Inhaltsverzeichnis

Einleitung

Das Ziel ist die Fernsteuerung eines ESP32 über WLAN. Der ESP32 stellt dabei auf einem Webserver die Benutzeroberfläche zur Verfügung.

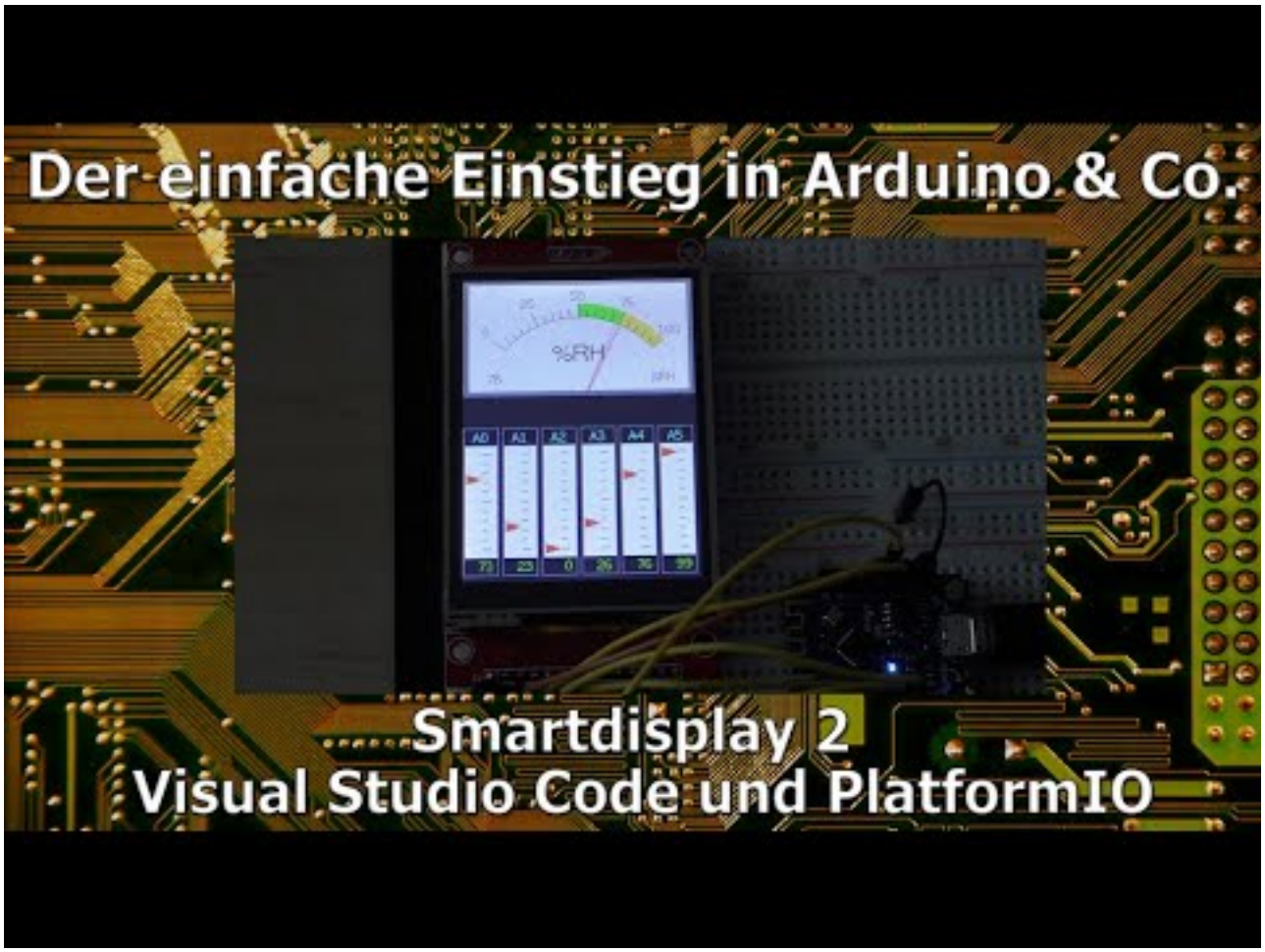
Die notwendigen Erkenntnisse habe ich aus dem Buch ***Build Web servers with ESP32 and ESP8266*** von ***Rui Santos und Sara Santos.***

https://randomnerdtutorials.com/build-web-servers-esp32-esp8266-ebook/

Das hier vorgestellte Projekt ist auf meiner Github - Seite zu finden.

https://github.com/hobbyelektroniker/Arduino_Einstieg

Für das Projekt verwende ich Visual Studio Code und PlatfomIO. Falls du das noch nicht kennst, findest du eine kleine Einführung in diesem Video.



https://youtu.be/zDvnHF9hlUI

Das Projekt

Neues Projekt erstellen

Das neue Projekt wird in PlatformIO unter dem Namen ***ESP32Remote_1*** angelegt.

Board: WEMOS LOLIN D32 PRO
Framework: Arduino

Dateien und Verzeichnisse

Die automatisch angelegten Dateien
PlatformIO legt einige Verzeichnisse und Dateien an. Wir müssen uns nur mit einem kleinen Teil davon befassen.

`src/main.cpp`	Das Hauptprogramm
`platformio.ini`	Projektkonfiguration

platformio.ini
Hier werden einige Ergänzungen angebracht.

```
[env:lolin_d32_pro]
platform = espressif32
board = lolin_d32_pro
framework = arduino
monitor_speed = 115200
monitor_port = /dev/cu.wchusbserial14410
upload_speed = 115200
upload_port = /dev/cu.wchusbserial14410
lib_deps = ESP Async WebServer
```

Die Ports sollten definiert werden. Wir verwenden für Monitor und Upload denselben Port. Es wird ein Mac verwendet, bei Windows müsste der entsprechende COM-Port angegeben werden. PlatformIO könnte zwar den Port selbstständig suchen, doch manchmal führt diese Suche zu falschen Resultaten.

Die Geschwindigkeit könnte auch höher sein (ausprobieren!). 115200 sollte aber auf jeden Fall funktionieren.

Besonders wichtig ist die letzte Zeile (***lib_deps***). Wir möchten den ESP32 als Webserver verwenden und müssen die entsprechende Library angeben. Zusätzliche Informationen zu diesem Webserver findet man auf Github: https://github.com/me-no-dev/ESPAsyncWebServer

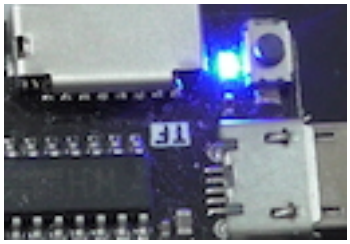
settings.h
Diese Datei müssen wir im Verzeichnis ***src*** anlegen. Hier werden Konfigurationsinformationen abgelegt. Momentan sind das nur die Zugangsdaten zum WLAN.

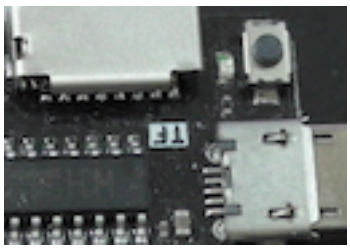
```
const char* ssid = "ssid";
const char* password = "password";
```

Das Verzeichnis data
Wir möchten einen Webserver betreiben. Dazu benötigen wir .html, .css und .js - Files. Diese legen wir in einem Verzeichnis ***data*** ab.
Die Webseite wird sehr einfach. Aus diesem Grund genügt es, wenn wir die Dateien ***index.html***, ***style.css, webSocket.js*** und ***script.js*** anlegen.

Alle Dateien in diesem Verzeichnis sind Bestandteil der Client - Programmierung. Wir werden später sehen, wie wir sie auf den ESP32 bringen.

Zusätzlich benötigen wir noch die Dateien ***led_on.jpg*** und ***led_off.jpg***. Das sind Fotos vom realen ESP32 - Board.





Die Webseite

Wir bauen eine einfache Webseite auf. Diese läuft noch unabhängig vom ESP32.

Die html - Dateien

Das Beispiel ist sehr einfach, wir verwenden nur eine einzige html - Seite.

index.html

```
<!DOCTYPE html>
<html>

<head>
  <title>ESP Led - Controller</title>
  <link rel="icon" href="data:,">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" type="text/css" href="style.css">
  <script src="webSocket.js"></script>
  <script src="script.js"></script>
</head>

<body>
  <h1>ESP Led - Controller</h1>
  <img id="imageLed" src="led_off.jpg">
  <button id="toggle_button">EIN</button>
</body>

</html>
```

Das können wir bereits im Browser anschauen. Es genügt, wenn wir die Datei mit einem Browser öffnen. Wir machen hier keinen HTML - Kurs, daher sind wir mit wenig zufrieden.

Die Seite hat nur zwei wesentliche Elemente: ein Bild und einen Button. Beide sind mit einer id versehen, so dass wir darauf zugreifen können.

```
<img id="imageLed" src="led_off.jpg">
<button id="toggle_button">EIN</button>
```

Um doch noch einige kleine Darstellungseffekte einzubauen, binden wir die Datei ***style.css*** ein. Das machen wir im Header.

```
<link rel="stylesheet" type="text/css" href="style.css">
```

Da es sich um eine aktive Seite handelt, binden wir noch die beiden Java Script Dateien ein.

```
<script src="webSocket.js"></script>
<script src="script.js"></script>
```

Die css - Dateien

Auch hier benötigen wir nur eine Datei.

style.css

```
html {
    font-family: Arial, Helvetica, sans-serif;
    text-align: center;
}

Button {
    border: none;
    color: #0a0a8a;
    background-color: #c2ccd6;
    padding: 15px 32px;
    text-align: center;
    font-size: 16px;
    width: 100px;
    border-radius: 4px;
    transition-duration: 0.4s;
```

```
}

button:hover {
    background-color: #1282A2;
    color: #e2e2eb;
}
```

Der Button wird etwas vergrössert und noch mit einem hover - Effekt versehen.

Die Java Script Dateien

Diesmal benötigen wir zwei Dateien.

webSocket.js

Hier haben wir nur zwei Regionen:

```
//#region Globale Variablen, Konstanten und Objekte
//#region Initialisierung und Default - Eventhandler


// Globale Variablen, Konstanten und Objekte
var websocket;

// Initialisierung und Default - Eventhandler
//=== Default - Eventhandler ====
function initWebSocket() {
    var gateway = `ws://${window.location.hostname}/ws`;
    console.log(gateway)
    websocket = new WebSocket(gateway);
    websocket.onopen    = onWebSocketOpen;
    websocket.onclose   = onWebSocketClose;
    websocket.onmessage = onWebSocketMessage;
}

// Default - Eventhandler
function onWebSocketOpen(event) {
    console.log("WebSocket Open");
}

function onWebSocketClose(event) {
    console.log("WebSocket Close");
    // Bei Verbindungsverlust nach 2 Sekunden
    // neue Verbindungsaufnahme versuchen
    setTimeout(initWebSocket, 2000);
}

function onWebSocketMessage(event) {
    console.log("WebSocket Message: " + event.data);
}
```

Diese Datei muss im Normalfall nicht verändert werden. Sie initialisiert die WebSocket - Verbindung und weist dem WebSocket - Objekt default - Handler zu.

script.js

Hier haben wir fünf Regionen:

```
//#region Globale Variablen, Konstanten und Objekte
//#region Startcode
//#region Events von HTML-Elementen
//#region Eigene Eventhandler für WebSocket Events
//#region Andere Funktionen
```

Das ganze clientseitige Programm finden wir in dieser Datei. Ich habe sie in 6 Bereiche aufgeteilt, die wir jetzt einzeln anschauen.

```
// Globale Variablen, Konstanten und Objekte
var isOn = false;
```
Wir benötigen eine einzige Variable, um den Status der Led zu speichern.

```
// Startcode
window.addEventListener("load",  onload);

function  onload(event) {
    initWebSocket();      // WebSocket initialisieren
    addWebSocketEvents(); // Eigene Eventhandler für WebSocket Events
    addHTMLEvents();      // HTML-Events den Eventhandlern zuweisen
}
```
Wir definieren den Event ***load***. Er wird immer ausgelöst, wenn die Seite neu geladen wird. Hier werden die WebSocket - Verbindung aufgebaut und verschiedene Events ihren Handlern zugewiesen.

```
// Eigene Eventhandler für WebSocket Events
function addWebSocketEvents() {
    websocket.onopen    = onOpen;
    websocket.onmessage = onMessage;
}

function onOpen(event) {
    // Info über den aktuellen Zustand vom Server anfordern
    websocket.send("INFO");
}

function onMessage(event) {
    // Wir erhalten ON oder OFF
    isOn = (event.data == "ON");
    refreshPage();
}
```
Für die WebSocket - Events ***opopen*** und ***onmessage*** verwenden wir eigene Handler. ***onOpen()*** sendet eine ***INFO*** - Anfrage an den Server, ***onMessage()*** setzt die ***isOn*** Variable abhängig von der Antwort.

```
// Events von HTML-Elementen
function addHTMLEvents() {
    document.getElementById('toggle_button').addEventListener('click',
                                                               toggleLed);
}

// Eventhandler für den Click auf den Button
function toggleLed(event) {
    isOn = !isOn;
    // Den Server informieren
    if (isOn) websocket.send('ON'); else websocket.send('OFF');
}
```
Dieser Event reagiert auf einen Druck auf den Button und sendet die Information an den Server weiter.

```
// Andere Funktionen
// Aktualisieren der Webseite
function refreshPage() {
    if (isOn) {
        document.getElementById("toggle_button").innerHTML="AUS";
        document.getElementById("imageLed").src="led_on.jpg";
    } else {
        document.getElementById("toggle_button").innerHTML="EIN";
        document.getElementById("imageLed").src="led_off.jpg";
    }
}
```

Das Server - Programm (im Verzeichnis *src*)

Das ist der anspruchvollste Teil. Obwohl wir nebst settings.h nur noch eine einzige Datei verwenden, ist dieser Programmteil schon recht komplex.

settings.h

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

Hier musst du deine Zugansdaten eintragen.

main.cpp

Auch dieses Programm ist in verschiedene Regionen aufgeteilt.

```
#pragma region Includes
#pragma region Konstanten, globale Variablen und Objekte
#pragma region Forward declarations
#pragma region Initialisierungen und Start des Webservers
#pragma region Ansteuerung der Led
#pragma region Kommunikation über WebSocket
#pragma region Andere Funktionen
```

// Includes

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include "settings.h"
```

Das sind die Bibliotheken, die wir benötigen. Ausserdem finden wir hier den Include unserer eigenen Datei ***settings.h***.

```
// Konstanten, globale Variablen und Objekte
// Konstanten
const int ledPin = LED_BUILTIN;

// Objekte
AsyncWebServer server(80); // Webserver auf Port 8080
AsyncWebSocket ws("/ws");  // Websocket - Verbindung
```

Wir arbeiten mit der OnBoard - Led und weisen der Variablen ledPin den entsprechenden Pin zu.
Ausserdem benötigen wir Instanzen für den Webserver und die WebSocket - Verbindung. Beide arbeiten auf Port 80.

```
// Forward declarations
// Diese Funktionen müssen weiter unten noch implementiert werden!
void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client,
     AwsEventType type, void *arg, uint8_t *data, size_t len);

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len);
```

Wir müssen die Reihenfolge der Funktionen einhalten. Deshalb befinden sich hier Forward - Deklarationen für Funktionen, die erst weiter unten implementiert werden.

```
// Initialisierungen und Start des Webservers
// Mit WLAN verbinden
void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.setHostname("espled");
  WiFi.begin(ssid, password);
  Serial.print("Verbinden mit WLAN ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
```

```
  Serial.println(WiFi.localIP());
}

// Filesystem initialisieren
void initSPIFFS() {
  if (!SPIFFS.begin(true)) {
    Serial.println("SPIFFS konnte nicht initialisiert werden");
  }
  Serial.println("SPIFFS bereit");
}

void initWebSocket() {
  ws.onEvent(onEvent);
  server.addHandler(&ws);
}

void startWebServer() {
  // Requesthandler definieren
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/index.html", "text/html");
  });
  server.serveStatic("/", SPIFFS, "/");

  // Start server
  server.begin();
}
```

Hier werden die einzelnen Komponenten initialisiert. In ***initWiFi()*** verbinden wir und mit dem WLAN, ***initSPIFFS()*** stellt uns ein FileSystem zur Verfügung, in dem wir die Daten aus dem Verzeichnis ***data*** ablegen können. Näheres dazu folgt im nächsten Kapitel.
initWebSocket() initialisiert WebSocket und ***startWebServer()*** startet den Server.

```
// Ansteuerung der Led
// Ein- oder Ausschalten der Led
// Meine Led leuchtet bei LOW!
// Daher die Invertierung mit !
void setLed(bool on) {
  digitalWrite(ledPin, !on);
  if (on) Serial.println("on"); else Serial.println("off");
}

// Abfrage der Led
bool getLed() {
  return !digitalRead(ledPin);
}
```

Hier wird die Led angesteuert. Mit ***setLed()*** kann sie ein- oder ausgeschaltet werden. ***getLed()*** kann ihren Zustand abfragen.

```
// Kommunikation über WebSocket
// Alle Clients benachrichtigen
void notifyClients(String state) {
  ws.textAll(state);
}

// Empfangene Meldung verarbeiten
void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
  AwsFrameInfo *info = (AwsFrameInfo*)arg;
  if (info->final && info->index == 0 &&
      info->len == len && info->opcode == WS_TEXT) {
    data[len] = 0;
    if (strcmp((char*)data, "INFO") == 0) {
       // Ein Client hat Informationen angefordert
       if (getLed()) notifyClients("ON"); else notifyClients("OFF");
    }
    if (strcmp((char*)data, "ON") == 0) {
      setLed(true);
```

```
      notifyClients("ON");
    }
    if (strcmp((char*)data, "OFF") == 0) {
      setLed(false);
      notifyClients("OFF");
    }
  }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client,
             AwsEventType type, void *arg, uint8_t *data, size_t len) {
  switch (type) {
    case WS_EVT_CONNECT:
      Serial.printf("WebSocket client #%u connected from %s\n",
                    client->id(), client->remoteIP().toString().c_str());
      break;
    case WS_EVT_DISCONNECT:
      Serial.printf("WebSocket client #%u disconnected\n", client->id());
      break;
    case WS_EVT_DATA:
      handleWebSocketMessage(arg, data, len);
      break;
    case WS_EVT_PONG:
    case WS_EVT_ERROR:
      break;
  }
}
```

Hier wird WebSocket behandelt. ***handleWebSocketMessage()*** behandrlt die empfangenen Meldungen und muss den eigenen Anforderungen angepasst werden. ***onEvent()*** habe ich unverändert aus einem Beispiel übernommen.

```
// Andere Funktionen
void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  setLed(false); // LED ausschalten
  initWiFi();
  initSPIFFS();
  initWebSocket();
  startWebServer();
}
```

Setup beinhaltet keine Überraschungen. Hier werden einfach alle Initialisierungen der Reihe nach aufgerufen.

```
void loop() {
  // Das wird momentan nicht benötigt
}
```

loop() ist leer, da der Server im Hintergrund läuft.

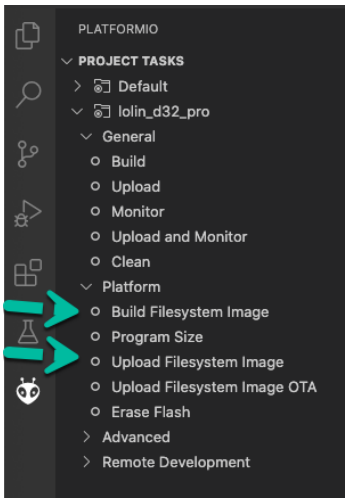
Das Filesystem SPIFFS

Wir möchten die Dateien im ***data*** - Verzeichnis durch den Webserver ausliefern lassen. Dazu sollten die Dateien auf dem ESP32 zur Verfügung stehen. Glücklicherweise stellt SPIFFS ein Filesystem für den ESP zur Verfügung. Wir können darin unsere Dateien speichern. Wer sich für die Details interessiert, findet diese in der Espressif - Dokumentation (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html).

Wir verwenden die Unterstützung durch PlattformIO, was die ganze Sache sehr einfach macht.

Erstellen des Fileimages und Übertragen der Dateien

In Visual Studio Code gehen wir auf PlatformIO / Project Tasks.



Zuerst führen wir ***Build Filesystem Image*** aus. Danach kann mit ***Upload Filesystem Imag***e das Image auf den ESP32 kopiert werden.

Das funktioniert nur, wenn der serielle Port nicht in Benutzung ist. Es sollte also kein aktives Terminal offen sein.

Nach jeder Änderung einer Datei in ***data*** muss dieser Vorgang wiederholt werden.

Server starten

Wir laden das Programm auf den ESP32 und führen es aus. Im Terminal sehen wir, ob die Verbindung mit dem WLAN klappt und welche IP-Adresse der ESP32 bekommt.

Im Browser können wir dann ***http://<IP>*** aufrufen.