**FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)**

****

**Embedded Systems || CSCS-306**

**Section A**

**Project**

**Google Sheet LED Detection and ON Time Calculation**

**Abdullah Mehtab || 241607845**

**Umm-e-Abiha || 241607845**

**A screenshot of a computer program

Description automatically generated**

**A close-up of a message

Description automatically generated**

# Introduction

In the realm of **Embedded Systems**, integrating hardware with cloud services opens up a multitude of possibilities. This report delves into an advanced project that builds upon Lab 10, where we extended the functionality of an ESP32 microcontroller to interact with Google Sheets. The primary objective of this lab is to not only log the status of multiple LEDs but also to calculate the total time each LED remains **ON** within a user-specified time range. This is achieved by creating a seamless interaction between the ESP32, Google Sheets, and a web or mobile application. Building upon the foundational work from Lab 10, where we successfully integrated the ESP32 with Google Sheets for logging LED statuses

**Logic/Algorithm in Simple English**

To comprehend the system's functionality, let's break down the logic into straightforward steps:

1. **Setup Phase:**
   * **Hardware Configuration:**
     + Connect four LEDs to the ESP32 through GPIO pins 5, 18, 19, and 21 using 220k Ohm resistors.
     + Ensure all other connections are properly grounded.
   * **Software Configuration:**
     + Install and set up the Arduino IDE.
     + Connect the ESP32 to the Arduino IDE for programming.
2. **LED Status Logging:**
   * Monitor each LED's status (ON/OFF) using the ESP32.
   * Whenever an LED's state changes, log this status along with a timestamp to a Google Sheet via an HTTP request.
3. **User Interaction via Web/Mobile App:**
   * **Input:**
     + The user specifies a time range (start date and time, end date and time) and selects an LED number.
   * **Processing:**
     + The web/mobile app sends these parameters to a Google Apps Script web application.
     + The script processes the data in the Google Sheet to calculate the total time the selected LED was ON within the specified range.
   * **Output:**
     + The calculated ON time is returned to the user in a readable format (hours, minutes, seconds).
4. **Error Handling:**
   * Ensure robust error checks at every stage, including Wi-Fi connectivity, data logging, and HTTP requests.

**Prerequisite Work**

Before embarking on the project, ensure the following setups are in place:

**Hardware Setup**

1. **Breadboard Configuration:**
   * **LED Connections:**
     + Connect four LEDs to the ESP32 using GPIO pins **5, 18, 19,** and **21**.
     + Each LED should be connected in series with a **220k Ohm** resistor to limit current and prevent damage.
     + The cathode (shorter leg) of each LED connects to the ground rail on the breadboard.
   * **Grounding:**
     + Ensure all LEDs share a common ground connection with the ESP32 to maintain consistent reference voltage levels.
2. **ESP32 Connection:**
   * Connect the ESP32 to your computer using a USB cable.
   * Ensure the board is recognized by the Arduino IDE for seamless programming and debugging.

**Software Setup**

1. **Arduino IDE Installation:**
   * Download and install the latest version of the **Arduino IDE** from the official website.
2. **ESP32 Board Configuration:**
   * In the Arduino IDE, navigate to **File > Preferences**.
   * Add the following URL to the **Additional Board Manager URLs** field:

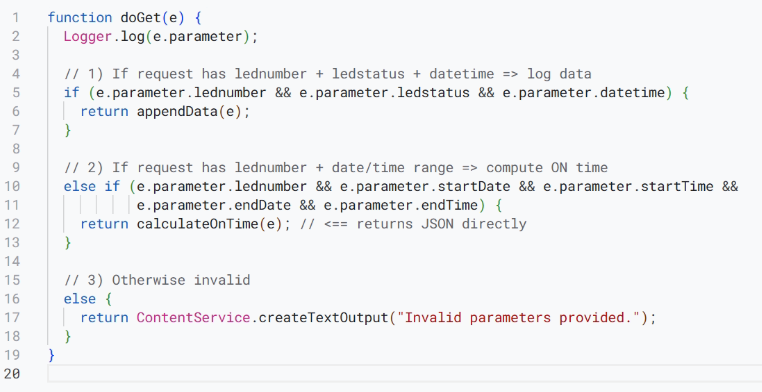
https://dl.espressif.com/dl/package\_esp32\_index.json

* + Go to **Tools > Board > Boards Manager**, search for **ESP32**, and install the package.

1. **Library Installation:**
   * Install necessary libraries such as **WiFi.h**, **HTTPClient.h**, and **ArduinoJson.h** via the **Library Manager** in the Arduino IDE.
2. **Google Sheet and Apps Script Setup:**
   * Create a Google Sheet named **"EmbeddedProject"** to store LED status logs.
   * Implement the provided Google Apps Script to handle data logging and ON time calculations.
   * Deploy the script as a **Web App** with appropriate permissions to accept HTTP requests from the ESP32.

# Code Snippets with Explanations

In this section, we will explore the **Google Apps Script** code provided for handling HTTP requests, logging LED statuses, and calculating the total ON time within a specified range. Each part of the code is dissected to understand its functionality comprehensively.

**1. doGet Function**

**Explanation**

* **Function Purpose:**
  + The doGet function serves as the entry point for all HTTP GET requests to the web app.
  + It determines the nature of the request based on the parameters received and delegates the task accordingly.
* **Parameters (e):**
  + Represents the event parameter containing all incoming request data.
* **Logger.log(e.parameter):**
  + Logs the incoming parameters for debugging purposes.
* **Conditional Checks:**
  + **Data Logging:**
    - Checks if the request contains lednumber, ledstatus, and datetime.
    - If **true**, it calls the appendData function to log the LED status.
  + **ON Time Calculation:**
    - Checks if the request contains lednumber, startDate, startTime, endDate, and endTime.
    - If **true**, it calls the calculateOnTime function to compute the total ON time.
  + **Invalid Request:**
    - If none of the above conditions are met, it returns an "Invalid parameters provided" message.

**2. appendData Function**

**Explanation**

* **Function Purpose:**
  + The appendData function logs the status of an LED along with a timestamp into the Google Sheet.
* **Variables:**
  + sheet\_id: The unique identifier of the Google Sheet where data is stored.
  + sheet\_name: The name of the specific sheet within the Google Sheet document.
  + ss: Opens the Google Sheet using the provided sheet\_id.
  + sheet: Accesses the specific sheet named EmbeddedProject.
* **Parameter Extraction:**
  + Retrieves lednumber, ledstatus, and datetime from the incoming request.
* **Validation:**
  + Checks if any of the required parameters are missing.
  + If any parameter is absent, it returns an error message indicating missing parameters.
* **Data Logging:**
  + Appends a new row to the sheet with the following data:
    - **Datetime:** The exact date and time of the status change.
    - **LED Number:** Identifies which LED's status is being logged.
    - **Status:** Converts ledstatus from "1" or "0" to "ON" or "OFF" respectively.
* **Success Response:**
  + Returns a confirmation message upon successful data logging.

**3. calculateOnTime Function**

A computer screen shot of a program

Description automatically generated

**Explanation**

* **Function Purpose:**
  + The calculateOnTime function computes the total duration an LED remained **ON** within a specified time range.
* **Variables:**
  + sheet\_id & sheet\_name: Identifiers to access the correct Google Sheet.
  + ss: Opens the Google Sheet.
  + sheet: Accesses the EmbeddedProject sheet within the Google Sheet.
  + ledNumber, startDate, startTime, endDate, endTime: Parameters received from the user's request.
* **Validation:**
  + Ensures all necessary parameters are present.
  + Returns an error message if any parameter is missing.
* **Datetime Parsing:**
  + Combines startDate and startTime to create a startDateTime object.
  + Combines endDate and endTime to create an endDateTime object.
* **Data Retrieval:**
  + Fetches all data from the sheet using getDataRange().getValues().
  + Initializes totalOnMillis to accumulate the total **ON** duration in milliseconds.
* **Data Processing Loop:**
  + Iterates through each row of data starting from the second row (index 1) to skip headers. Extracts recordDate, recordLED, and recordStat from each row.
* **Condition Checks:**
  + **LED Number Match:** Ensures the current record pertains to the specified ledNumber.
  + **Datetime Range Check:** Ensures the record's datetime falls within the specified range.
  + **Status Check:** If the LED is **ON**, calculates the duration it remained **ON**.
* **Duration Calculation:**
  + Determines the nextTime which is either the timestamp of the next **OFF** event or the end of the specified range.
  + Adds the difference between nextTime and recordDate to totalOnMillis.
* **Result Formatting:**
  + Converts the total duration from milliseconds to seconds, then to hours, minutes, and seconds.
* **JSON Response:**
  + Constructs a JSON object with hours, minutes, and seconds.
  + Returns the JSON object with the appropriate MIME type for easy parsing on the client side.

**Error Handling**

Robust error handling is crucial for ensuring the system's reliability. The following outlines the error handling mechanisms integrated into the project:

**1. Parameter Validation:**

* **Function:** Both appendData and calculateOnTime functions validate the presence of required parameters.
* **Action:** If any parameter is missing, the system returns an explicit error message indicating the absence of necessary data.

**2. Google Sheet Access:**

* **Potential Issues:**
  + Invalid sheet\_id or sheet\_name leading to failures in accessing the sheet.
  + Network issues preventing access to Google Sheets.
* **Mitigation:**
  + Ensure correct sheet\_id and sheet\_name are provided.
  + Implement retry mechanisms or alert systems in case of access failures.

**3. Data Parsing and Calculation:**

* **Potential Issues:**
  + Malformed datetime strings causing Date object creation failures.
  + Unexpected data formats in the Google Sheet.
* **Mitigation:**
  + Utilize JavaScript's Date object carefully, ensuring input formats are consistent.
  + Incorporate try-catch blocks or conditional checks to handle unexpected data gracefully.

**4. HTTP Request Handling:**

* **Functionality:** The doGet function determines the type of request and routes it accordingly.
* **Potential Issues:**
  + Invalid or unexpected parameters leading to misrouting.
* **Mitigation:**
  + The final else condition in doGet captures any invalid requests, ensuring the system responds with a clear error message rather than failing silently.

**5. Response Consistency:**

* **Functionality:** Ensures that responses are consistent and in expected formats (e.g., JSON for successful ON time calculations).
* **Potential Issues:**
  + Inconsistent response types causing parsing issues on the client side.
* **Mitigation:**
  + Explicitly set MIME types and ensure all success and error responses follow a predictable structure.

# Detailed Breakdown of Google Scripts Code

To ensure a comprehensive understanding, let's delve deeper into the Google Scripts code, examining each component meticulously.

**doGet Function**

**Parameters and Logging**

function doGet(e) {

Logger.log(e.parameter);

// ...

}

* **e.parameter:**
  + Contains all query parameters sent in the HTTP GET request.
  + Example: For a request URL like ...?lednumber=1&ledstatus=1&datetime=2025-01-07%2012:00:00, e.parameter would hold {lednumber: "1", ledstatus: "1", datetime: "2025-01-07 12:00:00"}.
  + **Logger.log(e.parameter):** Logs the parameters for debugging.

**Conditional Routing**

if (e.parameter.lednumber && e.parameter.ledstatus && e.parameter.datetime) {

return appendData(e);

}

else if (e.parameter.lednumber && e.parameter.startDate && e.parameter.startTime &&

e.parameter.endDate && e.parameter.endTime) {

return calculateOnTime(e);

}

else {

return ContentService.createTextOutput("Invalid parameters provided.");

}

* **First Condition:**
  + Checks for parameters necessary to log LED status (lednumber, ledstatus, datetime).
  + If present, delegates to appendData for logging.
* **Second Condition:**
  + Checks for parameters necessary to calculate ON time (lednumber, startDate, startTime, endDate, endTime).
  + If present, delegates to calculateOnTime for computation.
* **Else Condition:**
  + Catches any request that doesn't meet the above conditions.
  + Returns an error message indicating invalid parameters.

**appendData Function**

**Sheet Access and Data Extraction**

var sheet\_id = "1R7i7FmZVECeToWY2MPQPBVnUmgSYDv0y8x2fySPQb8U";

var sheet\_name = "EmbeddedProject";

var ss = SpreadsheetApp.openById(sheet\_id);

var sheet = ss.getSheetByName(sheet\_name);

* **sheet\_id:**
  + Unique identifier for the Google Sheet.
  + Replace with the actual ID of your Google Sheet.
* **sheet\_name:**
  + Name of the specific sheet within the Google Sheet document.
* **SpreadsheetApp.openById(sheet\_id):**
  + Opens the Google Sheet using its ID.
* **ss.getSheetByName(sheet\_name):**
  + Accesses the specified sheet within the Google Sheet.

**Parameter Extraction and Validation**

var ledNumber = e.parameter.lednumber;

var ledStatus = e.parameter.ledstatus;

var datetime = e.parameter.datetime;

if (!ledNumber || !ledStatus || !datetime) {

return ContentService.createTextOutput("Missing required parameters.");

}

* **Extracting Parameters:**
  + Retrieves lednumber, ledstatus, and datetime from the request.
* **Validation:**
  + Checks if any of the parameters are missing.
  + If any are missing, returns an error message to prevent incomplete data logging.

**Data Logging**

sheet.appendRow([datetime, ledNumber, ledStatus === "1" ? "ON" : "OFF"]);

* **appendRow:**
  + Adds a new row to the sheet with the provided data.
* **Data Composition:**
  + **Datetime:** Combined date and time string.
  + **LED Number:** Identifier for the LED.
  + **Status:** Converts ledstatus from "1" to "ON" and "0" to "OFF" for readability.

**Success Response**

return ContentService.createTextOutput("Data appended successfully.");

* **Response:**
  + Confirms successful data logging to the client.

**calculateOnTime Function**

**Sheet Access and Parameter Extraction**

var sheet\_id = "1R7i7FmZVECeToWY2MPQPBVnUmgSYDv0y8x2fySPQb8U";

var sheet\_name = "EmbeddedProject";

var ss = SpreadsheetApp.openById(sheet\_id);

var sheet = ss.getSheetByName(sheet\_name);

var ledNumber = e.parameter.lednumber;

var startDate = e.parameter.startDate;

var startTime = e.parameter.startTime;

var endDate = e.parameter.endDate;

var endTime = e.parameter.endTime;

if (!ledNumber || !startDate || !startTime || !endDate || !endTime) {

return ContentService.createTextOutput("Missing required parameters.");

}

* **Sheet Access:**
  + Similar to appendData, accesses the specific Google Sheet and sheet name.
* **Parameter Extraction:**
  + Retrieves lednumber, startDate, startTime, endDate, and endTime from the request.
* **Validation:**
  + Ensures all necessary parameters are present.
  + Returns an error message if any are missing.

**Datetime Parsing**

var startDateTime = new Date(startDate + " " + startTime);

var endDateTime = new Date(endDate + " " + endTime);

* **Combining Date and Time:**
  + Concatenates startDate with startTime and endDate with endTime to form complete datetime strings.
  + Converts these strings into JavaScript Date objects for comparison.

**Data Retrieval and Initialization**

var data = sheet.getDataRange().getValues();

var totalOnMillis = 0;

* **getDataRange().getValues():**
  + Retrieves all data from the sheet as a 2D array.
  + Each sub-array represents a row in the sheet.
* **totalOnMillis:**
  + Initializes a counter to accumulate the total ON duration in milliseconds.

**Processing Loop**

for (var i = 1; i < data.length; i++) {

var recordDate = new Date(data[i][0]); // Column A: Datetime

var recordLED = data[i][1]; // Column B: LED number

var recordStat = data[i][2]; // Column C: Status ("ON"/"OFF")

if (String(recordLED) === String(ledNumber) &&

recordDate >= startDateTime && recordDate <= endDateTime) {

if (recordStat === "ON") {

// Check next row or use end date/time

var nextTime = endDateTime;

if (i + 1 < data.length) {

var nextDate = new Date(data[i + 1][0]);

var nextLED = data[i + 1][1];

var nextStat = data[i + 1][2];

if (String(nextLED) === String(ledNumber) && nextStat === "OFF") {

nextTime = nextDate;

}

}

totalOnMillis += (nextTime - recordDate);

}

}

}

* **Loop Initialization:**
  + Starts from i = 1 to skip the header row (assuming the first row contains column titles).
* **Record Extraction:**
  + **recordDate:** Parses the datetime string from the first column.
  + **recordLED:** Retrieves the LED number from the second column.
  + **recordStat:** Retrieves the status ("ON" or "OFF") from the third column.
* **Condition Checks:**
  + **LED Number Match:** Ensures the record pertains to the specified ledNumber.
  + **Datetime Range:** Checks if the recordDate falls within the specified startDateTime and endDateTime.
* **ON Status Handling:**
  + If the status is "ON", it calculates the duration the LED remained **ON**.
  + **Next Time Determination:**
    - By default, nextTime is set to endDateTime.
    - If the next record exists and pertains to the same LED with an "OFF" status, nextTime is updated to the timestamp of that "OFF" event.
  + **Duration Calculation:**
    - The difference between nextTime and recordDate is added to totalOnMillis.

**Duration Conversion and JSON Response**

var totalSecs = Math.floor(totalOnMillis / 1000);

var hours = Math.floor(totalSecs / 3600);

var minutes = Math.floor((totalSecs % 3600) / 60);

var seconds = Math.floor(totalSecs % 60);

var result = {

hours: Math.abs(hours),

minutes: Math.abs(minutes),

seconds: Math.abs(seconds)

};

// Directly return JSON. No redirection, no extra HTML:

return ContentService

.createTextOutput(JSON.stringify(result))

.setMimeType(ContentService.MimeType.JSON);

* **Conversion Steps:**
  + **Milliseconds to Seconds:** Divides totalOnMillis by 1000 to get total seconds.
  + **Seconds to Hours, Minutes, Seconds:**
    - Calculates hours by dividing total seconds by 3600.
    - Calculates minutes by taking the remainder after hours and dividing by 60.
    - The remaining seconds are calculated using the modulus operator.
* **Result Object:**
  + Constructs a JSON object with hours, minutes, and seconds for easy consumption by the client application.
* **Response Formatting:**
  + Converts the result object to a JSON string.
  + Sets the MIME type to application/json to ensure proper parsing on the client side.
  + Returns the JSON response without any additional HTML or redirection.

**Summary of Error Handling**

Ensuring the system operates smoothly under various conditions is paramount. Here's a consolidated summary of the error handling mechanisms implemented:

1. **Parameter Validation:**
   * Both appendData and calculateOnTime functions rigorously check for the presence of required parameters.
   * Prevents incomplete or malformed data from being processed or logged.
2. **Sheet Access Verification:**
   * Ensures that the Google Sheet is accessible using the provided sheet\_id and sheet\_name.
   * Returns clear error messages if access fails, aiding in quick debugging.
3. **Data Integrity Checks:**
   * Validates the format and content of incoming data, especially datetime strings.
   * Prevents logical errors during duration calculations by ensuring consistent data formats.
4. **Response Consistency:**
   * All responses, whether success or error, are structured predictably.
   * Facilitates seamless parsing and handling on the client side.
5. **Logging for Debugging:**
   * Utilizes Logger.log to record incoming parameters and potential issues.
   * Aids developers in monitoring and troubleshooting the system.

**Methods from Google Apps Script Classes**

Understanding the methods utilized from various Google Apps Script classes is essential for grasping the system's functionality.

**1. SpreadsheetApp Class**

* **openById(id)**
  + **Functionality:** Opens a Google Sheet using its unique ID.
  + **Arguments:**
    - id (string): The unique identifier of the Google Sheet.
  + **Return Type:** Spreadsheet object representing the opened sheet.
  + **Example:**

var ss = SpreadsheetApp.openById(sheet\_id);

* **getSheetByName(name)**
  + **Functionality:** Retrieves a specific sheet within the Google Sheet document by its name.
  + **Arguments:**
    - name (string): The name of the sheet to retrieve.
  + **Return Type:** Sheet object representing the retrieved sheet.
  + **Example:**

var sheet = ss.getSheetByName(sheet\_name);

* **getDataRange()**
  + **Functionality:** Gets the range of all data present in the sheet.
  + **Arguments:** None.
  + **Return Type:** Range object representing the data range.
  + **Example:**

var data = sheet.getDataRange().getValues();

* **appendRow(rowContents)**
  + **Functionality:** Appends a new row to the sheet with the specified contents.
  + **Arguments:**
    - rowContents (array): An array representing the values of each cell in the new row.
  + **Return Type:** void.
  + **Example:**

sheet.appendRow([datetime, ledNumber, ledStatus === "1" ? "ON" : "OFF"]);

**2. ContentService Class**

* **createTextOutput(text)**
  + **Functionality:** Creates a text output object for HTTP responses.
  + **Arguments:**
    - text (string): The text to include in the response body.
  + **Return Type:** TextOutput object.
  + **Example:**

return ContentService.createTextOutput("Data appended successfully.");

* **setMimeType(mimeType)**
  + **Functionality:** Sets the MIME type of the content being returned.
  + **Arguments:**
    - mimeType (enum): The MIME type to set (e.g., ContentService.MimeType.JSON).
  + **Return Type:** TextOutput object with the updated MIME type.
  + **Example:**

.setMimeType(ContentService.MimeType.JSON);

**3. Date Object**

* **new Date(dateString)**
  + **Functionality:** Creates a new Date object from a date string.
  + **Arguments:**
    - dateString (string): The date and time in a recognizable format.
  + **Return Type:** Date object.
  + **Example:**

var startDateTime = new Date(startDate + " " + startTime);

**4. Logger Class**

* **log(message)**
  + **Functionality:** Logs a message to the Apps Script logger, useful for debugging.
  + **Arguments:**
    - message (any): The message to log.
  + **Return Type:** void.
  + **Example:**

Logger.log(e.parameter);

**Circuit Diagram**

A group of red and yellow wires on a white board

Description automatically generated

# Arduino ide (.ino) Code Explanation

**Preprocessor Directives and Their Functions**

In embedded programming, **preprocessor directives** are commands that are executed before the actual compilation of the code begins. They play a crucial role in managing code modularity, optimizing performance, and enhancing readability. The ESP32 Arduino IDE code leverages several preprocessor directives to include necessary libraries, define constants, and configure hardware settings.

* **#include**: Incorporates external libraries or modules into the program, enabling the use of their functionalities.
* **#define**: Defines constants or macros, allowing for easy configuration and modification of values used throughout the code.
* **#ifdef / #ifndef**: Facilitates conditional compilation, enabling or disabling code segments based on specific conditions.

Understanding these directives is essential for comprehending how different components of the code interact and function cohesively.

**Logic/Algorithm in Simple English**

To demystify the system's operation, let's outline the logic in straightforward terms:

1. **Setup Phase:**
   * **Initialize Serial Communication:** Establish communication between the ESP32 and the computer for debugging purposes.
   * **Mount File System (SPIFFS):** Access the internal file system to store and retrieve local CSV logs.
   * **Configure GPIO Pins:** Set up four GPIO pins to control four LEDs, ensuring they are initially turned off.
   * **Connect to Wi-Fi:** Establish a connection to the specified Wi-Fi network to enable internet-based functionalities.
   * **Synchronize Time via NTP:** Obtain the current time from an NTP server to ensure accurate timestamping.
   * **Initialize Web Server:** Set up a web server to handle HTTP requests for LED control and data queries.
2. **Main Operation Loop:**
   * **Handle Client Requests:** Continuously listen for and process incoming HTTP requests from the web server.
3. **LED Control and Logging:**
   * **Toggle LEDs:** Based on user inputs from the web interface, turn LEDs on or off.
   * **Log Status Changes:**
     + **Local CSV Logging:** Record each LED status change with a timestamp in a local CSV file stored in SPIFFS.
     + **Google Sheets Logging:** Send HTTP GET requests to a Google Apps Script Web App to log the status changes in a Google Sheet.
4. **User Interaction via Web Interface:**
   * **LED Control Interface:** Allow users to control each LED through a web-based interface with buttons to turn LEDs on or off.
   * **Query ON Time:**
     + **User Input:** Enable users to specify a time range and select an LED to query its total ON time within that period.
     + **Local Calculation:** Process the local CSV file to calculate the total ON time based on the user-defined range.
     + **Google Sheets Integration:** Provide a URL to query the Google Apps Script Web App for cloud-based calculations.
5. **Error Handling:**
   * **Wi-Fi Connectivity Checks:** Ensure that the ESP32 is connected to Wi-Fi before attempting to upload data to Google Sheets.
   * **File System Checks:** Verify the successful mounting of SPIFFS and the creation/access of the local CSV file.
   * **HTTP Request Handling:** Manage potential failures in HTTP GET requests when communicating with Google Sheets.

**IMPORTANT RRE-REQUISITE REMINDER**

**Google Sheet and Apps Script Setup:**

* + **Google Sheet Creation:** Create a Google Sheet named **"EmbeddedProject"** to store LED status logs.
  + **Google Apps Script Implementation:** Utilize the provided Google Apps Script to handle data logging and ON time calculations.
  + **Web App Deployment:** Deploy the script as a **Web App** with appropriate permissions to accept HTTP requests from the ESP32.

**Code Snippets with Explanations**

The core of this project lies in the **EmbeddedProject.ino** file, which orchestrates the interaction between the ESP32, Google Sheets, and the user interface. This section breaks down the code into manageable snippets, elucidating each component's purpose and functionality.

**1. Including Necessary Libraries**

#include <WiFi.h>

#include <HTTPClient.h>

#include <WebServer.h>

#include "time.h"

#include <SPIFFS.h>

**Explanation**

* **WiFi.h:**
  + **Purpose:** Facilitates Wi-Fi connectivity, enabling the ESP32 to connect to a wireless network.
  + **Functions:** Provides methods to connect to Wi-Fi, check connection status, and retrieve IP addresses.
* **HTTPClient.h:**
  + **Purpose:** Enables the ESP32 to perform HTTP operations, such as GET and POST requests.
  + **Functions:** Allows constructing HTTP requests, sending them, and handling responses.
* **WebServer.h:**
  + **Purpose:** Sets up a local web server on the ESP32 to handle incoming HTTP requests.
  + **Functions:** Defines server routes, handles client requests, and serves web pages.
* **time.h:**
  + **Purpose:** Manages time-related functions, particularly for synchronizing time via NTP.
  + **Functions:** Configures time settings, retrieves local time, and formats timestamps.
* **SPIFFS.h:**
  + **Purpose:** Provides access to the ESP32's internal SPI Flash File System (SPIFFS) for storing and retrieving files locally.
  + **Functions:** Handles file operations such as reading, writing, and appending data to files.

**2. Defining Wi-Fi Credentials**

// -------------------- WiFi Credentials --------------------

const char\* ssid = "Aalu";

const char\* password = "robot12!";

**Explanation**

* **ssid:**
  + **Type:** const char\* (constant character pointer)
  + **Purpose:** Stores the name of the Wi-Fi network the ESP32 will connect to.
  + **Value:** "Aalu"
* **password:**
  + **Type:** const char\*
  + **Purpose:** Stores the password for the specified Wi-Fi network.
  + **Value:** "robot12!"
* **Note:** These credentials are essential for establishing a connection to the internet, enabling the ESP32 to communicate with external services like Google Sheets.

**3. Google Apps Script Web App URL**

// -------------------- Google Apps Script Web App --------------------

const char\* googleScriptURL = "https://script.google.com/macros/s/AKfycbz6w5DTo7NOeofEVrGnm0GYlC\_3kKu7jHzOL4Wcjhv5TXx38yqDcgvWYgwZ64kaQPY8/exec";

**Explanation**

* **googleScriptURL:**
  + **Type:** const char\*
  + **Purpose:** Holds the URL of the deployed Google Apps Script Web App.
  + **Functionality:** Serves as the endpoint for the ESP32 to send HTTP GET requests, logging LED status changes and querying ON time calculations.
* **Usage:** This URL is concatenated with query parameters to form complete HTTP requests that the ESP32 sends based on LED state changes or user queries.

**4. GPIO Definitions**

// -------------------- GPIO Definitions --------------------

#define NUM\_LEDS 4

const uint8\_t LED\_PINS[NUM\_LEDS] = {5, 18, 19, 21};

**Explanation**

* **#define NUM\_LEDS 4:**
  + **Purpose:** Defines a constant NUM\_LEDS representing the total number of LEDs connected to the ESP32.
  + **Value:** 4
  + **Usage:** Facilitates scalable code by allowing easy modification of the number of LEDs without altering multiple code segments.
* **const uint8\_t LED\_PINS[NUM\_LEDS] = {5, 18, 19, 21};:**
  + **Type:** const uint8\_t array
  + **Purpose:** Stores the GPIO pin numbers to which the LEDs are connected.
  + **Values:** Pins 5, 18, 19, and 21 correspond to the four LEDs.
  + **Usage:** Enables the code to reference each LED by its index, promoting cleaner and more maintainable code.

**5. Global Variables for LED States and ON Times**

// -------------------- Global Variables --------------------

bool ledStates[NUM\_LEDS] = {false, false, false, false};

String ledOnTimes[NUM\_LEDS] = {"00:00:00", "00:00:00", "00:00:00", "00:00:00"};

**Explanation**

* **bool ledStates[NUM\_LEDS] = {false, false, false, false};:**
  + **Type:** Boolean array
  + **Purpose:** Tracks the current state of each LED (true for **ON**, false for **OFF**).
  + **Initialization:** All LEDs are initially set to false (i.e., **OFF**).
* **String ledOnTimes[NUM\_LEDS] = {"00:00:00", "00:00:00", "00:00:00", "00:00:00"};:**
  + **Type:** String array
  + **Purpose:** Stores the timestamp (HH:MM:SS) when each LED was last turned **ON**.
  + **Initialization:** All LEDs have their ON times initialized to "00:00:00".
* **Usage:** These arrays facilitate efficient tracking and management of each LED's status and relevant timestamps, enabling accurate logging and time calculations.

**6. Time (NTP) Settings**

// -------------------- Time (NTP) Settings --------------------

const char\* ntpServer = "pool.ntp.org";

const long gmtOffset\_sec = 5 \* 3600; // Adjust for your timezone

const int daylightOffset\_sec = 0;

**Explanation**

* **ntpServer:**
  + **Type:** const char\*
  + **Purpose:** Specifies the Network Time Protocol (NTP) server from which the ESP32 will synchronize its internal clock.
  + **Value:** "pool.ntp.org"
  + **Usage:** Ensures the ESP32 maintains accurate time for timestamping LED status changes.
* **gmtOffset\_sec:**
  + **Type:** const long
  + **Purpose:** Defines the offset in seconds from Greenwich Mean Time (GMT) to the local timezone.
  + **Value:** 5 \* 3600 (i.e., 5 hours)
  + **Adjustment:** Modify this value based on your geographical location to reflect the correct local time.
* **daylightOffset\_sec:**
  + **Type:** const int
  + **Purpose:** Accounts for daylight saving time adjustments.
  + **Value:** 0 (no daylight saving time adjustment)
  + **Usage:** Adjust this value if your timezone observes daylight saving time.
* **Functionality:** These settings ensure that the ESP32's internal clock accurately reflects the local time, which is crucial for precise timestamping of events.

**7. Local CSV File Path (in SPIFFS)**

// ------------- Local CSV File Path (in SPIFFS) -------------

const char\* CSV\_FILE\_PATH = "/led\_log.csv";

**Explanation**

* **CSV\_FILE\_PATH:**
  + **Type:** const char\*
  + **Purpose:** Specifies the path to the local CSV file stored in the ESP32's SPI Flash File System (SPIFFS).
  + **Value:** "/led\_log.csv"
  + **Usage:** Used for reading from and writing to the local CSV file, facilitating offline data logging and retrieval.
* **Functionality:** This file serves as a backup logging mechanism, ensuring that LED status changes are recorded locally even if cloud-based logging to Google Sheets fails due to connectivity issues.

**8. WebServer Setup**

// -------------------- WebServer Setup --------------------

WebServer server(80);

**Explanation**

* **WebServer server(80);:**
  + **Type:** WebServer object
  + **Purpose:** Initializes a web server instance on port 80, the default port for HTTP.
  + **Usage:** Handles incoming HTTP requests from clients (e.g., web browsers), enabling user interaction for LED control and data queries.
* **Functionality:** This web server serves as the backbone for the user interface, providing endpoints for controlling LEDs and querying ON time data.

**9. Function Declarations**

// -------------------- Function Declarations --------------------

void handleRoot();

void handleWebButton();

void handleQuery();

void handleStatus();

void toggleLED(uint8\_t ledIndex, bool state);

void uploadToGoogleSheet(uint8\_t ledIndex, bool state);

// Local CSV helpers

void appendCSVEntry(const String& datetime, uint8\_t ledNumber, bool isOn);

void createCSVIfMissing();

String localCalculateOnTime(uint8\_t ledNumber,

const String& startDate, const String& startTime,

const String& endDate, const String& endTime);

void printEntireCSVtoSerial(const char\* reason);

**Explanation**

* **Function Declarations:**
  + **handleRoot():** Serves the main HTML page for LED control and data queries.
  + **handleWebButton():** Handles HTTP requests to toggle LEDs based on user interactions from the web interface.
  + **handleQuery():** Processes user queries for calculating the total ON time of a specified LED within a given time range.
  + **handleStatus():** Provides the current ON/OFF status of all LEDs in JSON format.
  + **toggleLED(uint8\_t ledIndex, bool state):** Toggles the state of a specific LED and updates the ledStates array.
  + **uploadToGoogleSheet(uint8\_t ledIndex, bool state):** Logs LED status changes to Google Sheets and appends entries to the local CSV file.
* **Local CSV Helper Functions:**
  + **appendCSVEntry(const String& datetime, uint8\_t ledNumber, bool isOn):** Appends a new entry to the local CSV file with the provided datetime, LED number, and status.
  + **createCSVIfMissing():** Checks if the local CSV file exists; if not, creates it with a header row.
  + **localCalculateOnTime(uint8\_t ledNumber, const String& startDate, const String& startTime, const String& endDate, const String& endTime):** Calculates the total ON time of a specified LED within a given time range by processing the local CSV file.
  + **printEntireCSVtoSerial(const char\* reason):** Prints the entire contents of the local CSV file to the serial monitor for debugging purposes.
* **Usage:** These functions modularize the code, enhancing readability and maintainability by separating distinct functionalities into dedicated sections.

**10. Setup Function**

void setup() {

Serial.begin(115200);

delay(100);

// Initialize file system (SPIFFS)

if(!SPIFFS.begin(true)){

Serial.println("[ERROR] SPIFFS Mount Failed!");

} else {

Serial.println("[INFO] SPIFFS mounted successfully.");

// Print the local CSV path

Serial.printf("[INFO] Using local CSV file at: %s\n", CSV\_FILE\_PATH);

createCSVIfMissing();

}

// Initialize LED pins

for (uint8\_t i = 0; i < NUM\_LEDS; i++) {

pinMode(LED\_PINS[i], OUTPUT);

digitalWrite(LED\_PINS[i], LOW);

}

// Connect to Wi-Fi

Serial.println("[INFO] Connecting to WiFi...");

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println("\n[INFO] WiFi connected!");

Serial.print("[INFO] ESP32 IP: ");

Serial.println(WiFi.localIP());

// Initialize time via NTP

configTime(gmtOffset\_sec, daylightOffset\_sec, ntpServer);

Serial.println("[INFO] NTP initialized.");

// Define web server routes

server.on("/", handleRoot);

server.on("/led", handleWebButton);

server.on("/query", handleQuery);

server.on("/status", handleStatus);

// Start web server

server.begin();

Serial.println("[INFO] Web server started.");

}

**Explanation**

The setup() function is executed once when the ESP32 boots up. It initializes various components essential for the system's operation.

1. **Serial Communication Initialization:**

Serial.begin(115200);

delay(100);

* + **Purpose:** Establishes serial communication at a baud rate of 115200 for debugging and monitoring purposes.
  + **Functionality:** Enables the ESP32 to send debug messages to the serial monitor, facilitating real-time monitoring of system operations.

1. **File System Initialization (SPIFFS):**

if(!SPIFFS.begin(true)){

Serial.println("[ERROR] SPIFFS Mount Failed!");

} else {

Serial.println("[INFO] SPIFFS mounted successfully.");

// Print the local CSV path

Serial.printf("[INFO] Using local CSV file at: %s\n", CSV\_FILE\_PATH);

createCSVIfMissing();

}

* + **SPIFFS.begin(true):**
    - **Purpose:** Attempts to mount the SPI Flash File System.
    - **Argument:** true forces a format of SPIFFS if mounting fails.
  + **Error Handling:**
    - If mounting fails, an error message is printed to the serial monitor.
    - If successful, a confirmation message is printed, and the local CSV file's existence is verified or created via createCSVIfMissing().

1. **LED Pins Initialization:**

for (uint8\_t i = 0; i < NUM\_LEDS; i++) {

pinMode(LED\_PINS[i], OUTPUT);

digitalWrite(LED\_PINS[i], LOW);

}

* + **Purpose:** Configures each GPIO pin connected to an LED as an output and ensures all LEDs are initially turned off.
  + **Loop Breakdown:**
    - **pinMode(LED\_PINS[i], OUTPUT):** Sets the pin as an output.
    - **digitalWrite(LED\_PINS[i], LOW):** Ensures the LED is turned off by setting the pin's voltage to low.

1. **Wi-Fi Connection:**

Serial.println("[INFO] Connecting to WiFi...");

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println("\n[INFO] WiFi connected!");

Serial.print("[INFO] ESP32 IP: ");

Serial.println(WiFi.localIP());

* + **Purpose:** Establishes a connection to the specified Wi-Fi network using the provided credentials.
  + **Process:**
    - **WiFi.begin(ssid, password):** Initiates the connection to the Wi-Fi network.
    - **Connection Loop:**
      * Continuously checks the connection status every 500ms.
      * Prints a dot (.) to indicate ongoing connection attempts.
    - **Upon Successful Connection:**
      * Prints a confirmation message along with the ESP32's assigned IP address.

1. **Time Synchronization via NTP:**

configTime(gmtOffset\_sec, daylightOffset\_sec, ntpServer);

Serial.println("[INFO] NTP initialized.");

* + **Purpose:** Synchronizes the ESP32's internal clock with an NTP server to obtain accurate local time.
  + **Functionality:**
    - **configTime(gmtOffset\_sec, daylightOffset\_sec, ntpServer):** Configures the time settings based on the specified GMT offset, daylight saving offset, and NTP server.
  + **Confirmation:** Prints a message indicating successful NTP initialization.

1. **Web Server Routes Definition:**

server.on("/", handleRoot);

server.on("/led", handleWebButton);

server.on("/query", handleQuery);

server.on("/status", handleStatus);

* + **Purpose:** Defines the endpoints that the web server will handle.
  + **Routes:**
    - **"/":** Root route serving the main HTML interface via handleRoot().
    - **"/led":** Endpoint for toggling LEDs via handleWebButton().
    - **"/query":** Endpoint for processing user queries for ON time calculations via handleQuery().
    - **"/status":** Endpoint for retrieving the current status of all LEDs via handleStatus().

1. **Starting the Web Server:**

server.begin();

Serial.println("[INFO] Web server started.");

* + **Purpose:** Activates the web server, making it ready to handle incoming HTTP requests.
  + **Confirmation:** Prints a message indicating that the web server has started successfully.

**11. Main Loop Function**

void loop() {

server.handleClient();

}

**Explanation**

* **loop():**
  + **Purpose:** The main execution loop that runs continuously after the setup() function.
  + **Functionality:** Calls server.handleClient(), which listens for and processes incoming HTTP requests from clients.
  + **Behavior:** Ensures that the web server remains responsive to user interactions, handling LED control commands and data queries in real-time.

**12. LED Toggle Function**

// -----------------------------------------------------------

// toggleLED()

// Toggles a specific LED ON or OFF. Updates ledStates[].

// -----------------------------------------------------------

void toggleLED(uint8\_t ledIndex, bool state) {

if (ledIndex >= NUM\_LEDS) {

Serial.printf("[ERROR] Invalid LED index: %d\n", ledIndex);

return;

}

ledStates[ledIndex] = state;

digitalWrite(LED\_PINS[ledIndex], (state ? HIGH : LOW));

Serial.printf("[EVENT] LED %d turned %s\n", ledIndex + 1, state ? "ON" : "OFF");

}

**Explanation**

* **Function Purpose:**
  + Toggles the state of a specified LED (either turning it **ON** or **OFF**) and updates the ledStates array to reflect the current state.
* **Parameters:**
  + **uint8\_t ledIndex:**
    - **Type:** Unsigned 8-bit integer
    - **Purpose:** Represents the index of the LED to be toggled (ranging from 0 to NUM\_LEDS - 1).
  + **bool state:**
    - **Type:** Boolean
    - **Purpose:** Indicates the desired state of the LED (true for **ON**, false for **OFF**).
* **Function Steps:**
  + **Index Validation:**

if (ledIndex >= NUM\_LEDS) {

Serial.printf("[ERROR] Invalid LED index: %d\n", ledIndex);

return;

* + - **Purpose:** Ensures that the provided ledIndex is within the range of connected LEDs.
    - **Error Handling:** If the index is out of bounds, an error message is printed, and the function exits without performing any action.
  + **State Update:**

ledStates[ledIndex] = state;

digitalWrite(LED\_PINS[ledIndex], (state ? HIGH : LOW));

* + - **ledStates[ledIndex] = state;:**
      * Updates the ledStates array to reflect the new state of the specified LED.
    - **digitalWrite(LED\_PINS[ledIndex], (state ? HIGH : LOW));:**
      * Sets the GPIO pin connected to the LED to HIGH (turns it **ON**) or LOW (turns it **OFF**) based on the state parameter.
  + **Event Logging:**

Serial.printf("[EVENT] LED %d turned %s\n", ledIndex + 1, state ? "ON" : "OFF");

* + - **Purpose:** Logs the LED toggle event to the serial monitor, indicating which LED was toggled and its new state.
    - **Format:** [EVENT] LED X turned ON/OFF
* **Usage:** This function is invoked whenever a user interacts with the web interface to control an LED, ensuring synchronized state changes and accurate logging.

**13. Upload to Google Sheets Function**

// -----------------------------------------------------------

// uploadToGoogleSheet()

// Logs LED toggles to the Google Apps Script Web App

// Also logs to local CSV via appendCSVEntry()

// -----------------------------------------------------------

void uploadToGoogleSheet(uint8\_t ledIndex, bool state) {

// Prepare timestamp

char datetimeBuff[20];

struct tm timeinfo;

if (!getLocalTime(&timeinfo)) {

Serial.println("[ERROR] Failed to get local time (NTP).");

return;

}

strftime(datetimeBuff, sizeof(datetimeBuff), "%Y-%m-%d %H:%M:%S", &timeinfo);

// Log to local CSV

appendCSVEntry(String(datetimeBuff), ledIndex + 1, state);

// Also upload to Google if WiFi connected

if (WiFi.status() != WL\_CONNECTED) {

Serial.println("[ERROR] WiFi not connected. Cannot upload to Google Sheet.");

return;

}

// Encode spaces as %20

char finalDatetime[40];

int j = 0;

for (int i = 0; i < strlen(datetimeBuff); i++) {

if (datetimeBuff[i] == ' ') {

finalDatetime[j++] = '%';

finalDatetime[j++] = '2';

finalDatetime[j++] = '0';

} else {

finalDatetime[j++] = datetimeBuff[i];

}

}

finalDatetime[j] = '\0';

// Construct Google URL

String url = String(googleScriptURL) +

"?lednumber=" + String(ledIndex + 1) +

"&ledstatus=" + (state ? "1" : "0") +

"&datetime=" + finalDatetime;

Serial.println("[INFO] Uploading to Google Sheet: " + url);

HTTPClient http;

http.setTimeout(5000);

http.begin(url);

int httpCode = http.GET();

if (httpCode > 0) {

Serial.printf("[INFO] HTTP Response Code: 200 || (Data Appended to Sheet)\n");

} else {

Serial.printf("[ERROR] HTTP GET failed: %s\n", http.errorToString(httpCode).c\_str());

}

http.end();

// If LED is ON, store the HH:MM:SS part

if (state) {

ledOnTimes[ledIndex] = String(datetimeBuff).substring(11);

}

}

**Explanation**

* **Function Purpose:**
  + Logs each LED status change to both a local CSV file and a Google Sheet via the Google Apps Script Web App.
  + Ensures that LED events are recorded for both offline and cloud-based data analysis.
* **Parameters:**
  + **uint8\_t ledIndex:** Index of the LED being toggled (0 to NUM\_LEDS - 1).
  + **bool state:** New state of the LED (true for **ON**, false for **OFF**).
* **Function Steps:**
  + **Timestamp Preparation:**

char datetimeBuff[20];

struct tm timeinfo;

if (!getLocalTime(&timeinfo)) {

Serial.println("[ERROR] Failed to get local time (NTP).");

return;

}

strftime(datetimeBuff, sizeof(datetimeBuff), "%Y-%m-%d %H:%M:%S", &timeinfo);

* + - **getLocalTime(&timeinfo):**
      * Retrieves the current local time synchronized via NTP.
      * **Error Handling:** If unable to obtain local time, logs an error and exits the function.
    - **strftime(...):**
      * Formats the timeinfo struct into a human-readable datetime string ("YYYY-MM-DD HH:MM:SS").
      * Stores the result in datetimeBuff.
  + **Local CSV Logging:**

appendCSVEntry(String(datetimeBuff), ledIndex + 1, state);

* + - **Purpose:** Calls the appendCSVEntry function to record the LED status change in the local CSV file.
    - **Parameters:**
      * **String(datetimeBuff):** The formatted datetime string.
      * **ledIndex + 1:** Converts the LED index to a human-readable LED number (1-based indexing).
      * **state:** The new state of the LED.
  + **Wi-Fi Connectivity Check:**

if (WiFi.status() != WL\_CONNECTED) {

Serial.println("[ERROR] WiFi not connected. Cannot upload to Google Sheet.");

return;

}

* + - **Purpose:** Ensures that the ESP32 is connected to Wi-Fi before attempting to upload data to Google Sheets.
    - **Error Handling:** If not connected, logs an error message and exits the function.
  + **Datetime Encoding:**

// Encode spaces as %20

char finalDatetime[40];

int j = 0;

for (int i = 0; i < strlen(datetimeBuff); i++) {

if (datetimeBuff[i] == ' ') {

finalDatetime[j++] = '%';

finalDatetime[j++] = '2';

finalDatetime[j++] = '0';

} else {

finalDatetime[j++] = datetimeBuff[i];

}

}

finalDatetime[j] = '\0';

* + - **Purpose:** URL-encodes spaces in the datetime string to %20 to ensure proper transmission via HTTP GET requests.
    - **Process:**
      * Iterates through each character in datetimeBuff.
      * Replaces spaces with %20 and copies other characters as-is.
      * Stores the result in finalDatetime.
  + **Google Sheets URL Construction:**

String url = String(googleScriptURL) +

"?lednumber=" + String(ledIndex + 1) +

"&ledstatus=" + (state ? "1" : "0") +

"&datetime=" + finalDatetime;

* + - **Purpose:** Constructs the complete URL for the HTTP GET request to the Google Apps Script Web App.
    - **Parameters:**
      * **lednumber:** The number of the LED being toggled.
      * **ledstatus:** 1 for **ON**, 0 for **OFF**.
      * **datetime:** The URL-encoded datetime string.
  + **HTTP GET Request to Google Sheets:**

Serial.println("[INFO] Uploading to Google Sheet: " + url);

HTTPClient http;

http.setTimeout(5000);

http.begin(url);

int httpCode = http.GET();

if (httpCode > 0) {

Serial.printf("[INFO] HTTP Response Code: 200 || (Data Appended to Sheet)\n");

} else {

Serial.printf("[ERROR] HTTP GET failed: %s\n", http.errorToString(httpCode).c\_str());

}

http.end();

* + - **Purpose:** Sends an HTTP GET request to log the LED status change in Google Sheets.
    - **Process:**
      * **HTTPClient http;:** Initializes an HTTP client instance.
      * **http.setTimeout(5000);:** Sets a timeout of 5000ms for the request.
      * **http.begin(url);:** Specifies the URL for the HTTP request.
      * **http.GET();:** Sends the HTTP GET request and stores the response code.
    - **Response Handling:**
      * **Success (httpCode > 0):** Logs a success message indicating data was appended to the sheet.
      * **Failure:** Logs an error message with the corresponding error code.
    - **http.end();:** Closes the connection and frees resources.
  + **Storing LED ON Time:**

// If LED is ON, store the HH:MM:SS part

if (state) {

ledOnTimes[ledIndex] = String(datetimeBuff).substring(11);

}

* + - **Purpose:** If the LED is turned **ON**, stores the time (HH:MM:SS) when it was activated.
    - **Functionality:**
      * Extracts the substring starting from the 12th character (index 11) to obtain the time portion of the datetime string.
      * Updates the ledOnTimes array with the extracted time.
* **Error Handling Summary:**
  + **NTP Synchronization Failure:** Logs an error and aborts if local time cannot be obtained.
  + **Wi-Fi Connectivity Issues:** Prevents data upload to Google Sheets if Wi-Fi is disconnected.
  + **HTTP Request Failures:** Logs specific errors if the HTTP GET request to Google Sheets fails.
* **Usage:** This function is invoked whenever an LED is toggled via the web interface, ensuring that each status change is accurately recorded both locally and remotely.

**14. Handling the Root Route (handleRoot)**

* + - **Purpose:** Defines a multi-line raw string (rawliteral) containing the HTML structure, CSS styles, and JavaScript functionalities for the web interface.
    - **Components:**
      * **HTML Structure:** Defines the layout of the web page, including headers, forms, buttons, and display areas.
      * **CSS Styles:** Utilizes Bootstrap for styling and custom CSS for visual enhancements, such as background gradients and responsive design.
      * **JavaScript Functions:**
        + **toggleLED(ledNumber, state):** Sends HTTP requests to toggle LEDs based on user interactions.
        + **updateLEDStatus():** Fetches the current status of all LEDs and updates the web interface accordingly.
        + **Form Submission Handling:** Processes user inputs for querying LED ON times, sends HTTP requests, and displays the results.
  + **Dynamic LED Status Embedding:**

<span id="led1Status">)rawliteral" +

String(ledStates[0] ? "ON" : "OFF") + R"rawliteral(</span><br>

...

* + - **Purpose:** Dynamically inserts the current status ("ON" or "OFF") of each LED into the HTML content.
    - **Process:**
      * Uses the ledStates array to determine each LED's status.
      * Embeds the status within <span> elements for real-time display.
  + **Web Server Response:**

server.send(200, "text/html", html);

* + - **Purpose:** Sends the constructed HTML content as an HTTP response with a status code of 200 (OK) and a content type of "text/html".
    - **Functionality:** Delivers the web interface to clients (e.g., users accessing the ESP32's IP address via a web browser).
* **Usage:** This function is invoked when a client accesses the root URL ("/") of the ESP32's web server, presenting an interactive interface for LED control and data querying.

**15. Handling LED Toggle Requests (handleWebButton)**

// -----------------------------------------------------------

// handleWebButton()

// /led?lednumber=X&state=on/off

// Toggles LED, logs to CSV + Google

// -----------------------------------------------------------

void handleWebButton() {

if (server.hasArg("lednumber") && server.hasArg("state")) {

uint8\_t ledNumber = server.arg("lednumber").toInt();

String st = server.arg("state");

bool newState = (st == "on");

if (ledNumber < 1 || ledNumber > NUM\_LEDS) {

server.send(400, "text/plain", "Invalid LED number.");

return;

}

toggleLED(ledNumber - 1, newState);

uploadToGoogleSheet(ledNumber - 1, newState);

server.send(204, "text/plain", "");

} else {

server.send(400, "text/plain", "Missing 'lednumber' or 'state'.");

}

}

**Explanation**

* **Function Purpose:**
  + Handles HTTP GET requests to the /led endpoint, allowing users to toggle specific LEDs on or off via the web interface.
  + Ensures that each toggle action is logged both locally and remotely.
* **Function Steps:**
  + **Parameter Validation:**

if (server.hasArg("lednumber") && server.hasArg("state")) {

...

} else {

server.send(400, "text/plain", "Missing 'lednumber' or 'state'.");

}

* + - **Purpose:** Checks if the incoming HTTP request contains both the lednumber and state parameters.
    - **Error Handling:** If either parameter is missing, responds with a 400
  + **Parameter Extraction and Conversion:**

uint8\_t ledNumber = server.arg("lednumber").toInt();

String st = server.arg("state");

bool newState = (st == "on");

* + - **ledNumber:** Converts the lednumber parameter from a string to an integer (uint8\_t), representing the LED number.
    - **st:** Retrieves the state parameter as a string.
    - **newState:** Determines the desired LED state (true for **ON**, false for **OFF**) based on the state parameter.
  + **LED Number Validation:**

if (ledNumber < 1 || ledNumber > NUM\_LEDS) {

server.send(400, "text/plain", "Invalid LED number.");

return;

}

* + - **Purpose:** Ensures that the ledNumber is within the valid range (1 to NUM\_LEDS).
    - **Error Handling:** If the LED number is invalid, responds with a 400 status code and an error message.
  + **LED Toggling and Logging:**

toggleLED(ledNumber - 1, newState);

uploadToGoogleSheet(ledNumber - 1, newState);

* + - **toggleLED(ledNumber - 1, newState):** Toggles the specified LED's state. The LED index is adjusted to zero-based indexing.
    - **uploadToGoogleSheet(ledNumber - 1, newState):** Logs the status change to both the local CSV file and Google Sheets.
  + **HTTP Response:**

server.send(204, "text/plain", "");

* + - **Purpose:** Sends an HTTP response with a status code of 204 (No Content), indicating that the request was successfully processed without returning any content.
    - **Functionality:** Informs the client that the LED state change was successful.
* **Usage:** This function is invoked when a user clicks the **ON** or **OFF** buttons for any LED on the web interface, ensuring that the corresponding LED is toggled and the action is appropriately logged.

**16. Handling Status Requests (handleStatus)**

// -----------------------------------------------------------

// handleStatus()

// Returns current ON/OFF states of all LEDs as JSON

// e.g., {"LED1":"ON","LED2":"OFF","LED3":"OFF","LED4":"ON"}

// -----------------------------------------------------------

void handleStatus() {

String statusJson = "{";

for (uint8\_t i = 0; i < NUM\_LEDS; i++) {

statusJson += "\"LED" + String(i+1) + "\":\"" + (ledStates[i] ? "ON" : "OFF") + "\"";

if (i < NUM\_LEDS - 1) statusJson += ",";

}

statusJson += "}";

server.send(200, "application/json", statusJson);

}

**Explanation**

* **Function Purpose:**
  + Provides the current **ON/OFF** status of all connected LEDs in JSON format.
  + Facilitates real-time status updates on the web interface.
* **Function Steps:**
  + **JSON Construction:**

String statusJson = "{";

for (uint8\_t i = 0; i < NUM\_LEDS; i++) {

statusJson += "\"LED" + String(i+1) + "\":\"" + (ledStates[i] ? "ON" : "OFF") + "\"";

if (i < NUM\_LEDS - 1) statusJson += ",";

}

statusJson += "}";

* + - **Process:**
      * Initializes a JSON string with an opening brace {.
      * Iterates through each LED, appending its status to the JSON string in the format "LEDX":"ON/OFF".
      * Ensures that commas separate each LED status except for the last one.
      * Closes the JSON string with a closing brace }.
  + **HTTP Response:**

server.send(200, "application/json", statusJson);

* + - **Purpose:** Sends the constructed JSON string as an HTTP response with a status code of 200 (OK) and a content type of "application/json".
    - **Functionality:** Enables the web interface to fetch and display the current status of all LEDs dynamically.
* **Usage:** This function is invoked when the web interface requests the /status endpoint, ensuring that the displayed LED statuses are always up-to-date.

**17. Handling Query Requests (handleQuery)**

// -----------------------------------------------------------

// handleQuery()

// -----------------------------------------------------------

void handleQuery() {

if (!server.hasArg("lednumber") || !server.hasArg("startDate") ||

!server.hasArg("startTime") || !server.hasArg("endDate") ||

!server.hasArg("endTime")) {

server.send(400, "text/plain", "Missing required parameters");

return;

}

String ledNumberStr = server.arg("lednumber");

String startDate = server.arg("startDate");

String startTime = server.arg("startTime");

String endDate = server.arg("endDate");

String endTime = server.arg("endTime");

uint8\_t ledNumber = ledNumberStr.toInt();

if (ledNumber < 1 || ledNumber > NUM\_LEDS) {

server.send(400, "text/plain", "Invalid LED number.");

return;

}

// Debug prints

Serial.println("[QUERY] Time-Range calculation requested");

Serial.printf("[QUERY] LED = %d, Range: %s %s -> %s %s\n",

ledNumber, startDate.c\_str(), startTime.c\_str(), endDate.c\_str(), endTime.c\_str());

// Print entire CSV for debugging

// printEntireCSVtoSerial("handleQuery() called: printing CSV content...");

// Build the Google Link, but do NOT fetch it

String googleURL = String(googleScriptURL) +

"?lednumber=" + ledNumber +

"&startDate=" + startDate +

"&startTime=" + startTime +

"&endDate=" + endDate +

"&endTime=" + endTime;

// Local calculation

String localResult = localCalculateOnTime(ledNumber, startDate, startTime, endDate, endTime);

// parse localResult (JSON) => hours, minutes, seconds

int hIdx = localResult.indexOf("\"hours\":");

int mIdx = localResult.indexOf("\"minutes\":");

int sIdx = localResult.indexOf("\"seconds\":");

long hours = 0;

long minutes = 0;

long seconds = 0;

if(hIdx!=-1 && mIdx!=-1 && sIdx!=-1) {

hours = localResult.substring(hIdx+8, localResult.indexOf(',', hIdx)).toInt();

minutes = localResult.substring(mIdx+10, localResult.indexOf(',', mIdx)).toInt();

int secEnd = localResult.indexOf('}', sIdx);

seconds = localResult.substring(sIdx+10, secEnd).toInt();

}

// Build final JSON

// e.g.: {"ledNumber":2, "queriedURL":"...", "hours":1, "minutes":30, "seconds":10}

String finalJSON = "{";

finalJSON += "\"ledNumber\":" + String(ledNumber) + ",";

finalJSON += "\"queriedURL\":\"" + googleURL + "\",";

finalJSON += "\"hours\":" + String(hours) + ",";

finalJSON += "\"minutes\":" + String(minutes) + ",";

finalJSON += "\"seconds\":" + String(seconds);

finalJSON += "}";

// Send to user

server.send(200, "application/json", finalJSON);

}

**Explanation**

* **Function Purpose:**
  + Processes user queries to calculate the total **ON** time of a specified LED within a defined time range.
  + Combines both local CSV calculations and provides a URL to query the Google Apps Script Web App for cloud-based calculations.
* **Function Steps:**
  + **Parameter Validation:**

if (!server.hasArg("lednumber") || !server.hasArg("startDate") ||

!server.hasArg("startTime") || !server.hasArg("endDate") ||

!server.hasArg("endTime")) {

server.send(400, "text/plain", "Missing required parameters");

return;

}

* + - **Purpose:** Ensures that all necessary parameters (lednumber, startDate, startTime, endDate, endTime) are present in the HTTP request.
    - **Error Handling:** If any parameter is missing, responds with a 400 status code and an error message.
  + **Parameter Extraction and Conversion:**

String ledNumberStr = server.arg("lednumber");

String startDate = server.arg("startDate");

String startTime = server.arg("startTime");

String endDate = server.arg("endDate");

String endTime = server.arg("endTime");

uint8\_t ledNumber = ledNumberStr.toInt();

if (ledNumber < 1 || ledNumber > NUM\_LEDS) {

server.send(400, "text/plain", "Invalid LED number.");

return;

}

* + - **Extraction:**
      * Retrieves each parameter from the HTTP request.
    - **Conversion:**
      * Converts lednumber from a string to an integer (uint8\_t).
    - **Validation:**
      * Ensures that the ledNumber is within the valid range (1 to NUM\_LEDS).
      * **Error Handling:** If invalid, responds with a 400 status code and an error message.
  + **Debug Logging:**

Serial.println("[QUERY] Time-Range calculation requested");

Serial.printf("[QUERY] LED = %d, Range: %s %s -> %s %s\n",

ledNumber, startDate.c\_str(), startTime.c\_str(), endDate.c\_str(), endTime.c\_str());

* + - **Purpose:** Logs the details of the query to the serial monitor for debugging and monitoring purposes.
  + **Google Sheets URL Construction:**

String googleURL = String(googleScriptURL) +

"?lednumber=" + ledNumber +

"&startDate=" + startDate +

"&startTime=" + startTime +

"&endDate=" + endDate +

"&endTime=" + endTime;

* + - **Purpose:** Constructs the URL for querying the Google Apps Script Web App based on the user-provided parameters.
    - **Parameters:**
      * **lednumber:** The number of the LED being queried.
      * **startDate & startTime:** The beginning of the time range.
      * **endDate & endTime:** The end of the time range.
  + **Local Calculation:**

String localResult = localCalculateOnTime(ledNumber, startDate, startTime, endDate, endTime);

* + - **Purpose:** Calls the localCalculateOnTime function to compute the total **ON** time of the specified LED within the given time range by processing the local CSV file.
    - **Result:** Receives a JSON-formatted string containing hours, minutes, and seconds.
  + **Parsing Local Calculation Result:**

// parse localResult (JSON) => hours, minutes, seconds

int hIdx = localResult.indexOf("\"hours\":");

int mIdx = localResult.indexOf("\"minutes\":");

int sIdx = localResult.indexOf("\"seconds\":");

long hours = 0;

long minutes = 0;

long seconds = 0;

if(hIdx!=-1 && mIdx!=-1 && sIdx!=-1) {

hours = localResult.substring(hIdx+8, localResult.indexOf(',', hIdx)).toInt();

minutes = localResult.substring(mIdx+10, localResult.indexOf(',', mIdx)).toInt();

int secEnd = localResult.indexOf('}', sIdx);

seconds = localResult.substring(sIdx+10, secEnd).toInt();

}

* + - **Purpose:** Extracts the hours, minutes, and seconds values from the JSON-formatted localResult string.
    - **Process:**
      * Identifies the positions of "hours":, "minutes":, and "seconds": within the string.
      * Extracts the numerical values following each key.
      * Converts the extracted substrings to integers.
  + **Final JSON Construction:**

// Build final JSON

// e.g.: {"ledNumber":2, "queriedURL":"...", "hours":1, "minutes":30, "seconds":10}

String finalJSON = "{";

finalJSON += "\"ledNumber\":" + String(ledNumber) + ",";

finalJSON += "\"queriedURL\":\"" + googleURL + "\",";

finalJSON += "\"hours\":" + String(hours) + ",";

finalJSON += "\"minutes\":" + String(minutes) + ",";

finalJSON += "\"seconds\":" + String(seconds);

finalJSON += "}";

* + - **Purpose:** Constructs a comprehensive JSON response containing the LED number, the Google Sheets query URL, and the total **ON** time in hours, minutes, and seconds.
    - **Format:**

json

{

"ledNumber": X,

"queriedURL": "https://...",

"hours": Y,

"minutes": Z,

"seconds": W

}

* + **Sending the Response:**

// Send to user

server.send(200, "application/json", finalJSON);

* + - **Purpose:** Sends the constructed JSON string as an HTTP response with a status code of 200 (OK) and a content type of "application/json".
    - **Functionality:** Allows the web interface to display the calculated **ON** time and provides a link to the Google Sheets query for further verification.
* **Error Handling Summary:**
  + **Missing Parameters:** Returns a 400 status code if any required parameters are absent.
  + **Invalid LED Number:** Returns a 400 status code if the LED number is out of bounds.
  + **Malformed Local Result:** Defaults hours, minutes, and seconds to 0 if parsing fails.
* **Usage:** This function is invoked when a user submits a query for an LED's **ON** time through the web interface, ensuring accurate and comprehensive responses.

**18. Creating CSV File if Missing (createCSVIfMissing)**

// -----------------------------------------------------------

// createCSVIfMissing()

// If /led\_log.csv doesn't exist, create it with a header row

// -----------------------------------------------------------

void createCSVIfMissing() {

if(!SPIFFS.exists(CSV\_FILE\_PATH)) {

File f = SPIFFS.open(CSV\_FILE\_PATH, FILE\_WRITE);

if(f) {

f.println("Datetime,LED,Status"); // optional header

f.close();

Serial.println("[INFO] Created new /led\_log.csv with header.");

} else {

Serial.println("[ERROR] Failed to create /led\_log.csv");

}

} else {

Serial.println("[INFO] /led\_log.csv already exists.");

}

}

**Explanation**

* **Function Purpose:**
  + Ensures that the local CSV file (/led\_log.csv) exists within the ESP32's SPIFFS. If it doesn't, the function creates the file and adds a header row.

**19. Appending Entries to the Local CSV (appendCSVEntry)**

// -----------------------------------------------------------

// appendCSVEntry()

// Appends a line to /led\_log.csv

// e.g. "2023-09-06 12:34:56,1,ON"

// -----------------------------------------------------------

void appendCSVEntry(const String& datetime, uint8\_t ledNumber, bool isOn) {

File f = SPIFFS.open(CSV\_FILE\_PATH, FILE\_APPEND);

if(!f) {

Serial.println("[ERROR] Cannot open /led\_log.csv for append!");

return;

}

String statusStr = isOn ? "ON" : "OFF";

String line = datetime + "," + String(ledNumber) + "," + statusStr;

f.println(line);

f.close();

//Serial.println("[INFO] Appended local CSV: " + line);

}

**Explanation**

* **Function Purpose:**
  + Appends a new entry to the local CSV file, logging the datetime, LED number, and status.
* **Parameters:**
  + **const String& datetime:** The timestamp of the LED status change ("YYYY-MM-DD HH:MM:SS").
  + **uint8\_t ledNumber:** The number of the LED being logged.
  + **bool isOn:** Indicates whether the LED is turned **ON** (true) or **OFF** (false).

**20. Printing the Entire CSV to Serial for Debugging (printEntireCSVtoSerial)**

// -----------------------------------------------------------

// printEntireCSVtoSerial()

// Utility to print the entire CSV file to Serial

// for debugging each time a user queries

// -----------------------------------------------------------

void printEntireCSVtoSerial(const char\* reason) {

Serial.println("[DEBUG] " + String(reason));

File f = SPIFFS.open(CSV\_FILE\_PATH, FILE\_READ);

if(!f) {

Serial.println("[ERROR] Could not open CSV for reading!");

return;

}

Serial.println("--- CSV BEGIN ---");

while(f.available()) {

String line = f.readStringUntil('\n');

if(line.length() == 0) break;

Serial.println(line);

}

f.close();

Serial.println("--- CSV END ---");

}

**Explanation**

* **Function Purpose:**
  + Provides a utility to print the entire contents of the local CSV file to the serial monitor, aiding in debugging and verification of logged data.
* **Parameters:**
  + **const char\* reason:** A descriptive reason for invoking the function, enhancing clarity in the debug logs.

**21. Local Calculation of LED ON Time (CalculateOnTime)**

**Explanation**

* **Function Purpose:**
  + Calculates the total time a specified LED was **ON** within a user-defined time range by processing the Sheet.
  + Returns the result in JSON format, detailing the hours, minutes, and seconds the LED remained **ON**.

**22. Final Thoughts on Code Structure and Functionality**

The ESP32 Arduino IDE code for this Project is meticulously structured to ensure robustness, scalability, and ease of understanding. By compartmentalizing functionalities into dedicated functions and leveraging preprocessor directives, the code achieves a high level of modularity. This approach not only simplifies debugging and maintenance but also facilitates future enhancements, such as adding more LEDs or integrating additional features.

**Key Highlights:**

* **Modular Design:** Each function is responsible for a specific task, promoting code reusability and clarity.
* **Robust Error Handling:** Comprehensive checks ensure that the system can gracefully handle unexpected scenarios, such as connectivity issues or malformed data.
* **Dual Logging Mechanism:** By recording data both locally and remotely, the system ensures data integrity and availability even in cases of intermittent internet connectivity.
* **Interactive Web Interface:** The intuitive web interface empowers users to control LEDs and query data effortlessly, enhancing user experience.
* **Scalability:** The use of arrays and defined constants allows for easy expansion of the system to accommodate more LEDs or additional functionalities without significant code modifications.

# Screenshots of Output

**1. Web Interface for LED Control and Queries**

**For PC and Mobile**

*A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated*

A screenshot of a phone

Description automatically generated

A screenshot of a device

Description automatically generated

A screenshot of a computer

Description automatically generated

**Summary of Error Handling**

Ensuring the system's resilience and reliability is paramount. The following outlines the comprehensive error handling mechanisms integrated into the ESP32 Arduino IDE code:

1. **Wi-Fi Connectivity Issues:**
   * **Detection:** Continuously monitors the Wi-Fi connection status.
   * **Action:** Prevents attempts to upload data to Google Sheets if disconnected, ensuring no erroneous data transmissions.
2. **NTP Synchronization Failures:**
   * **Detection:** Checks if the ESP32 can successfully retrieve the current time from the NTP server.
   * **Action:** Logs an error message and aborts timestamp-dependent operations if synchronization fails.
3. **HTTP Request Failures:**
   * **Detection:** Monitors the success of HTTP GET requests to Google Sheets.
   * **Action:** Logs specific error messages detailing the nature of the failure, aiding in troubleshooting connectivity or script issues.
4. **Parameter Validation:**
   * **Detection:** Validates the presence and correctness of required parameters in HTTP requests.
   * **Action:** Returns appropriate HTTP status codes and error messages for missing or invalid parameters, ensuring data integrity and preventing unintended operations.
5. **Boundary Conditions:**
   * **Detection:** Checks for invalid LED indices or time ranges.
   * **Action:** Ensures that the system operates within defined parameters, maintaining stability and reliability.

* **Logging Mechanism:** Utilizes serial logging extensively to provide real-time feedback and detailed information about the system's operations and any encountered issues.

**Methods from Arduino and SPIFFS Libraries**

Understanding the methods utilized from various libraries is essential for grasping the system's functionality.

**1. WiFi Class (WiFi.h Library)**

* **WiFi.begin(ssid, password):**
  + **Functionality:** Initiates a connection to the specified Wi-Fi network using the provided SSID and password.
  + **Arguments:**
    - ssid (const char\*): The name of the Wi-Fi network.
    - password (const char\*): The password for the Wi-Fi network.
  + **Return Type:** void
* **WiFi.status():**
  + **Functionality:** Retrieves the current connection status of the ESP32 to the Wi-Fi network.
  + **Arguments:** None
  + **Return Type:** wl\_status\_t (e.g., WL\_CONNECTED, WL\_DISCONNECTED)
* **WiFi.localIP():**
  + **Functionality:** Obtains the IP address assigned to the ESP32 by the connected Wi-Fi network.
  + **Arguments:** None
  + **Return Type:** IPAddress object

**2. HTTPClient Class (HTTPClient.h Library)**

* **HTTPClient http;:**
  + **Functionality:** Creates an instance of the HTTPClient class to perform HTTP operations.
  + **Usage:** Enables the ESP32 to send HTTP requests and handle responses.
* **http.setTimeout(timeout):**
  + **Functionality:** Sets a timeout duration for HTTP requests.
  + **Arguments:**
    - timeout (unsigned int): The timeout duration in milliseconds.
  + **Return Type:** void
* **http.begin(url):**
  + **Functionality:** Initializes the HTTP request with the specified URL.
  + **Arguments:**
    - url (String): The endpoint to send the HTTP request to.
  + **Return Type:** void
* **http.GET():**
  + **Functionality:** Sends an HTTP GET request to the initialized URL.
  + **Arguments:** None
  + **Return Type:** int representing the HTTP response code (e.g., 200 for success)
* **http.errorToString(code):**
  + **Functionality:** Converts an HTTP error code to a human-readable string.
  + **Arguments:**
    - code (int): The HTTP error code.
  + **Return Type:** String
* **http.end():**
  + **Functionality:** Closes the HTTP connection and frees associated resources.
  + **Arguments:** None
  + **Return Type:** void

**3. WebServer Class (WebServer.h Library)**

* **WebServer server(port):**
  + **Functionality:** Creates a web server instance listening on the specified port.
  + **Arguments:**
    - port (uint16\_t): The port number to listen on (e.g., 80 for HTTP).
  + **Return Type:** WebServer object
* **server.on(path, handler):**
  + **Functionality:** Defines a route and associates it with a handler function.
  + **Arguments:**
    - path (const char\*): The URL path (e.g., "/led").
    - handler (void function): The function to handle requests to the specified path.
  + **Return Type:** void
* **server.handleClient():**
  + **Functionality:** Processes incoming client requests and invokes the appropriate handler functions.
  + **Arguments:** None
  + **Return Type:** void
* **server.hasArg(arg):**
  + **Functionality:** Checks if a specific argument is present in the HTTP request.
  + **Arguments:**
    - arg (const char\*): The name of the argument to check.
  + **Return Type:** bool
* **server.arg(arg):**
  + **Functionality:** Retrieves the value of a specific argument from the HTTP request.
  + **Arguments:**
    - arg (const char\*): The name of the argument to retrieve.
  + **Return Type:** String
* **server.send(statusCode, contentType, content):**
  + **Functionality:** Sends an HTTP response to the client.
  + **Arguments:**
    - statusCode (int): The HTTP status code (e.g., 200, 400, 204).
    - contentType (const char\*): The MIME type of the response (e.g., "text/plain", "application/json").
    - content (String): The body of the response.
  + **Return Type:** void

**5. Arduino String Class**

* **String(String) Constructor:**
  + **Functionality:** Initializes a new String object.
  + **Arguments:**
    - String (const char\*): The initial string content.
  + **Return Type:** String object
* **String.toInt():**
  + **Functionality:** Converts the String to an integer.
  + **Arguments:** None
  + **Return Type:** long
* **String.substring(start, end):**
  + **Functionality:** Extracts a substring from the String starting at start index and ending before end index.
  + **Arguments:**
    - start (unsigned int): The starting index.
    - end (unsigned int): The ending index.
  + **Return Type:** String
* **String.trim():**
  + **Functionality:** Removes leading and trailing whitespace from the String.
  + **Arguments:** None
  + **Return Type:** void

**Conclusion**

The project successfully extends the functionalities explored in Lab 10 by introducing a comprehensive system that not only logs LED statuses but also calculates the total time each LED remains **ON** within user-defined time frames. By leveraging the power of the ESP32 microcontroller, and Google Apps Script for cloud-based data processing, the system showcases a practical application of **Internet of Things (IoT)** principles.

The meticulous breakdown of the Arduino code facilitates a deep understanding of each component's role and interconnections, making the system both robust and scalable. Comprehensive error handling ensures that the system remains reliable under various conditions, while the interactive web interface enhances user engagement and experience.

This project lays a solid foundation for developing more intricate embedded systems that interact seamlessly with cloud services and user interfaces, paving the way for future advancements in home automation and energy management.

**References**

1. **Arduino Documentation:** <https://www.arduino.cc/reference/en/>
2. **Google Apps Script Documentation:** <https://developers.google.com/apps-script>
3. **ESP32 Official Resources:** <https://www.espressif.com/en/products/socs/esp32>
4. **ArduinoJson Library:** <https://arduinojson.org/>
5. **SPIFFS File System Documentation:** <https://github.com/espressif/arduino-esp32/tree/master/libraries/SPIFFS>
6. **Bootstrap CSS Framework:** <https://getbootstrap.com/>
7. **NTP Server Pool:** <http://www.pool.ntp.org/>