Lab 1: Introduction

Task 1: Warning System

Code Implementation for Task 1

In developing the code for Task 1, I integrated functionality from previous modules and incorporated new components. The code architecture revolves around three specialized C++ classes designed for modularity.

Class Design

- **LED Class**: Controls LED functions, simplifying and reducing redundancy in LED operations.
- **Status Controller Class**: Manages blink timing and logic for LEDs, allowing configurable intervals and durations.
- **Log Class**: Helps to print the Sensor information out to the Serial Monitor.

Core Functions

Two primary functions manage the system's core monitoring:

- **handleVolumeStatus**: Monitors and responds to volume levels, triggering alerts if volume exceeds the set threshold.
- **handlePitchStatus**: Tracks pitch orientation data, enabling alerts when tilt surpasses the maximum allowed angle.

Setup Phase

During initialization, three global objects are instantiated:

- Two **Status Controller Class** objects manage pitch and volume indicators independently.
- A **LED Class** instance for a green LED signals the system's normal operating state (illuminated when no alerts are active).

Main Loop Execution

Within the main loop, both volume and pitch status functions assess the system's current state, updating the blinking indicators as necessary. Each **Status Controller** instance toggles its associated LED based on updated thresholds.

A conditional check then evaluates the status of pitch and volume indicators. If either is active, the green LED turns off, signaling an alert; if both are inactive, it remains on to indicate normal operation.

System Output

The Singleton instance of the **Log class** prints the sensor information to the Serial Monitor, including the current pitch and volume status. This information is set and continuously updated within the core functions previously mentioned. At the **end** of the **loop**, it prints the gathered information.

Linux Permissions and Library Challenges in Arduino Development

When **working** on a Debian-based Linux distro, using the **latest** Arduino IDE for coding on an Arduino RP2040 Connect seemed straightforward—until it came time to upload code to the board. While writing and compiling code in the IDE posed no issues, the upload attempt returned a **permission's error**, blocking the IDE from writing to the device. Even setting read and write permissions globally with `chmod 666 /dev/ttyACM0` didn't solve the issue.

Eventually, to continue work without interruption, I had to install a Windows OS on my machine, where uploading through the IDE worked **seamlessly**.

During the development phase, most testing occurred locally on the development machine, which allowed for code verification and troubleshooting before transferring to the Arduino. However, using the library proved tricky, especially with the IMU sensor. Errors arose when trying to read values, which I traced back to **missing** a successful read check. Despite using the IMU's `Available` method, the values weren't consistently checked for success, leading to intermittent read failures.

Suggested Improvements

Enhancing the current setup with an LC-Display (LCD), instead of relying solely on an RGB LED, would greatly improve usability and provide more comprehensive feedback directly on the device. With an LCD, users could **see real-time data** like temperature, ambient sound levels, and pitch measurements. To add a fun twist, visual elements like a plane graphic could illustrate pitch, yaw, and roll, enhancing clarity for end users. Additionally, icons—such as a volume symbol or a temperature warning—could light up when thresholds are reached, similar to dashboard indicators in a car, **making it clear** when values exceed safe or expected limits.

For data enthusiasts, integrating a **REST API** to log values over time would allow for **statistical analysis**. This setup could chart readings, creating visual diagrams for easier data tracking and long-term analysis.

Task 2: Setting up WiFi and a Basic HTTP Server to Control the Arduino

Establishing or Creating a WiFi Network

The Arduino RP2040 Connect can either connect to an existing WiFi network or create its own WiFi Access Point (AP) for direct control. The official Arduino documentation provides guidance on both approaches, enabling users to either join a network or create one for local control. This setup is particularly useful for projects requiring remote management over a WiFi connection, such as through a simple HTTP server.

However, it's important to note some hardware and library limitations. The Arduino's WiFi chip supports only **IEEE 802.11b/g/n single-band at 2.4 GHz**, which means it operates on WiFi 4 or lower but not on the 5 GHz band. This won't impact basic data transfer needs for most Arduino projects, though it may **limit connectivity options** in areas where 5 GHz is the only available network (ABX00053-datasheet.pdf).

Another consideration is that the **WiFiNINA** library, which provides essential WiFi functionality, only supports WPA and WPA2 security protocols; it currently lacks WPA3 compatibility. This limitation may affect **network security or availability** in WPA3-only environments.

The WiFiNINA library also includes a DHCP server, simplifying setup by automatically assigning IP addresses to connected devices. By default, the Arduino will assume the IP address `192.168.4.1`, but this can be customized using `WiFi.config`, which also enables IP configuration when connecting to an existing network. In such cases, if no IP is assigned via `WiFi.config`, the board will rely on DHCP.

When connecting to a standard WiFi network, an SSID and password are required. Note that for WPA, both the SSID and password must be **at least 8 characters** in length; shorter credentials will result in connection failures.

Setting Up the HTTP Server on Arduino

Once connected to a network as a client or an access point (AP), you can use `WiFiServer` to create a basic HTTP server on the Arduino. This server operates over a TCP socket, handling simple requests to enable remote control, ideal for IoT and interactive applications. To start, specify the port in the `WiFiServer` constructor for the server to listen on. After setup, any TCP client (like a web browser) can connect to it. To accept incoming connections, use the `WiFiServer.available()` method, which returns a client object if a **connection is present**; otherwise, it returns `NULL`.

To deploy an HTTP service on an Arduino, developers need to implement fundamental HTTP handling methods directly on the device, as the Arduino lacks a dedicated web server framework. In this guide, we'll walk through creating a simple HTTP/1.1 handler capable of serving a basic HTML index file and processing commands via GET requests.

Using the Arduino's client object from its server, we can monitor for incoming data by invoking `client.available()`, which signals when new data is present in the client buffer. Once data is available, **characters are retrieved sequentially** using `client.read()`. If no data is detected, the system terminates the connection to free resources.

To send responses back to the client, the Arduino provides `client.print(...)` for writing individual characters to the data stream and `client.println(...)` to write data with a newline appended automatically. Together, these functions offer the foundational tools needed for simple HTTP communication on Arduino’s limited hardware.

Implementing a Basic HTTP/1.1 Handler

The HTTP/1.1 protocol is structured in two main sections: the **header** and the **body**. Here’s a simplified approach to handling HTTP requests and responses:

1. **HTTP Request**: An HTTP request typically starts with a method, such as `GET`, followed by the requested URL. For instance, `GET /index.html` requests the main page.
2. **HTTP Response**: In the HTTP response, the first line contains the HTTP version (`HTTP/1.1`), followed by a status code and a status message, such as `200 OK`, indicating a successful request. Then, include a `Content-Type` header specifying the data type, like `text/html`, which tells the client (e.g., a browser) that it will receive HTML content.
3. **Body Content**: After defining headers, add a blank line to signal the start of the body content. Here, you can include HTML or other data. For example, to display a basic web page, include your HTML code within the response body.
4. **Sending Data**: If you need to send additional data from the client to the server, the process is similar: start with headers, a blank line, and then the content in the body section.
5. **Closing the Connection**: End each request and response with an additional blank line. This ensures the browser recognizes the completion of the communication.

This simple HTTP handler implementation on the Arduino allows for serving basic web pages and handling simple GET commands, enabling remote interaction with the Arduino.

Controlling the HTTP Server with Arduino

To control our Arduino LED via the HTTP Server, we need to develop a straightforward HTML interface that includes buttons linking to **specific URLs**. These URLs allow the server handler to determine the appropriate action. For RGB functionality, we establish six endpoints: three for enabling specific colors and three for disabling them. The HTML interface will feature six buttons, each corresponding to these endpoints, enabling users to easily turn the LEDs on or off.

To implement this functionality in our handler, we check if the incoming client request begins with `GET /RH`. If it does, this indicates that the user intends to **enable** (set high) the red LED. And for `GET /RL` indicates that the User wants to turn off the LED.