Inhaltsverzeichnis

- Inhaltsverzeichnis
  - Einführung
  - IDE-Auswahl und Hardware-Aufteilung
  - Home Assistant und MQTT-Integration
  - Erste Tests und Probleme
  - LED-Streifen und Motorsteuerung
  - Lautsprecher und Tonerzeugung
  - Fotoresistor und Lichtsteuerung
  - Zusammenfassung
  - Anhänge

Einführung

Um mit dem Projekt anzufangen, mussten wir zuerst eine IDE auswählen, um mit dem ESP32 zu coden. Wir entschieden uns für PlatformIO. Wir haben die Hardware aufgeteilt, damit wir in den Ferien selbst forschen und probieren konnten.

IDE-Auswahl und Hardware-Aufteilung

Tri hat nach einem Weg gesucht, um den ESP32 für eine Smarthome-Integration zu verwenden, und hat zunächst Google Home angeschaut. Ihm wurde schnell klar, dass es sehr kompliziert und vor allem gerade schwierig ist, weil Google ihre Tools migriert. AWS war auch eine Option, aber nur als Trial (Testzeit). Das Gleiche gilt für IFTTT, und Tri wollte eher etwas, das unabhängig von irgendwelchen Abonnements ist. Deshalb kam Home Assistant als Selfhosting-Instanz in Frage, bei der man über eine MQTT-Integration den ESP32 mit Home Assistant verbinden kann.

Home Assistant und MQTT-Integration

Tri hat also einen Docker-Container mit Home Assistant und einem MQTT-Broker eingerichtet. Das Erste, was Tri testen wollte, war, ob der ESP32 sich überhaupt mit meinem Heimnetzwerk verbinden konnte. Als es Zeit war, den Code auszuprobieren, gab es das Problem, dass der ESP32 nicht erkannt wurde. Es war nur ein Treiberproblem, und nach vielen chinesischen Treibern wurde er irgendwann erkannt. Nach einigen Code-Problemen, weil alles in C geschrieben ist, lief er schließlich und konnte Tri seine IP-Adresse liefern. Erfolg! Er hat sich verbunden, und man konnte ihn pingen.

```
services:
  home_assistant:
    image: lscr.io/linuxserver/homeassistant:latest
    container_name: home_assistant
    restart: unless-stopped
    ports:
      - "8124:8123"
    cap_add:
      - NET_ADMIN
      - NET_RAW
    volumes:
      - ./home_assistant/config:/config  # Home Assistant config folder
      - /var/run/dbus:/var/run/dbus:ro
      - /etc/localtime:/etc/localtime:ro
    environment:
      - TZ=Europe/Berlin # Set your timezone here
      - MQTT_HOST=localhost
      - MQTT_PORT=1883
      - MQTT_USERNAME=koti
      - MQTT_PASSWORD=kot


  mosquitto:
    image: eclipse-mosquitto:latest
    container_name: mosquitto
    restart: always
    ports:
      - "1883:1883"   # MQTT port
      - "9001:9001"   # WebSocket port (optional)
    volumes:
      - ./mosquitto/config:/mosquitto/config  # Mosquitto config folder
      - ./mosquitto/data:/mosquitto/data      # Mosquitto data folder
      - ./mosquitto/log:/mosquitto/log        # Mosquitto log folder
    environment:
      - TZ=Europe/Berlin # Set Mosquitto's timezone
```

Zuerst wurde ein Docker-Container erstellt, der sowohl Home Assistant als auch den MQTT-Broker (Mosquitto) enthält. Dies ermöglicht eine einfache Verwaltung und Skalierung der Smarthome-Infrastruktur.

```yaml
# Loads default set of integrations. Do not remove.
default_config:
# Load frontend themes from the themes folder
frontend:
  themes: !include_dir_merge_named themes
# Include automations, scripts, and scenes
automation: !include automations.yaml
script: !include scripts.yaml
scene: !include scenes.yaml
http:
  use_x_forwarded_for: true
  trusted_proxies:
    - 127.0.0.1
    - 192.168.178.79
    - ::1

# MQTT Configuration
mqtt:
  switch:
    - name: "My Led Strip"
      state_topic: "home/esp32/relay/state"
      command_topic: "home/esp32/relay/set"
      payload_on: "ON"
      payload_off: "OFF"
      state_on: "ON"
      state_off: "OFF"
      qos: 1
      retain: true

    - name: "LOFF/LON Switch"
      state_topic: "home/esp32/relay/state"
      command_topic: "home/esp32/relay/set2"
      payload_on: "LON"
      payload_off: "LOFF"
      state_on: "LON"
      state_off: "LOFF"
      qos: 1
      retain: true

# Input Text for Color Picker
input_text:
  color_picker:
    name: Color Picker
    initial: "FFFFFF"  # Default value, ensures the field is never empty
    min: 6             # Minimum length of 6 characters
    max: 6             # Maximum length of 6 characters
    pattern: "^[0-9A-F]{6}$"  # Only allows 6 characters of 0-9 and A-F

# Input Number for Slider and Number of Turns
input_number:
  number_field:
    name: Number of Turns
```

```
    initial: 0       # Start at 0
    min: -12         # Minimum value
    max: 12          # Maximum value
    step: 1          # Step size (whole numbers only)
    mode: slider     # Display as a slider in the UI

  slider_value:
    name: Motor Speed
    initial: 5       # Default value (midpoint of 1-10)
    min: 1           # Minimum value
    max: 10          # Maximum value
    step: 1          # Step size (whole numbers only)
    mode: slider     # Display as a slider in the UI

# Input Buttons
input_button:
  send_mon_button:
    name: Send mON

  send_sbn_button:  # Corrected slug
    name: Send sON

# Automations
automation:
  - alias: Send Color or Slider via MQTT
    trigger:
      - platform: state
        entity_id: input_text.color_picker
      - platform: state
        entity_id: input_number.slider_value
    action:
      - choose:
          - conditions: "{{ trigger.entity_id == 'input_text.color_picker' }}"
            sequence:
              - service: mqtt.publish
                data:
                  topic: "home/esp32/relay/set"
                  payload: "#{{ states('input_text.color_picker') }}"  # Add the
hashtag for color
                  qos: 1
                  retain: true
          - conditions: "{{ trigger.entity_id == 'input_number.slider_value' }}"
            sequence:
              - service: mqtt.publish
                data:
                  topic: "home/esp32/relay/set"
                  payload: "s{{ states('input_number.slider_value') | int }}"  #
Add 's' for slider
                  qos: 1
                  retain: true
  - alias: "Send LON or LOFF based on switch state"
    trigger:
      - platform: state
        entity_id: switch.loff_lon_switch  # Replace with the actual entity ID of
```

```
your switch
    action:
      - choose:
          - conditions: "{{ trigger.to_state.state == 'on' }}"
            sequence:
              - service: mqtt.publish
                data:
                  topic: "home/esp32/relay/set2"
                  payload: "LON"  # Send LON when the switch is turned on
                  qos: 1
                  retain: true
          - conditions: "{{ trigger.to_state.state == 'off' }}"
            sequence:
              - service: mqtt.publish
                data:
                  topic: "home/esp32/relay/set2"
                  payload: "LOFF"  # Send LOFF when the switch is turned off
                  qos: 1
                  retain: true
  - alias: Send mON and Number of Turns on Button Press
    trigger:
      - platform: event
        event_type: call_service
        event_data:
          domain: input_button
          service: press
          service_data:
            entity_id: input_button.send_mon_button
    action:
      - service: mqtt.publish
        data:
          topic: "home/esp32/relay/set"
          payload: "MON{{ states('input_number.number_field') | int }}"  # Send
'MON' and the number of turns
          qos: 1
          retain: true
  - alias: Send sON on Button Press
    trigger:
      - platform: event
        event_type: call_service
        event_data:
          domain: input_button
          service: press
          service_data:
            entity_id: input_button.send_sbn_button
    action:
      - service: mqtt.publish
        data:
          topic: "home/esp32/relay/set"
          payload: "AON"  # Send 'sON'
          qos: 1
          retain: true
```

Um MQTT und Homeassistant wirklich benutzen zu können musste Tri in der Config elemente Hinzufügen. In der Homeassistant config musste Tri Schalter (`My LED Strip` & `LOFF/LON Switch`) einfügen die den Zustand der letzt gesendeten Nachricht darstellen. Dazu einpaar Slider die bei Wert Änderungen, ein Buchstaben nehmen und den Wert anhängend zum mqtt broker schicken. Das gleiche haben wir für die Textbox (Hexadezimalen Farbcode) und 2 Buttons gemacht.

```
# General settings
persistence true
persistence_location /mosquitto/data/
log_dest file /mosquitto/log/mosquitto.log

# Default listener for MQTT
listener 1883
allow_anonymous false
password_file /mosquitto/config/passwords.txt

# WebSocket listener (optional)
listener 9001
protocol websockets

log_dest stderr
```

In der MQTT config hat Tri die Umgebung etwas definiert, welcher Port verwendet wird und vorallem ob sich geräte ohne Password und ID sich zu dem Broker verbinden können. In diesem Fall nicht (`allow_anonymous false`) und wir sagen ihm wo er das Passwort finden kann.

```
// Wi-Fi credentials
#define WIFI_SSID "Internet Name"
#define WIFI_PASSWORD "Internet Passwort"

// MQTT Broker settings
#define MQTT_BROKER_URI "mqtt://192.168.178.118:1883"
#define MQTT_USERNAME "koti"
#define MQTT_PASSWORD "kot"
```

Hier wird definiert, mit welchem Netzwerk und mit welchem Broker wir uns verbinden. Der Broker kann auch auf einem anderen Gerät gehostet werden.

```
static EventGroupHandle_t wifi_event_group;
#define WIFI_CONNECTED_BIT BIT0

static const char *TAG = "MQTT_APP";
static esp_mqtt_client_handle_t mqtt_client;

// Event handler for Wi-Fi events
static void wifi_event_handler(void *arg, esp_event_base_t event_base, int32_t
event_id, void *event_data) {
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
        ESP_LOGI(TAG, "Trying to connect to Wi-Fi...");
    } else if (event_base == WIFI_EVENT && event_id ==
WIFI_EVENT_STA_DISCONNECTED) {
        ESP_LOGW(TAG, "Wi-Fi disconnected. Reconnecting...");
        esp_wifi_connect();
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t *event = (ip_event_got_ip_t *)event_data;
        char ip_str[IP4ADDR_STRLEN_MAX];
        esp_ip4addr_ntoa(&event->ip_info.ip, ip_str, IP4ADDR_STRLEN_MAX);
        ESP_LOGI(TAG, "Connected successfully. IP Address: %s", ip_str);
        xEventGroupSetBits(wifi_event_group, WIFI_CONNECTED_BIT);
    }
}

void init_wifi(void) {
    esp_netif_init();
    esp_event_loop_create_default();
    esp_netif_create_default_wifi_sta();
    wifi_event_group = xEventGroupCreate();
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    esp_wifi_init(&cfg);
    esp_event_handler_instance_register(WIFI_EVENT, ESP_EVENT_ANY_ID,
&wifi_event_handler, NULL, NULL);
    esp_event_handler_instance_register(IP_EVENT, IP_EVENT_STA_GOT_IP,
&wifi_event_handler, NULL, NULL);
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = WIFI_SSID,
            .password = WIFI_PASSWORD,
        },
    };
    esp_wifi_set_mode(WIFI_MODE_STA);
    esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config);
    esp_wifi_start();
    ESP_LOGI(TAG, "Waiting for connection...");
    xEventGroupWaitBits(wifi_event_group, WIFI_CONNECTED_BIT, pdFALSE, pdTRUE,
portMAX_DELAY);
}
```

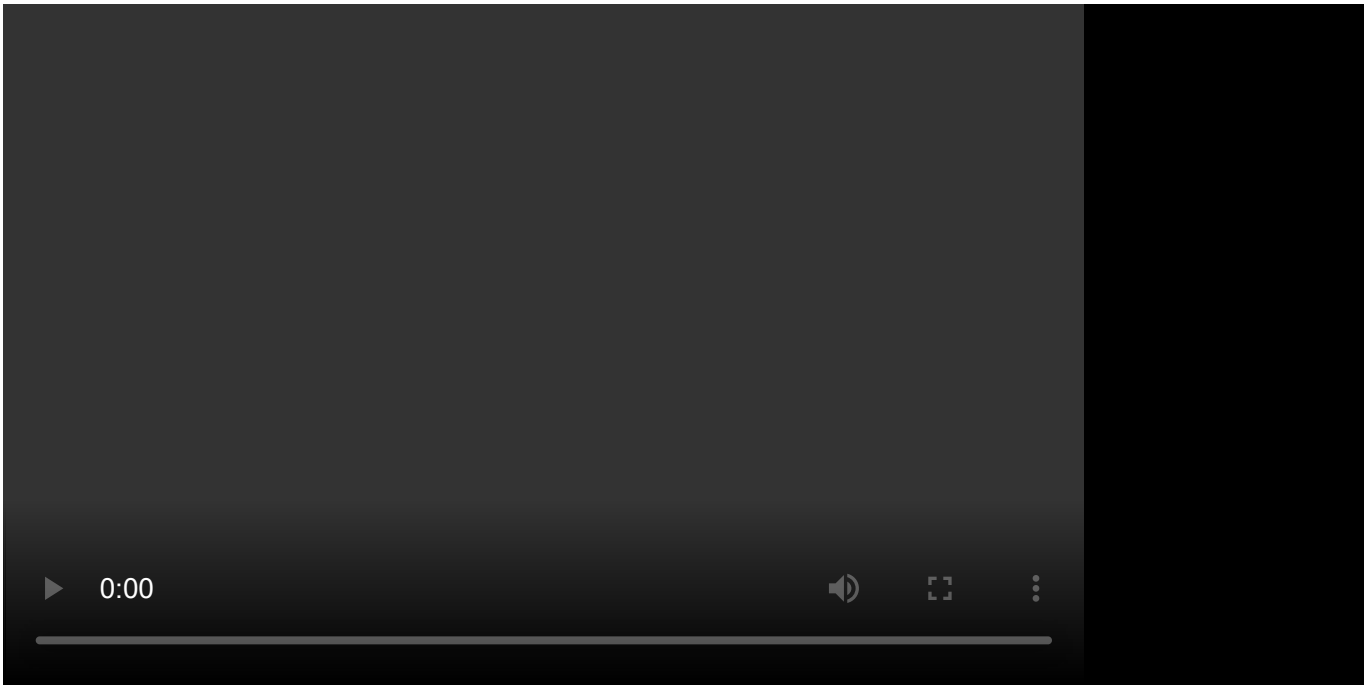
Als Nächstes wollte war zu testen, ob man einem MQTT-Topic subscriben und etwas publishen kann. Das ging relativ leicht und funktionierte direkt sowohl in Testterminals als auch im Home Assistant GUI. Somit war es möglich, Nachrichten an den ESP32 senden, und der ESP32 konnte darauf antworten und Aufgaben ausführen, wie zum Beispiel einen Pin auf High zu stellen. Genau das hat Tri auch gecodet. Das Testen war schwierig, weil er kein Multimeter hatte, also musste er sich am nächsten Morgen eins von seinem Onkel ausleihen. Es stellte sich heraus, dass alles funktioniert.

```
void mqtt_app_start(void) {
    ESP_LOGI(TAG, "Initializing MQTT...");
    esp_mqtt_client_config_t mqtt_cfg = {
        .broker.address.uri = MQTT_BROKER_URI,
        .credentials = {
            .username = MQTT_USERNAME,
            .authentication.password = MQTT_PASSWORD,
        },
    };
    ESP_LOGI(TAG, "MQTT configuration set.");

    mqtt_client = esp_mqtt_client_init(&mqtt_cfg);
    if (mqtt_client == NULL) {
        ESP_LOGE(TAG, "Failed to initialize MQTT client");
        return;
    }
    ESP_LOGI(TAG, "MQTT client initialized.");

    esp_mqtt_client_register_event(mqtt_client, ESP_EVENT_ANY_ID,
mqtt_event_handler, NULL);
    ESP_LOGI(TAG, "MQTT event handler registered.");

    esp_mqtt_client_start(mqtt_client);
    ESP_LOGI(TAG, "MQTT client started.");
}
```

Nachdem der ESP32 mit dem Wi-Fi-Netzwerk verbunden ist, wird eine Verbindung zum MQTT-Broker hergestellt. Der ESP32 abonniert ein Topic (home/esp32/relay/set), um Befehle von Home Assistant zu empfangen.

```
static void mqtt_event_handler(void *handler_args, esp_event_base_t base, int32_t
event_id, void *event_data) {
    esp_mqtt_event_handle_t event = (esp_mqtt_event_handle_t)event_data;
    ESP_LOGI(TAG, "Handling MQTT event...");
    switch (event_id) {
        case MQTT_EVENT_CONNECTED:
            ESP_LOGI(TAG, "Connected to MQTT broker");
            // Subscribe to the command topic
            esp_mqtt_client_subscribe(mqtt_client, "home/esp32/relay/set", 0);
            ESP_LOGI(TAG, "Subscribed to topic: home/esp32/relay/set");
            break;
        case MQTT_EVENT_DATA:
            ESP_LOGI(TAG, "Received message on topic: %.*s", event->topic_len,
event->topic);
            ESP_LOGI(TAG, "Message content: %.*s", event->data_len, event->data);
            // Handle command topic
            if (strncmp(event->topic, "home/esp32/relay/set", event->topic_len) ==
0) {
                if (strncmp(event->data, "ON", event->data_len) == 0) {
                    ESP_LOGI(TAG, "Turning relay ON");
                    gpio_set_level(RELAY_GPIO, 1);
                    // Publish the new state
                    esp_mqtt_client_publish(mqtt_client, "home/esp32/relay/state",
"ON", 0, 1, 1);
                } else if (strncmp(event->data, "OFF", event->data_len) == 0) {
                    ESP_LOGI(TAG, "Turning relay OFF");
                    gpio_set_level(RELAY_GPIO, 0);
                    // Publish the new state
                    esp_mqtt_client_publish(mqtt_client, "home/esp32/relay/state",
"OFF", 0, 1, 1);
                } else {
                    ESP_LOGW(TAG, "Unknown command: %.*s", event->data_len, event-
>data);
                }
            }
            break;
        case MQTT_EVENT_ERROR:
            ESP_LOGE(TAG, "MQTT_EVENT_ERROR");
            ESP_LOGE(TAG, "Reconnecting to MQTT broker...");
            esp_mqtt_client_reconnect(mqtt_client);
            break;
        default:
            ESP_LOGI(TAG, "Unhandled MQTT event: %" PRIi32, event_id);
            break;
    }
    ESP_LOGI(TAG, "Finished handling MQTT event.");
}
```

Der ESP32 reagiert auf MQTT-Nachrichten, die an das abonnierte Topic gesendet werden. Wenn eine Nachricht empfangen wird, wird die Funktion mqtt_event_handler aufgerufen. Diese Funktion überprüft den Inhalt der Nachricht und führt entsprechende Aktionen aus, z. B. das Einschalten eines Relais.



Erste Tests und Probleme

Da der Home Assistant und der ESP32 Nachrichten austauschen können, kann der ESP32 auf Wunsch des Home Assistans, wenn die richtige Hardware angeschlossen ist, zum Beispiel ein Licht einschalten oder einen Motor steuern, der eine Tür schließt. Der ESP32 kann auch als Sensor verwendet werden, der ein Signal an den Home Assistant sendet.

Während des Tests kam Tri mit den Pins in Kontakt und schloss sie versehentlich kurz, was den ESP32 zum Crashen brachte und dieser neustartete.

Er konnte jetzt allerdings nichts mehr machen, da er bisher nur seine privaten Teile verwendete und die anderen Hardwarekomponenten bei den anderen lagen.

LED-Streifen und Motorsteuerung

Es stellte sich heraus, dass es den anderen beiden nicht möglich war, etwas über die Ferien zu machen; einer von uns musste sich außerdem noch um Prüfungen kümmern. Es gab auch Probleme bei einem der Beiden, das dazu führte, dass sein ESP32 keine Daten empfangen konnte.

Nach den Ferien haben wir uns drangemacht und angefangen, mehr Hardware über den ESP32 anzusprechen. Wir haben den Code, der von Tri in C geschrieben wurde, von der Platform Espressif auf Arduino C++ umgestellt, weil das leichter zu coden ist. Jetzt, wo mehr Hardware zur Verfügung stand, konnten wir endlich testen, ob der ESP32 die Hardwareteile ansprechen kann.

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Arduino.h>

// Wi-Fi credentials
#define WIFI_SSID "Kotspot" // Replace with SSID
#define WIFI_PASSWORD "kotfressemagkot" // Replace with password

// MQTT Broker settings
#define MQTT_BROKER "192.168.167.225"
#define MQTT_PORT 1883
#define MQTT_USER "koti"
#define MQTT_PASSWORD "kot"

#define RELAY_GPIO 2

WiFiClient espClient;
PubSubClient mqttClient(espClient);

void setup_wifi() {
    delay(10);
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(WIFI_SSID);

    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }

    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
}

void mqtt_callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Message arrived [");
    Serial.print(topic);
    Serial.print("] ");
    for (int i = 0; i < length; i++) {
        Serial.print((char)payload[i]);
    }
    Serial.println();

    // Handle command topic
    if (strcmp(topic, "home/esp32/relay/set") == 0) {
        if (strncmp((char*)payload, "ON", length) == 0) {
            Serial.println("Turning relay ON");
            digitalWrite(RELAY_GPIO, HIGH);
```

```
            mqttClient.publish("home/esp32/relay/state", "ON");
        } else if (strncmp((char*)payload, "OFF", length) == 0) {
            Serial.println("Turning relay OFF");
            digitalWrite(RELAY_GPIO, LOW);
            mqttClient.publish("home/esp32/relay/state", "OFF");
        } else {
            Serial.print("Unknown command: ");
            Serial.println((char*)payload);
        }
    }
}

void reconnect() {
    // Loop until we're reconnected
    while (!mqttClient.connected()) {
        Serial.print("Attempting MQTT connection...");
        // Attempt to connect
        if (mqttClient.connect("ESP32Client", MQTT_USER, MQTT_PASSWORD)) {
            Serial.println("connected");
            // Subscribe
            mqttClient.subscribe("home/esp32/relay/set");
        } else {
            Serial.print("failed, rc=");
            Serial.print(mqttClient.state());
            Serial.println(" try again in 5 seconds");
            // Wait 5 seconds before retrying
            delay(5000);
        }
    }
}

void setup() {
    Serial.begin(115200);
    pinMode(RELAY_GPIO, OUTPUT);
    digitalWrite(RELAY_GPIO, LOW);  // Set the initial state to OFF

    setup_wifi();
    mqttClient.setServer(MQTT_BROKER, MQTT_PORT);
    mqttClient.setCallback(mqtt_callback);
}

void loop() {
    if (!mqttClient.connected()) {
        reconnect();
    }
    mqttClient.loop();
}
```

Dieser Code gleicht dem alten C Code von der Funktionsweise, verwendet aber das Arduino Framework. Zuerst verbindet sich der ESP32 mit dem WLAN-Netzwerk, in dem sich auch der Home Assistant und MQTT-Broker befinden; hierfür wurde ein Hotspot verwendet. Als nächstes verbindet er sich mit dem Broker, und Abonniert das Thema "home/esp32/relay/set". Wenn eine Nachricht mit diesem Thema empfangen wird, wird die Methode `mqtt_callback` aufgerufen. Diese untersucht den Inhalt der Nachricht handelt entsprechend; hierüber bauen wir alle neuen Funktionen ein.

Nach Implimentierung eines LED-Streifens und eines Motors sieht der Code dieser Methode wie folgt aus:

```
[neue Variablen]
CRGB ledcolor = CRGB::Gray; //Farbe des LED-Streifens
bool lightson = false; //Leuchtet der Streifen?
int motordelay = 1; //Konfigurierte Drehgeschwindikeit des Motors
bool turning = false; //Dreht sich der Motor?
[...]
```

```
[Ausschnitt aus mqtt_callback]
// Handle command topic
    if (strcmp(topic, "home/esp32/relay/set") == 0) {
        if (strncmp((char*)payload, "ON", length) == 0) {
            Serial.println("Turning relay ON");
            if (!turning) {
              lightson = true;
              std::thread(lightloop).detach();
            }
            mqttClient.publish("home/esp32/relay/state", "ON");
        } else if (strncmp((char*)payload, "OFF", length) == 0) {
            Serial.println("Turning relay OFF");
            lightson = false;
            mqttClient.publish("home/esp32/relay/state", "OFF");
        } else if (*payload == '#' && length == 7) {
            Serial.println("Special color command received.");
            std::string s((char*)payload);
            s = s.substr(1, 6);
            Serial.print("Setting color to ");
            std::cout << s << std::endl;
            ledcolor = std::stoul(s, nullptr, 16);
        } else if (*payload == 's') {
          //handle speed change
          std::string s((char*)payload, 1, length-1);
          motordelay = 11 - std::stoi(s);
          Serial.println(motordelay);
        } else if (strncmp((char*)payload, "MON", 3) == 0) {
          //start motor with specified turns
          std::string s((char*)payload, 3, length-3);
          if (!turning) {
            std::thread(turnmotor, std::stoi(s)).detach();
          }
        }
[weiterer Code]
```

Dieser Code nimmt fünf verschiede Befehle an: `ON` und `OFF` steuert die Lichterkette, ein Hexcode, eingeleitet mit `#`, ändert die Farbe des Lichts, `s1`-`s10` konfigurieren die Geschwindigkeit des Motors, und `MON-12` bis `MON12` lassen den Motor sich drehen, wobei eine Zahl einer Vierteldrehung entspricht. Die Lichter und der Motor werden von seperaten Threads gesteuert, damit der ESP32 während dieser Zeit weiterhin Befehle annehmen kann. Um diese Threads zu handhaben, werden die globalen booleans `lightson` und `turning` verwendet. Falls die LEDs während einer Motorbewegung nicht aktiv sind, fungieren sie auch als Fortschrittsleiste. Hier sind die methoden, die auf den Threads ausgeführt werden.

```
void lightloop() {
  while (lightson) {
  for(int whiteLed = 0; whiteLed < NUM_LEDS; whiteLed = whiteLed + 1) {
    if(!lightson) {
      break;
    }
    leds[whiteLed] = /*0xFFFFFF - */ledcolor; //color is inverted for some reason
    FastLED.show();
    delay(100);
    leds[whiteLed] = CRGB::Black;
    }
  for(int whiteLed = NUM_LEDS-2; whiteLed > 0; whiteLed = whiteLed - 1) {
    if(!lightson) {
      break;
    }
    leds[whiteLed] = /*0xFFFFFF - */ledcolor; //color is inverted for some reason
    FastLED.show();
    delay(100);
    leds[whiteLed] = CRGB::Black;
    }
  }
  FastLED.show();
}

int seq[8][4] = {
        {1, 0, 0, 0},
        {1, 1, 0, 0},
        {0, 1, 0, 0},
        {0, 1, 1, 0}, //Um den Motor zu drehen,
        {0, 0, 1, 0}, //werden seine Pins nach dieser Folge gesetzt.
        {0, 0, 1, 1},
        {0, 0, 0, 1},
        {1, 0, 0, 1} };

void turnmotor(int quarterturns) {
  turning = true;
  int c = 0;
  if (quarterturns < 0) {
  for(int i = 0; i < -128*quarterturns; i++) {
    for (int j = 7; j >= 0; j--) {
      digitalWrite(26, seq[j][0]);
      digitalWrite(25, seq[j][1]);
      digitalWrite(33, seq[j][2]);
      digitalWrite(32, seq[j][3]);
      delay(motordelay);
    }
    if(!lightson) {
      if (i % (-128*quarterturns / 13) == 0) {
        leds[c++] = 0x002000;
        FastLED.show();
      }
    }
```

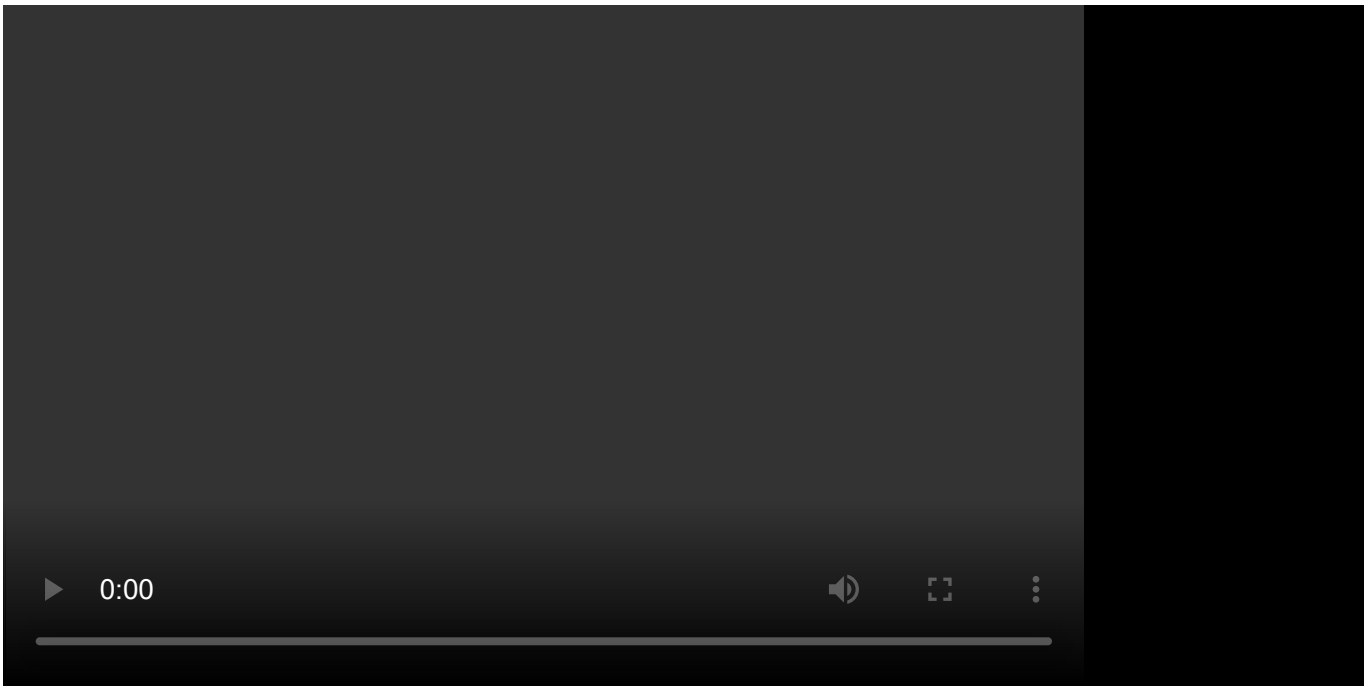
```
  }
  } else {
    for(int i = 0; i < 128*quarterturns; i++) {
    for (int j = 0; j < 8; j++) {
      digitalWrite(MOTOR_PIN_1, seq[j][0]);
      digitalWrite(MOTOR_PIN_2, seq[j][1]);
      digitalWrite(MOTOR_PIN_3, seq[j][2]);
      digitalWrite(MOTOR_PIN_4, seq[j][3]);
      delay(motordelay);
    }
    if(!lightson) {
      if (i % (128*quarterturns / NUM_LEDS) == 0) {
        leds[c++] = 0x002000;
        FastLED.show();
      }
    }
  }
  }
  for(int i = 0; i < NUM_LEDS; i++) {
    leds[i] = CRGB::Black;
  }
  FastLED.show();
  turning = false;
}
```

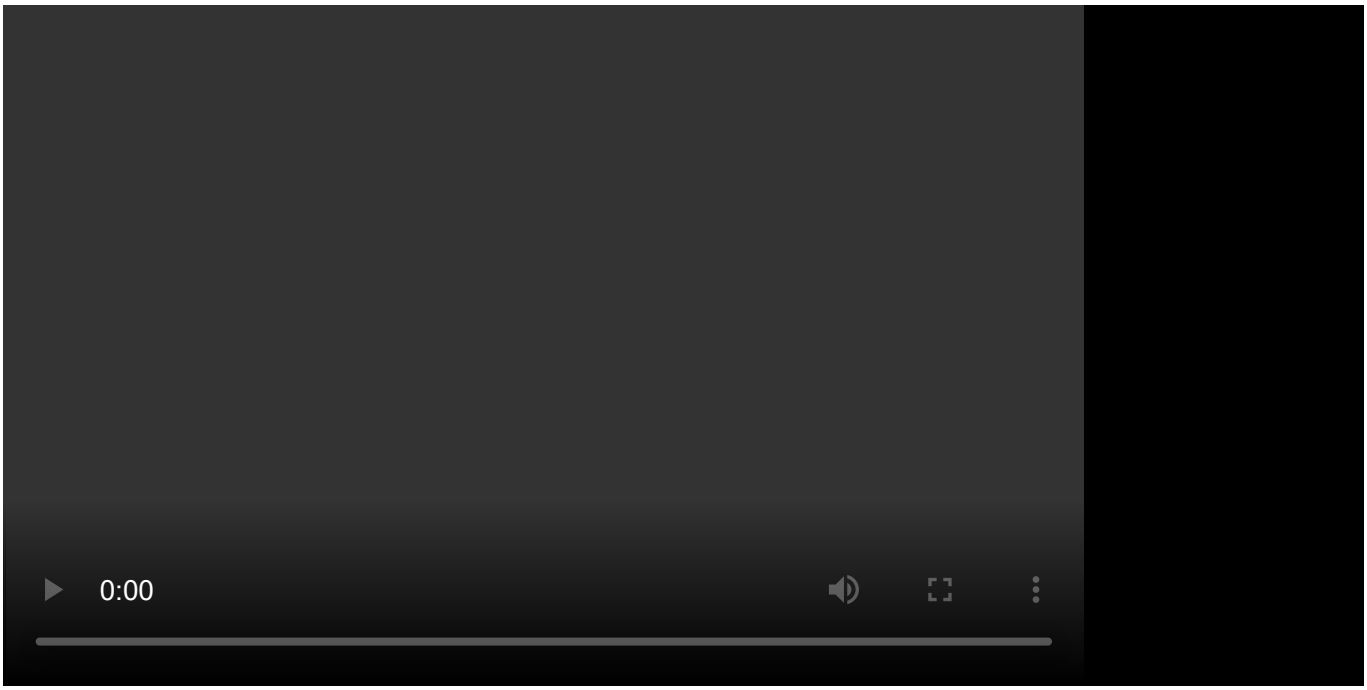
Diese Methoden steuern die Peripheriegeräte durch Ansprechen der GPIO-Pins des ESP32, an die die Geräte angeschlossen sind.

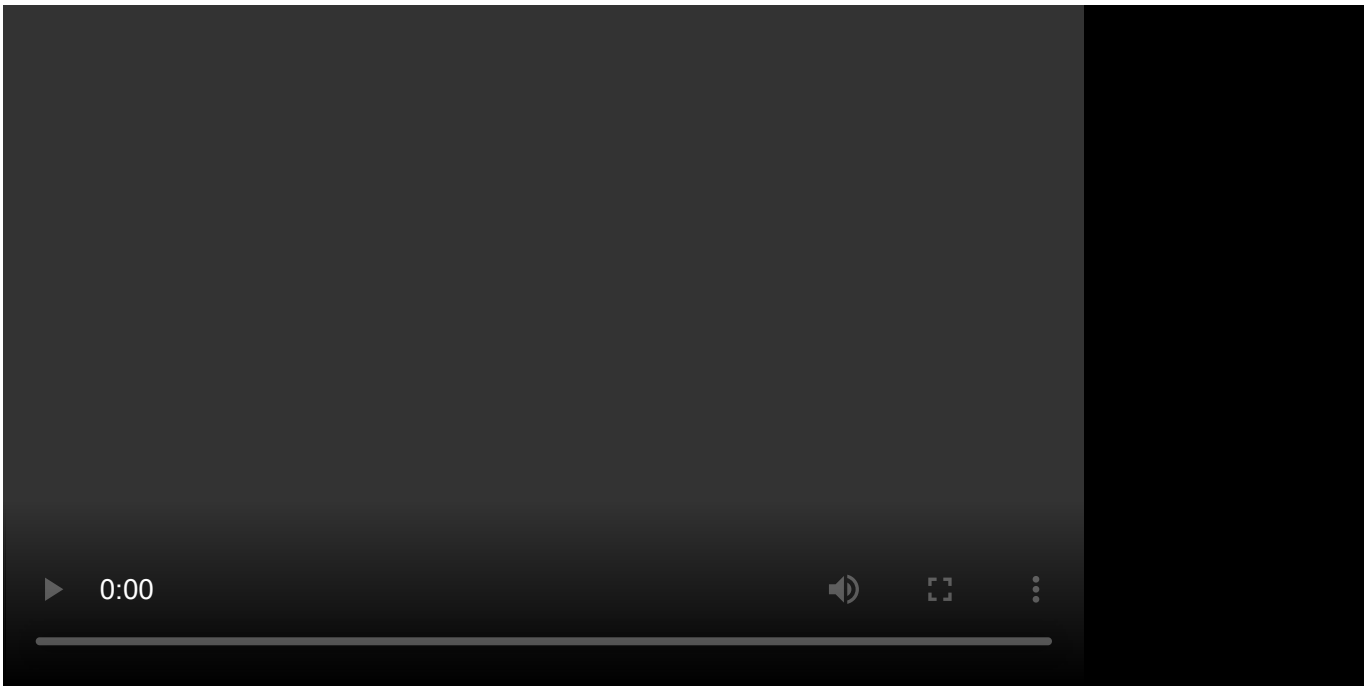
```
#define NUM_LEDS 13 // Wie lang ist die Lichterkette?
#define DATA_PIN 2 //Dieser Pin steuert die LEDs
#define CLOCK_PIN 4 //Dieser Pin ist für manche LED-Ketten erforderlich

#define MOTOR_PIN_1 26 //Diese vier Pins steuern den Motor
#define MOTOR_PIN_2 25
#define MOTOR_PIN_3 33
#define MOTOR_PIN_4 32
```

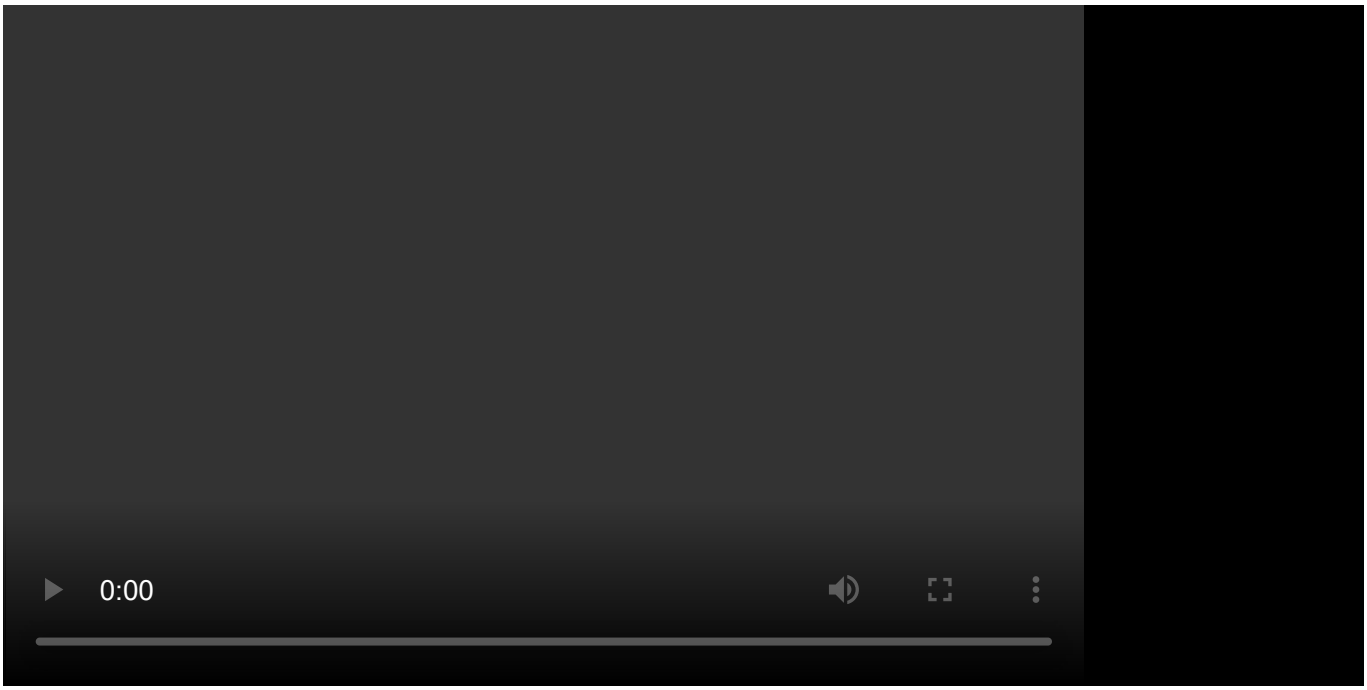
Der LED-Streifen wird über einen einzigen Pin mittels eines uns unbekannten Protokolls angesprochen. Deswegen haben wir ihn über die FastLED-Library gesteuert.

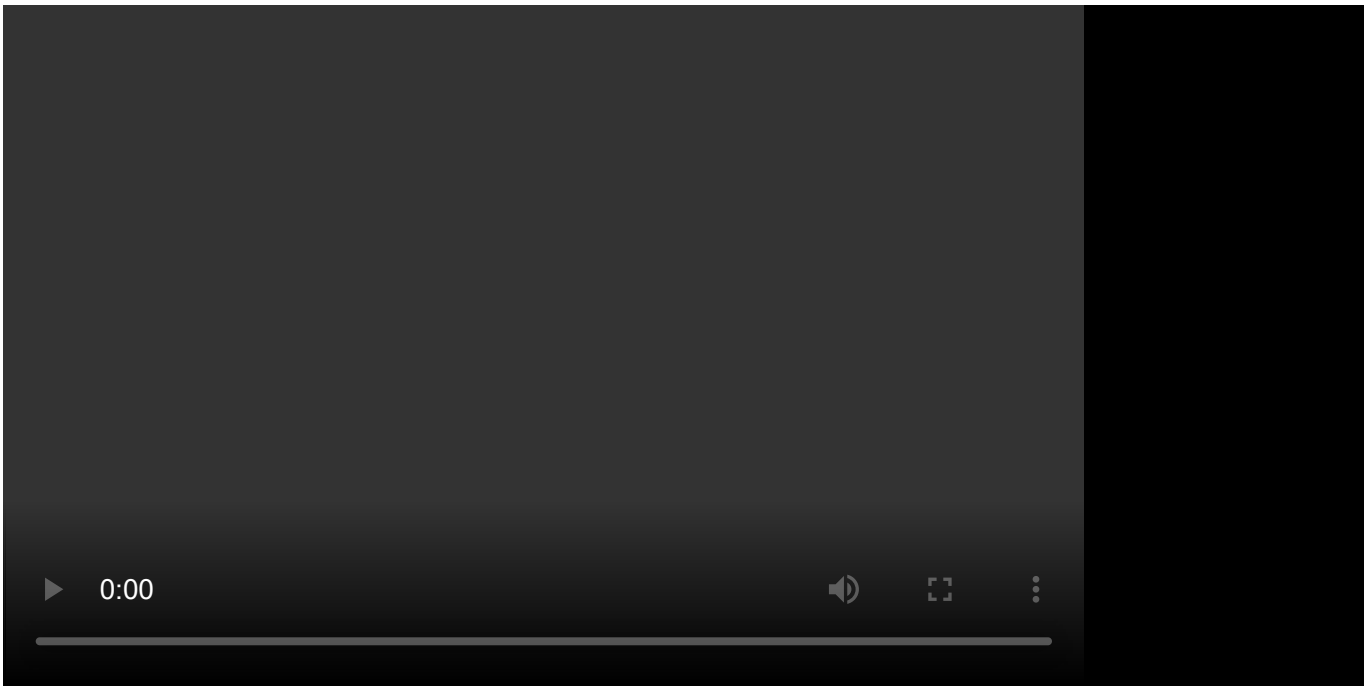
```
[Ausschnitt aus setup]
FastLED.addLeds<WS2811, DATA_PIN, GRB>(leds, NUM_LEDS);
//Aus irgendeinem Grund nimmt unser LED-Streifen Farben im GRB-Format entgegen
[weiterer Code]
```

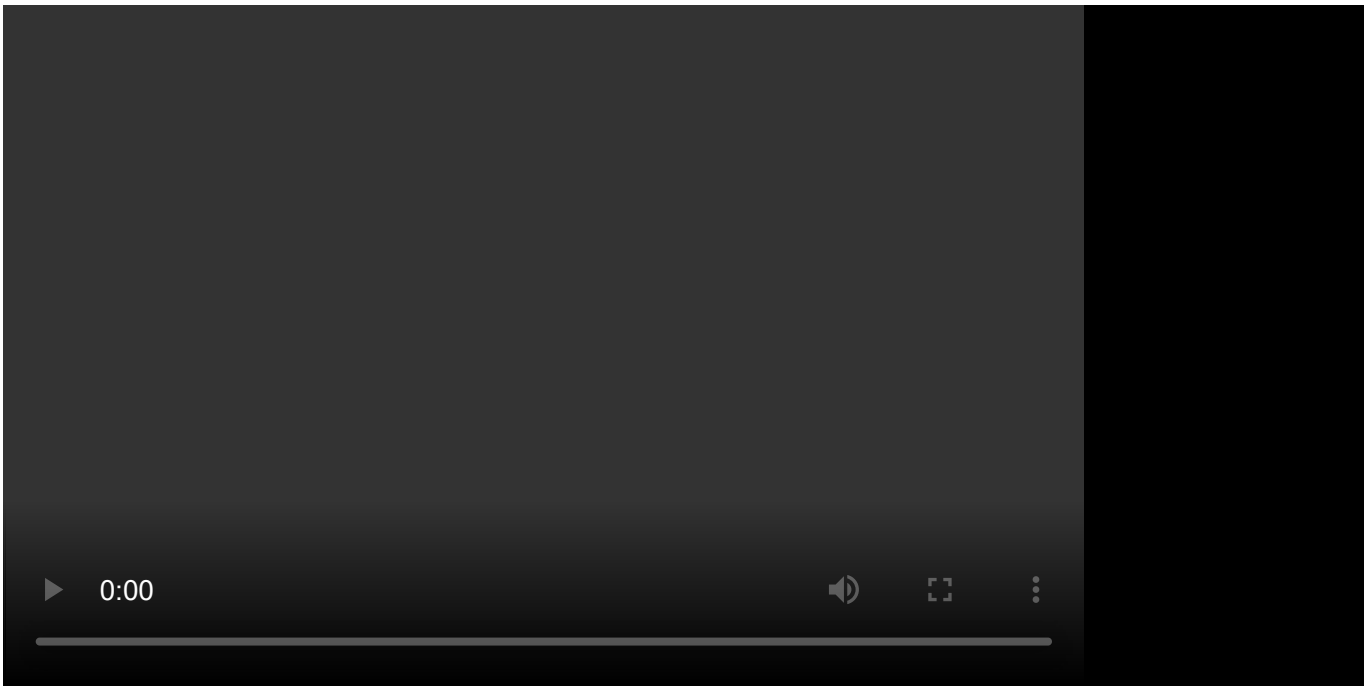

0:00


0:00


0:00


0:00


0:00


0:00

Lautsprecher und Tonerzeugung

Am Abend haben wir noch Lautsprecher ausprobiert, um zu sehen, ob wir Musik spielen können. Wir haben im Internet eine Notentabelle mit Frequenzen gefunden und die Melodie gecodet und abgespielt. Eigentlich gibt es in Arduino eine `tone`-Funktion, die wir verwenden wollten, aber in unserem Test-Projekt funkionierte sie nicht, also haben wir unsere eigene codet, die aber nicht perfekt war.

```
void GPTone(int pin, int frequency, int duration) {
  int period = 1000000L / frequency; // Calculate the period in microseconds
  int pulseWidth = period / 2;       // Square wave: half the period high, half
low
  for (long i = 0; i < duration * 1000L; i += period) {
    digitalWrite(pin, true);
    delayMicroseconds(pulseWidth);
    digitalWrite(pin, false);
    delayMicroseconds(pulseWidth);
  }
}
```

Diese von ChatGPT generierte Methode kann durchaus Töne auf Lautsprechern abspielen, klingt aber nicht so sauber wie die `tone`-Methode von Arudino. Wir haben einen weiteren MQTT-Befehl eingefügt, der ein Musikstück spielt.

```
[neue Variable]
bool musicplaying = false;
[...]
```

```
[Ausschnitt aus mqtt_callback]
 } else if (strncmp((char*)payload, "AON", length) == 0 && !musicplaying) {
          //sands undertale
          Serial.println("Commencing bad time...");
          musicplaying = true;
          std::thread(speakerv1).detach();
          std::thread(speakerv2).detach();
 }
[weiterer Code]
```

Das Musikstück ist zweistimmig und verwendet zwei Lautsprecher.

```
#define SPEAKER_1_PIN 16
#define SPEAKER_2_PIN 17
```

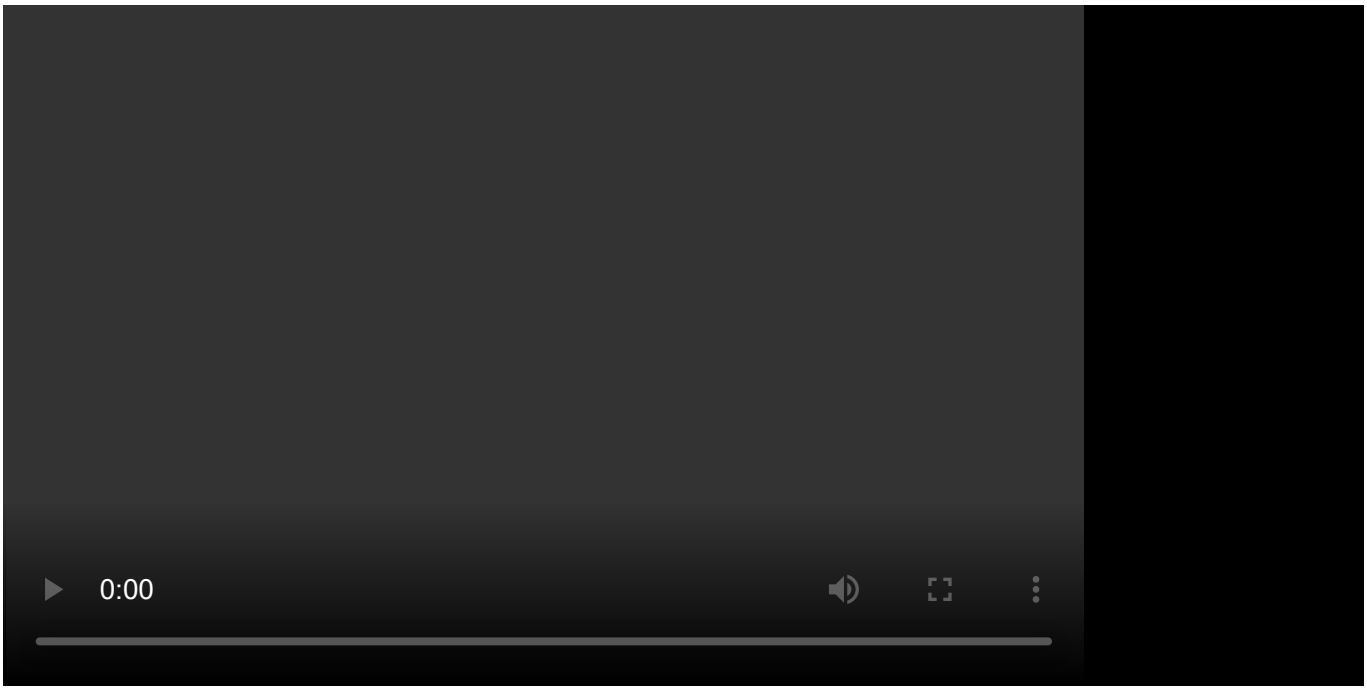
Deswegen werden zwei Threads gestartet; einer pro Lautsprecher. Lautsprecher 1 liest die zu spielenden Noten and Pausen aus zwei großen Arrays:

```
int melody[] = { //the pitch of each note (looked up from a table, rounded)
  294, 294, 587, 440, 415, 392, 349, 294, 349, 392,
  262, 262, 587, 440, 415, 392, 349, 294, 349, 392,
  247, 247, 587, 440, 415, 392, 349, 294, 349, 392,
  233, 233, 587, 440, 415, 392, 349, 294, 349, 392,
};

// the duration of pauses between each note (in milliseconds)
int noteDurations[] = {
  50, 50, 200, 400, 200, 200, 200, 25, 25, 25,
  50, 50, 200, 400, 200, 200, 200, 25, 25, 25,
  50, 50, 200, 400, 200, 200, 200, 25, 25, 25,
  50, 50, 200, 400, 200, 200, 200, 25, 25, 25,
};
void speakerv1() {
  Serial.println("v1 thread");
  for (int i = 0; i < 40; i++) {
    tone(SPEAKER_1_PIN, melody[i], 200);
    delay(noteDurations[i]+200);
  }
}
```

Erstaunlicherweise, anders als im Testprojekt, funkioniert die Arduino-`tone`-Methode einfach! Anders als die ChatGPT-Methode hält sie beim Spielen eines Tons den Programmfluss allerdings nicht auf, weswegen die Dauer des Tones zu der Pause danach addiert wird. Stimme 2 sieht wie folgt aus:

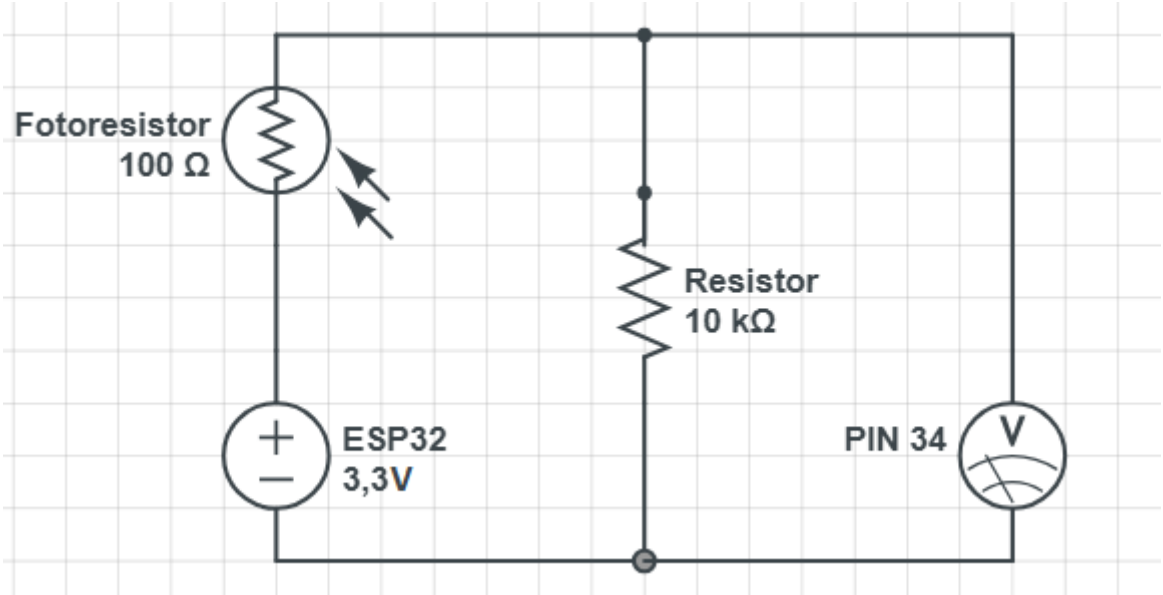
```
void speakerv2() {
  Serial.println("v2 thread");
  GPTone(SPEAKER_2_PIN, 294, 3375);
  GPTone(SPEAKER_2_PIN, 262, 3375);
  GPTone(SPEAKER_2_PIN, 247, 3375);
  GPTone(SPEAKER_2_PIN, 233, 1687);
  GPTone(SPEAKER_2_PIN, 262, 1687);
  musicplaying = false;
}
```

Da diese Stimme nur fünf Töne spielt, ist das Erstellen von Arrays hier weniger sinvoll. Bemerkbar ist auch, dass hier die ChatGPT-Tone-Methode verwendet wird. Die `tone`-Methode von Arduino kann nämlich nur einen Ton gleichzeitig spielen, auch auf verschiedenen Pins, weil sie einen globalen Timer dafür verwendet! Wir sind also wohl oder übel gezwungen, für die zweite Stimme diese Methode zu verwenden.



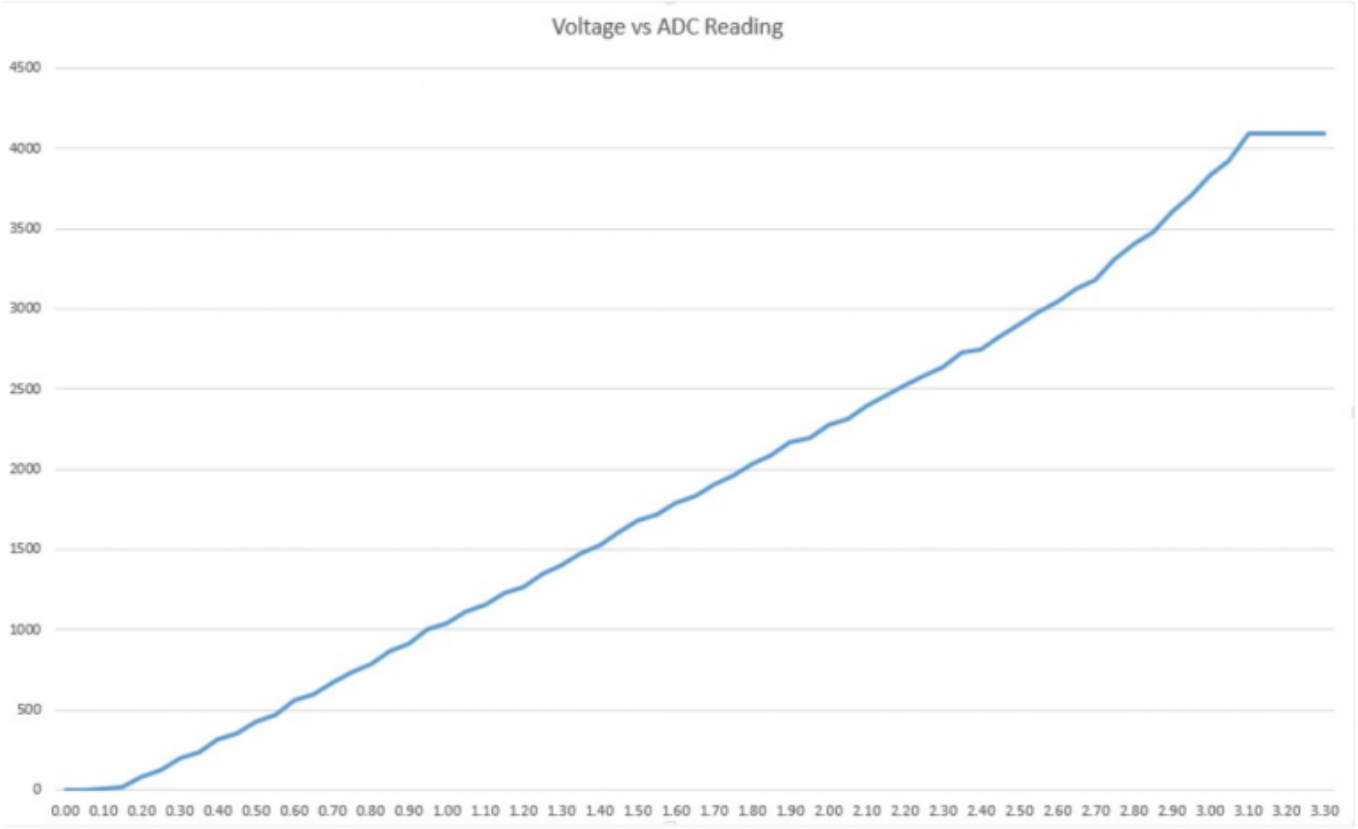
Fotoresistor und Lichtsteuerung

Unser nächster Versuch ist mit einem Fotoresistor, den wir mit unserem Gerät anschließen. Diesen wollen wir dazu verwenden, unter bestimmten Bedingungen Daten an den Home Assistant zu senden. Je nachdem, wie viel Licht der Fotoresistor bekommt, ändert sich sein Widerstand. Bei mehr Licht sinkt er, bei weniger Licht erhöht er sich. Wir schalten den Fotoresistor in Reihe mit einem 10k Ohm Widerstand und messen die Spannung auf Pin 34 des ESP32.




```
#define PHOTORES 34
```

Mittels der `analogRead`-Methode kann die Spannung, die an einem Pin anliegt, relativ genau gemessen werden. Der Rückgabewert der Methode liegt zwischen 0 für 0V und 4095 für 3,3V.



Wir messen die Spannung in der `loop`-Methode, weil sie andauernd ausgeführt wird.

```
[neue Variable]
bool lightpresent;
[...]
```

```
[Ausschnitt aus loop]
if (analogRead(PHOTORES) >= 2000 && !lightpresent) {
    mqttClient.publish("home/esp32/relay/state", "LOFF");
    Serial.println("Room lit up, turning off lights");
    lightpresent = true;
  } else if (analogRead(PHOTORES) < 2000 && lightpresent) {
    mqttClient.publish("home/esp32/relay/state", "LON");
    Serial.println("Room got dark, turning on lights");
    lightpresent = false;
  }
```

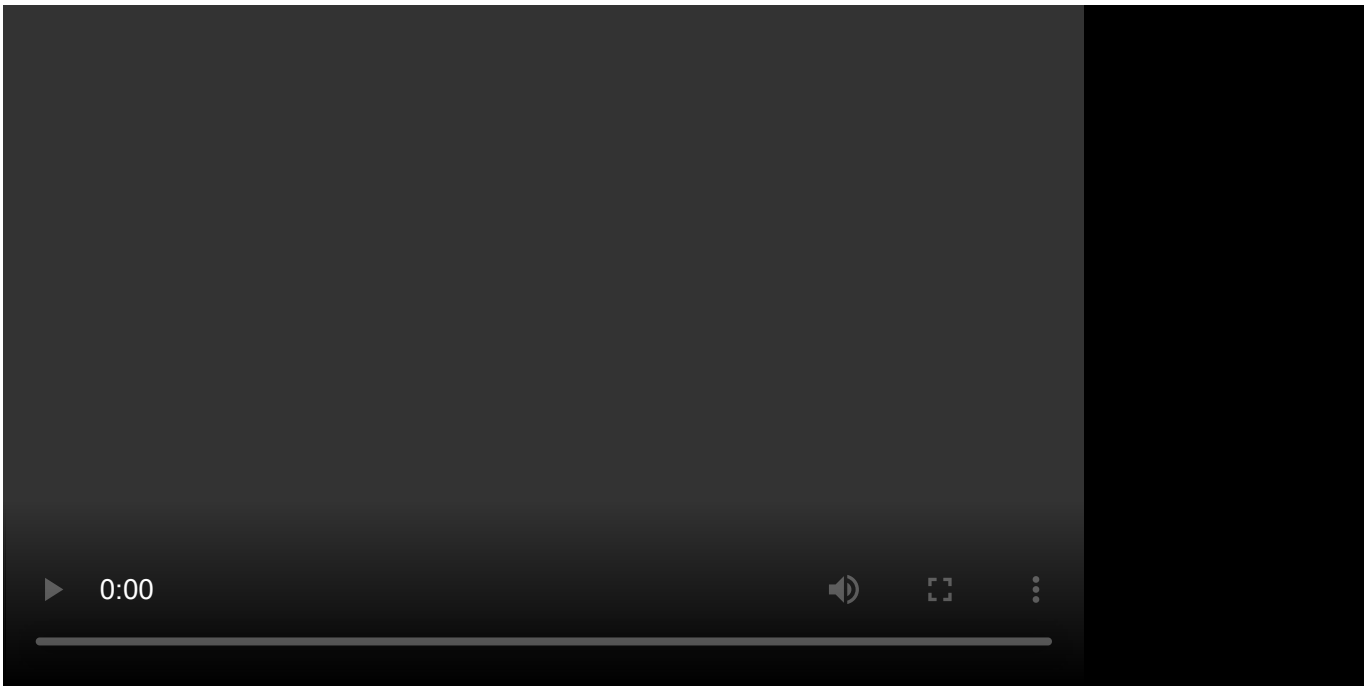
Der Wert von 2000 als Grenze zwischen einer hellen und einer dunklen Umgebung wurde experimentell ermittelt. Sobald sich die Lichtverhältnisse ändern, wird eine Nachricht des Themas `home/esp32/relay/state` gesendet. Der Home Assistant verwendet diese Nachricht um in seinem UI automatisch einen Schalter umzulegen. Dieser Schalter sendet eine Nachricht an einen zweiten ESP32, den wir verbunden haben. Dieser verwendet eine vereinfachte Version unseres Codes und aktiviert und deaktiviert Pin 0 diesen Nachrichten entsprechend.

```
[Ausschnitt aus mqtt_callback des anderen ESP32]
// Handle command topic
if (strcmp(topic, "home/esp32/relay/set2") == 0) {
    if (strncmp((char*)payload, "LON", length) == 0) {
        Serial.println("Turning Lights on");
        digitalWrite(LIGHTS, true);
    } else if (strncmp((char*)payload, "LOFF", length) == 0) {
        Serial.println("Turning lights off");
        digitalWrite(LIGHTS, false);
    } else {
        Serial.print("Unknown command: ");
        Serial.println((char*)payload);
    }
}
```

Anfangs funktionierte das allerdings nicht, weil die ESPs versuchten, sich mit der gleichen ID anzumelden, und einander andauernd aus der Verbindung herauswarfen. Durch das Verwenden verschiedener IDs wird dieses Problem gelöst.

```
[Codeausschnitt des ersten ESP32]
mqttClient.connect("ESP32Main", MQTT_USER, MQTT_PASSWORD)

[Codeaussnitt des zweiten ESP32]
mqttClient.connect("ESP32Lightslave", MQTT_USER, MQTT_PASSWORD)
```

Wir haben zur Demonstration einen Buzzer an Pin 0 des zweiten ESPs angeschlossen. Wenn es also bei ESP32 Nr. 1 dunkel wird, ertönt bei ESP32 Nr. 2 ein Buzzer.

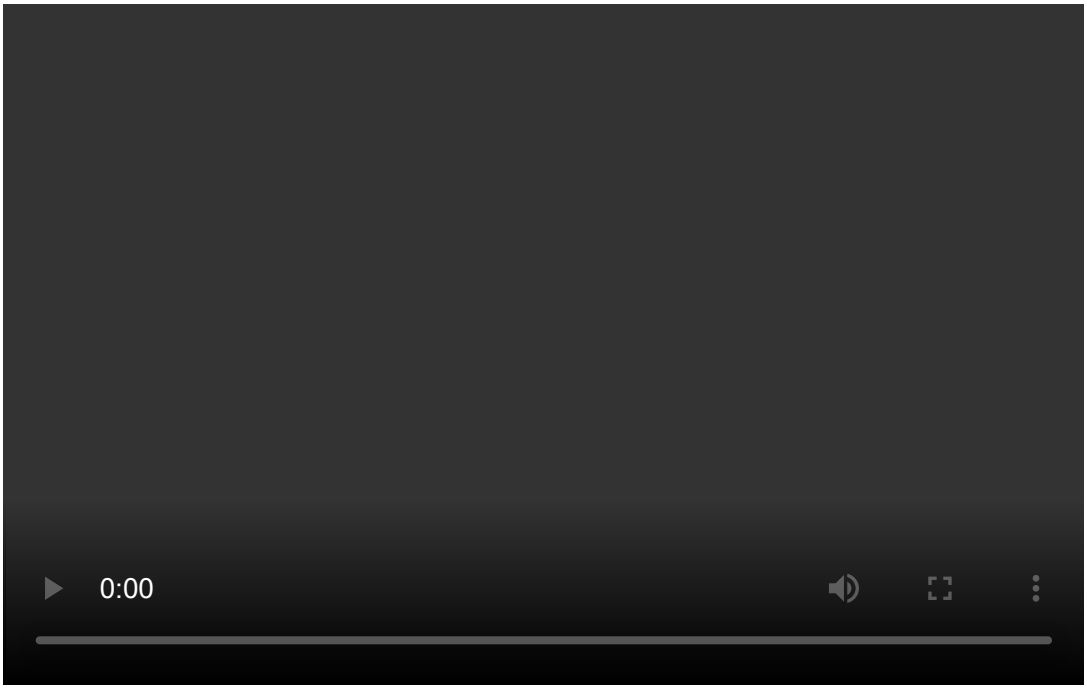


Zusammenfassung

Insgesamt haben wir viele Fortschritte gemacht, aber es gab auch einige Herausforderungen und ganz viele dumme Coding errors. Wir haben gelernt, wie man den ESP32 mit Home Assistant und MQTT integriert, LED-Streifen und Motoren steuert, sowie Tonerzeugung und Lichtsteuerung mit einem Fotoresistor umsetzt. Aus dem Projekt haben wir gelernt, dass man es endlos erweitern könnte und verschiedene Smart-Home Geräte miteinander sprechen lassen kann.

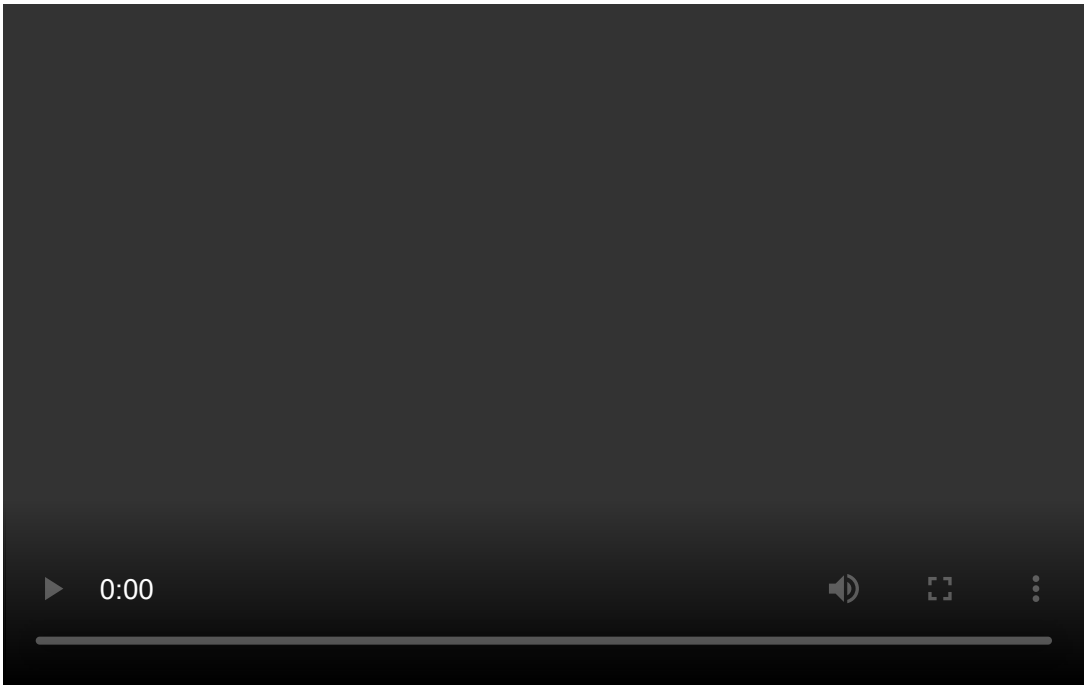
Anhänge

Outtakes:



Your browser does not support the video tag.

Auf der SemCon:



Your browser does not support the video tag.