data Processing:** Custom helper functions (`convertRawAcceleration`, `convertRawGyro`) were used to convert the raw 16-bit sensor values into standard units (G-forces and degrees/second), which are required by the `filter.updateIMU()` function.
3. **Posture Check (Pitch):** The correct posture was defined as the board being approximately vertical, which corresponds to the **Pitch** angle being around 90° (depending on mounting). The detection logic triggered an alarm if the `pitch` was outside the range of 100°, 80°.
4. **Alarm System:** The `alert_and_blink_led()` function was used for visual feedback. If the posture was wrong, the LED blinked for 5 cycles (10 steps) using a custom colour (Yellow-Green, achieved by setting `LEDG` to a partial value) before turning off.
5. **Temperature Integration:** The `check_temperature()` function from Exercise 2 was integrated, providing concurrent monitoring of the room environment (temperature-based LED colour change: Blue/Red for low/high temps).

Result: The Posture Detector successfully calculated stable orientation values. When the device was tilted beyond the acceptable 10° range from the target vertical angle, it triggered a visual blinking alarm and reported the error via the Serial Monitor.

4. Challenges and Resolution

Throughout the practical implementation, the group encountered several technical challenges typical of initial microcontroller development.

Challenge 1: Inverted RGB LED Logic

- **Description and Impact:** In Exercise 1, using `analogWrite(LED_R, 255)` resulted in the LED being OFF, and `analogWrite(LED_R, 0)` resulted in the LED being ON. This meant standard fading logic was initially reversed.
- **Resolution Strategy:** The inversion was compensated for by applying the formula $255 - \text{value}$ within the `analogWrite()` calls for fading. This ensured 0 meant fully ON and 255 meant fully OFF, fixing the unexpected hardware-level logic.

Challenge 2: IMU Initialization Failure

- **Description and Impact:** During the initial setup for Exercise 2 (Temperature), the `IMU.begin()` call frequently failed to return a successful status, halting the program execution with a "Failed to initialize IMU!" message.
- **Resolution Strategy:** This was resolved by adding a small delay (500 ms) after the initial `Serial.begin()` call and ensuring that the IMU was the *first* peripheral to be initialized in the `setup()` block before other sensor readings were attempted.

Challenge 3: PDM Microphone Callback Execution

- **Description and Impact:** In Exercise 3, we experienced difficulties reading PDM data reliably and occasionally faced compile-time warnings when trying to print `Serial` messages inside the `onPDMdata` function.
- **Resolution Strategy:** The group confirmed that the `onPDMdata` function operates in an **Interrupt Service Routine (ISR)** context, where calling functions like `Serial.print()` is unsafe and can lead to timing issues. The resolution was to only set the volatile `samplesRead` counter within the ISR and move all data processing and `Serial` printing into the main, non-interrupt-driven `loop()`.

Challenge 4: Madgwick Filter Rate Configuration

- **Description and Impact:** The Madgwick filter in Exercise 4 requires a `sampleRate` for accurate fusion. Initial calculations were unstable due to an incorrect rate or the high update frequency within the `loop()`.
- **Resolution Strategy:** We utilized the IMU's reported sample rate (approximately 104Hz) as the input for the filter initialization. Additionally, instead of relying on the CPU to time the updates, we used a long `delay(2000)` at the end of the `loop()` to control the output rate of the final orientation angle and prevent excessive `Serial` printing (controlled by the `counter` variable).

5. Conclusion

The successful completion of Lab 1 established a firm foundation in microcontroller programming and sensor integration using the Arduino Nano RP2040 Connect. The group gained valuable experience in working with crucial components of Ubiquitous Computing devices: sensing (IMU, Temperature, Microphone), actuation (RGB LED), and data fusion (MadgwickAHRS).

The final device in Exercise 4 demonstrated a functioning, multi-sensor system capable of monitoring both environmental conditions and user posture, with a clear visual feedback mechanism. The challenges encountered were instrumental in understanding the importance of low-level programming aspects, such as hardware-specific pin behaviour (inverted logic), stable sensor initialization, and proper handling of interrupt-driven data reading. These principles are fundamental for developing robust and efficient embedded systems.