Project Documentation

Data acquisition

We utilized a library available on GitHub, developed by Adafruit, as it offered convenient functions for directly accessing lux values. Additionally, the library provided useful example codes that facilitated our data acquisition testing.

Initially, our challenge involved determining which of the three values we required. The library presented us with ALS (Ambient Light Sensor), White, and Lux. The ALS solely measured the spectrum ranging from yellow to green light, while White encompassed measurements from blue to infrared light. Through our investigation, we discovered that Lux was calculated by multiplying ALS readings with a constant. Considering our objective, we made the decision to exclude White measurements, as they had the potential to distort our values by capturing infrared waves emitted by individuals or buildings. Moreover, the Lux value represented the calibrated ALS reading, further solidifying our choice to focus solely on measuring Lux.

By leveraging the Adafruit library's functionality and reference codes, we successfully obtained the desired lux measurements, enhancing the accuracy and reliability of our data acquisition process.

Data structure

The main data structure chosen for our implementation was the array *table_jsons*. We opted to directly convert the time and lux measurements into a char array, resulting in each data point being represented as [time, lux]. This particular format was selected because we required the data in the format [['X', 'Y'], [1, 1], [2, 4], [3, 2]] for plotting purposes. Therefore, *table_jsons* would store these char arrays, while the index_jsons would indicate the appropriate position to insert the next char array.

In the event that the array became full, we would reset *index_jsons* to zero and continue adding values from the beginning. This approach allowed us to maintain only the most recent data points, discarding the oldest one by freeing the corresponding memory and replacing it with the new value in the table.

This data structure offers several advantages. Firstly, it enables us to store a significant amount of data while retaining only the latest data points. Furthermore, we benefit from direct access to all the data through the index, which would not be possible with linked-list or double linked-list structures. Additionally, the already converted format of the data provides the advantage of quick and straightforward integration with the API, as it can be pushed directly.

However, it is important to note that this approach has a limitation. The time and value data are already formatted in a specific manner, which means that if one intends to use the data for other purposes, they would need to create separate arrays for time and value, following the desired format.

API

We utilized the Wi-Fi API in the access point mode to establish a connection. The process involved creating a hotspot with a designated name (SSID) and password using a phone. By doing so, the API initiated a web server on the hotspot. This allowed a computer or phone to access a local website hosted on the hotspot.

Our objective was to display the data in the form of a line plot on the website. In our exploration of available resources, we discovered that Google Charts provided examples of HTML code that facilitated easy implementation. Taking inspiration from these examples, we made necessary modifications and adjusted the data structure to conform to the HTML data format.

By adopting this approach, we successfully transformed the data into a format compatible with the line plot on the web page. This enabled users to visualize the data in a clear and organized manner, enhancing the overall user experience.

Architecture & Structure of files

The code is organized into several files. The main file, denoted by the .ino extension, serves as the executed file and is responsible for calling all the necessary functions from the other files:

1. Adafruit_VEML7700.cpp: This file handles the communication with the VEML7700 sensor. It contains essential functions that enable the retrieval of lux values. By interfacing with the sensor, this file plays a crucial role in acquiring accurate data.
2. Datastructure.cpp: This file focuses on the implementation of various array-related operations. It encompasses functions responsible for creating, destroying, printing, and adding elements to the array. These operations facilitate the manipulation and management of the data structure.
3. client.cpp: Within this file, you will find functions associated with establishing, terminating, and reattempting network connections. It also handles the transmission of data to the API. This file plays a pivotal role in ensuring seamless communication between the device and the server.
4. utility.cpp: The utility.cpp file comprises functions that, while not vital to the core functionality of the code, provide valuable auxiliary features. These functions assist with printing information and set constant values that support the code's overall execution.

Furthermore, each of the .cpp files is accompanied by a corresponding .h file. These header files declare the function prototypes, providing essential information and enabling proper utilization of the functions defined in their respective .cpp files.

Additionally, there is a folder named "Documents" that contains both a presentation and four individual reports.

Sources:

https://github.com/adafruit/Adafruit_VEML7700/blob/master/Adafruit_VEML7700.cpp

https://docs.espressif.com/projects/arduino-esp32/en/latest/api/wifi.html

https://developers.google.com/chart/interactive/docs/gallery/linechart?hl=de