Home Environment Dashboard Documentation

Introduction

Welcome to the documentation for the Inkplate Home Environment Dashboard! I created this comprehensive guide to help makers and electronics enthusiasts build their own smart home monitoring system. After receiving numerous requests from my YouTube community about building IoT projects, I decided to document this project in detail, breaking down every component and explaining the code structure thoroughly. This documentation reflects my commitment to helping others learn and build confidently.

You can find the latest version of this code, along with regular updates and improvements, on my website at educ8s.tv. I regularly update the code based on community feedback and new feature requests, so make sure to check there for the most recent version.

Project Overview

The **Home Environment Dashboard** is a smart monitoring system that uses an **Inkplate2 Board**, a **BME280 sensor**, and an **E-Ink display** to track **temperature, humidity, and barometric pressure**. The data is displayed both on the **E-Ink screen** and through a **local web dashboard** accessible from any device on the same network. This project ensures **privacy and efficiency** by keeping all data local and optimizing power consumption.

Features

- **Object-Oriented Modular Design**, allowing easy modification and scalability. *(If you're not comfortable with OOP design, check out my detailed course here: **Object-Oriented Programming Made Easy.**)*
- **Real-time environmental monitoring** using a **BME280 sensor**.
- **E-Paper display** for a **classic thermometer look**.
- **Web dashboard** for **real-time data visualization** and **historical tracking**.
- **Power efficiency**, updating the display only when necessary.
- **Local data storage**, ensuring **privacy** with no external cloud dependencies.

Hardware Requirements

- **Inkplate2 Board:** https://educ8s.tv/part/Inkplate2
- **BME280 Sensor:** https://educ8s.tv/part/BME280
- **MiniUSB Breakout Board:** https://educ8s.tv/part/MiniUSBBreakout

Software Requirements

- **Arduino IDE** with ESP32 board support
- **Required Libraries:**
  - `ArduinoJSON` (to create JSON data for the webserver)
  - `Inkplate` (for the board)
  - `Adafruit_BME280` (for the sensor)

You can install these libraries directly from the Arduino IDE, from the Library Manager.

How the Code Works

Code Flow & Execution

1. Initialization (`setup()` Function in `ESP32_HomeDashboard.ino`)

The execution begins in the `setup()` function:

- Serial communication starts (`Serial.begin(115200)`).
- The sensor is initialized (`sensor.begin()`).
- The E-Paper display is initialized (`epaper.init()`).
- WiFi connection is established (`connectToWiFi()`).
- The web server is set up (`setupServer()`).
- Initial sensor readings are taken and displayed.

2. Main Loop (`loop()` Function in `ESP32_HomeDashboard.ino`)

The `loop()` function runs continuously:

- Handles client requests (`server.handleClient()`).
- Reads temperature every 5 minutes (using a timer check with `millis()`).

- Determines if the display needs an update by comparing the new temperature with the last recorded value (`needsRedraw()`).
- If a significant change is detected, the **E-Paper display updates** (`epaper.drawTemperature(currentTemperature)`).
- Data is logged every hour to maintain a history for trend analysis.

Code Structure

Global Variables

- **Wi-Fi Credentials:**
  You enter your credentials here:

```
const char* ssid = "your_network_name";
const char* password = "your_network_password";
```

- **Metric System Setting:**

```
const bool metric = true; // Set to false for Fahrenheit readings
```

- **Instances of Classes:**
  - `Sensor sensor(metric);` - Handles temperature readings.
  - `WebServer server(80);` - Sets up an HTTP server on port 80.
  - `SensorData sensorData(48, sensor);` - Manages sensor data updates.
  - `Display epaper;` - Manages the E-Ink display.
- **Timers:**

```
const unsigned long tempUpdateInterval = 300000; // 5 minutes
```

`setup()` Function

- Initializes serial communication.
- Starts the sensor and E-Ink display.
- Connects to Wi-Fi.
- Sets up the web server.

`loop()` Function

- Handles incoming web requests.

- Updates temperature readings at regular intervals.
- Updates the E-Ink display **only when a significant temperature change occurs**.

`connectToWiFi()` Function

Handles Wi-Fi connection using SSID and password, ensuring stability in network connection. The function **waits until a connection is established** and prints the **IP address**.

`setupServer()` Function

- Defines API endpoints to provide temperature data to clients.
- Serves an HTML page stored in `index_html.h` .

Server Endpoints

Endpoint	Method	Description
`/`	GET	Serves the web interface page
`/sensor`	GET	Returns current temperature, humidity, and pressure data in JSON format
`/history`	GET	Returns historical sensor data in JSON format

Sensor Class

The `Sensor` class is responsible for interfacing with the **BME280 environmental sensor**, providing real-time **temperature, humidity, and pressure readings**. It supports both metric and imperial units and ensures accurate environmental data retrieval.

Class Definition

```
class Sensor {
public:
    Sensor(bool useMetric = true); // Constructor accepts metric flag
    bool begin(); // Initializes the sensor hardware
    float getTemperature(); // Returns temperature in the selected unit
    float getTemperatureC(); // Returns temperature in Celsius
    float getHumidity(); // Returns humidity percentage
    float getPressure(); // Returns atmospheric pressure in hPa

private:
    Adafruit_BME280 bme; // Object handling BME280 sensor operations
    bool metric; // Flag to determine whether to return values in metric units
};
```

How It Works

- **Initialization (`begin()`)**: Initializes the BME280 sensor and verifies its availability.
- **Temperature Reading (`getTemperature()`)**: Returns the current temperature in either Celsius or Fahrenheit, depending on the `metric` flag.
- **Direct Celsius Reading (`getTemperatureC()`)**: Provides the temperature in Celsius regardless of the metric setting.
- **Humidity Measurement (`getHumidity()`)**: Retrieves the current relative humidity percentage.
- **Pressure Measurement (`getPressure()`)**: Returns the atmospheric pressure in hectopascals (hPa) after conversion from Pascals.

Usage Example

```
Sensor sensor(true); // Create a sensor instance using metric units
if (sensor.begin()) {
    float temp = sensor.getTemperature(); // Get temperature in Celsius
    float humidity = sensor.getHumidity(); // Get humidity percentage
    float pressure = sensor.getPressure(); // Get atmospheric pressure in hPa
}
```

Display Class

The `Display` class is responsible for managing the **E-Ink display**, rendering real-time **temperature readings**, and providing a graphical representation of environmental data. It ensures that the display updates only when necessary to conserve power.

Class Definition

```
class Display {
public:
    Display(); // Constructor that initializes display settings
    void init(); // Initializes the display hardware
    void clear(); // Clears the screen before updating
    void drawFace(); // Draws a visual face layout on the display
    void drawTemperature(float temperature); // Updates the display with a graphical temperature
    bool needsRedraw(float currentTemp, float previousTemp); // Determines if an update is neces

private:
    Inkplate display; // Object handling E-Ink display operations
    const uint8_t* face; // Pointer to an image representing a static UI element
    int faceWidth; // Width of the face image
    int faceHeight; // Height of the face image
};
```

How It Works

- **Initialization (`init()`)**: Sets up the E-Ink display and prepares it for rendering data.
- **Clearing (`clear()`)**: Ensures that previous content does not interfere with new updates.
- **Drawing UI Elements (`drawFace()`)**: Displays a predefined UI image on the screen, such as a thermometer face.
- **Updating Temperature (`drawTemperature(float temperature)`)**: Visually represents temperature as a bar that grows or shrinks accordingly.
- **Redraw Optimization (`needsRedraw(float currentTemp, float previousTemp)`)**: Prevents unnecessary updates if the temperature hasn't changed significantly, reducing flicker and saving power.

Usage Example

```
Display epaper;
epaper.init(); // Initialize the display
epaper.drawFace(); // Draw initial UI layout
float temp = sensor.getTemperature();
epaper.drawTemperature(temp); // Display current temperature
```

Thermometer Image Data

The `thermometer.h` file contains **raw image data** for rendering a thermometer icon on the E-Ink display. The image is stored as a **byte array** and used to create a graphical representation of the thermometer.

How It Works

- The image is stored in a **constant byte array** (`const uint8_t thermometer[]`), ensuring efficient memory usage.
- The display class reads this array and renders the thermometer image onto the screen.
- The image provides a static visual reference that complements the **dynamic temperature bar**.

Usage in Code

```
extern const uint8_t thermometer[]; // Declaring external image array

// Usage in display class
void Display::drawFace() {
    display.drawBitmap(0, 0, thermometer, faceWidth, faceHeight, BLACK);
}
```

Why Use a Byte Array?

- **Efficient Storage**: Stores the image as a compressed bitmap in program memory (PROGMEM).
- **Low Power**: Reduces processing overhead by storing static images instead of dynamically generating graphics.
- **Faster Rendering**: Allows quick drawing on the E-Ink display.

SensorData Class

The `SensorData` class is responsible for collecting, storing, and providing access to historical **temperature, humidity, and pressure data** from the sensor. It maintains a rolling buffer to efficiently manage data points over time.

Class Definition

```
class SensorData {
public:
    SensorData(int dataPoints, Sensor& sensor); // Constructor that initializes storage
    ~SensorData(); // Destructor to free allocated memory
    void update(); // Updates stored sensor readings
    String getCurrentDataJson(); // Returns latest sensor readings in JSON format
    String getHistoricalDataJson(); // Returns historical data in JSON format

private:
    Sensor& sensor; // Reference to the sensor object
    const int DATA_POINTS; // Maximum number of data points to store
    int currentIndex; // Tracks the latest stored data index
    float* temperatureHistory; // Rolling buffer for temperature readings
    float* humidityHistory; // Rolling buffer for humidity readings
    float* pressureHistory; // Rolling buffer for pressure readings
};
```

How It Works

- **Initialization (`SensorData(int dataPoints, Sensor& sensor)`)**: Allocates memory for historical data storage and associates the sensor.
- **Updating Data (`update()`)**: Retrieves the latest sensor readings and stores them in rolling buffers, ensuring old values are replaced over time.
- **Fetching Current Data (`getCurrentDataJson()`)**: Returns a JSON-formatted string with the most recent temperature, humidity, and pressure readings.
- **Fetching Historical Data (`getHistoricalDataJson()`)**: Provides historical sensor readings in JSON format for visualization or external use.

Usage Example

```
SensorData sensorData(48, sensor);
sensorData.update(); // Capture latest sensor readings
String currentData = sensorData.getCurrentDataJson();
Serial.println(currentData); // Print JSON-formatted data
```

HTML & JavaScript Structure

The web interface is defined within `index_html.h`, stored in **program memory (PROGMEM)** to be served directly by the ESP32 web server.

Key Components

HTML Structure

- **Header (`<head>`)**
  - Includes **Chart.js** for graph rendering.
  - Contains a `<meta>` tag for viewport scaling, ensuring mobile compatibility.
  - Defines **CSS styles** for layout and theming.
- **Body (`<body>`)**
  - Displays the **Home Environment Dashboard title**.
  - Includes a **current values section** displaying:
    - **Temperature**
    - **Humidity**
    - **Pressure**
  - Contains a **graph section** displaying historical data for temperature, humidity, and pressure using `<canvas>` elements.
  - Includes a **footer** noting that data is stored for 48 hours.

CSS Styling

- Uses a **clean, modern design** with:
  - **Light background** (`#f4f4f9`) and dark text for contrast.
  - **Card-based UI** for displaying live data.
  - **Box shadows** for better readability.
  - **Flexbox layout** for responsiveness.

JavaScript Functionality

- **Data Fetching**
  - Calls `/sensor` endpoint for live temperature, humidity, and pressure data.
  - Calls `/history` endpoint to retrieve historical data.
  - Updates UI dynamically every **5 seconds**.
- **Graph Rendering**
  - Uses **Chart.js** to visualize sensor readings.
  - Displays data over the last **48 hours**.
  - Labels time intervals as "hours ago" for clarity.

Endpoints Used

Endpoint	Method	Description
`/sensor`	GET	Retrieves current temperature, humidity, and pressure in JSON format.
`/history`	GET	Fetches historical data (last 48 hours) in JSON format.

Usage Example

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/html", index_html);
});
```

Learning Opportunities

Building this Home Environment Dashboard is an excellent opportunity to enhance your programming skills across multiple domains. The project combines hardware interfacing, object-oriented programming, web development, and data visualization into a practical, real-world application. If you're eager to expand your knowledge further, 👋 visit **programmingwithnick.com**, where you'll find comprehensive tutorials and courses covering everything from basic electronics to advanced programming concepts. Whether you're interested in IoT development, web technologies, or embedded systems, there's always something new to learn and explore in the world of programming.