Projekt: https://github.com/ronaldsieber/ESP32SmartBoard_MqttSensors_BleCfg
Lizenz: MIT
Autor: Ronald Sieber

ESP32SmartBoard_MqttSensors_BleCfg

Dieses Arduino-Projekt wurde als Firmware für das *ESP32SmartBoard* (siehe Hardware Projekt *<ESP32SmartBoard_PCB>)* entwickelt. Es liest die Werte des Temperatur- und Luftfeuchtesensors (DHT22), des CO2-Sensors (MH-Z19) sowie die übrige Peripherie des Boards aus und überträgt die Informationen als MQTT-Nachrichten an einen MQTT-Broker. Über eingehende MQTT-Nachrichten lassen sich Ausgänge setzen und das Board konfigurieren.

Zudem veranschaulicht dieses Projekt die Integration des in *<ESP32BleConfig>* implementierten Bluetooth Configuration Frameworks in eine reale ESP32/Arduino-Anwendung.

Die Hardware-Konfiguration (Port-Definitionen) sind auf das *ESP32SmartBoard* (siehe Hardware Projekt *<ESP32SmartBoard_PCB>*) abgestimmt.

Historie

Version 1

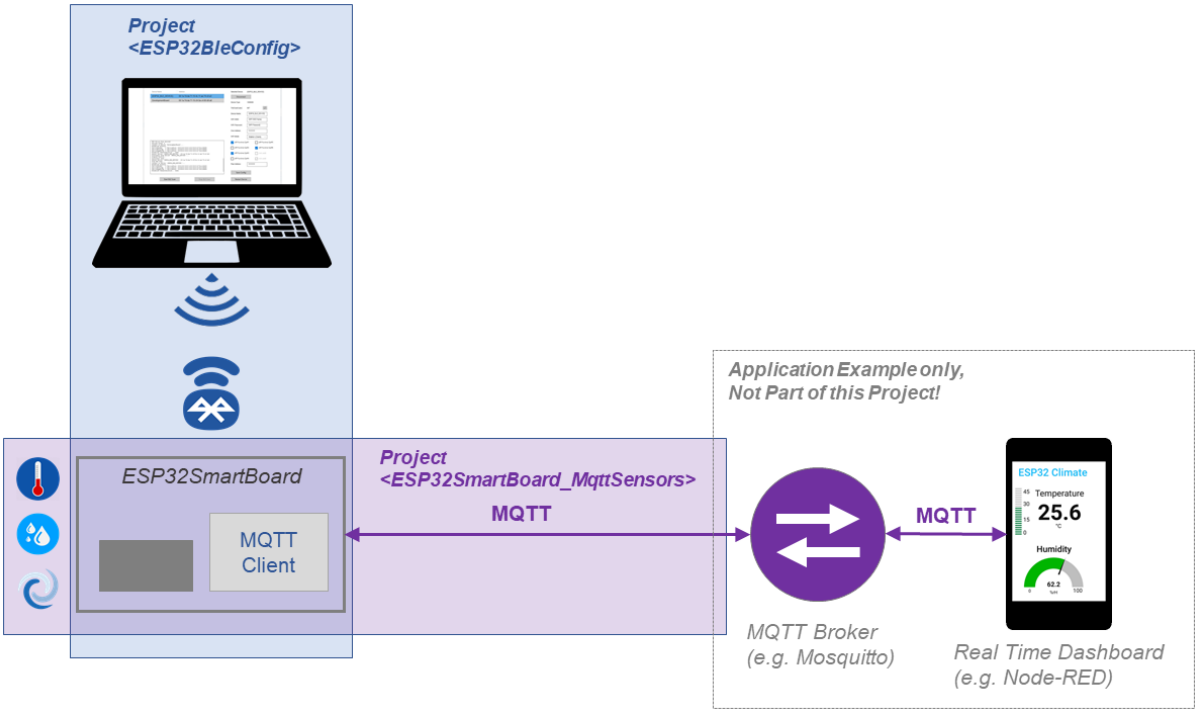
Initiale MQTT-basierte Firmware *<ESP32SmartBoard_MqttSensors>* für das *ESP32SmartBoard* (siehe Hardware Projekt *<ESP32SmartBoard_PCB>*).

Version 2

Symbiose der beiden autonomen Projekte *<ESP32SmartBoard_MqttSensors>* und *<ESP32BleConfig>* zu einem gemeinsamen Projekt. Dadurch die Einbindung des Bluetooth Konfigurations-Framework aus dem Projekt *<ESP32BleConfig>* entfallen die in Version 1 *<ESP32SmartBoard_MqttSensors>* beschriebenen Anpassungen im Quellcode für folgende Bereiche:

- WLAN-Konfiguration
- MQTT-Konfiguration
- Applikations-Konfiguration

Diese Konfigurationseinstellungen erfolgen nun über Bluetooth mit Hilfe des grafischen Konfigurations-Tools aus dem Projekt *<ESP32BleConfig>*.



[Project Components]

Version 3

In der aktuellen Version wurde die Software in wichtigen Bereichen überarbeitet und erweitert:

- Beim Systemstart sendet das Board eine Bootup-Nachricht im JSON-Format mit der aktuellen Konfiguration (siehe Abschnitt *"Bootup Datenblock"* unten)

- Statt in separaten Nachrichten für die jeweiligen Sensoren werden nun alle Sensordaten in einer einzigen Nachricht im JSON-Format versendet (siehe Abschnitt *"Sensor Datenblock"* unten)

- Die Ausführung der Software wird durch einen Watchdog Monitor überwacht, optional kann hierbei auch das Acknowledgement der MQTT-Paketnummern mit einbezogen werden (siehe Abschnitt *"Watchdog Monitor"* unten)

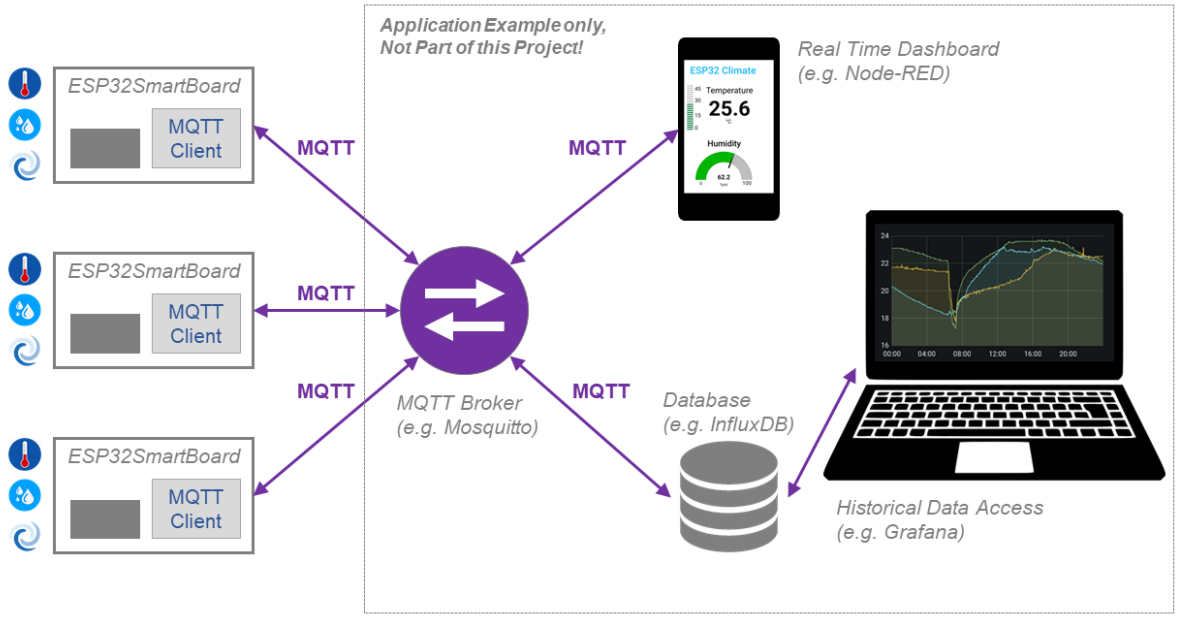
Zudem hat sich gezeigt, dass der 1-Wire Sensor DHT22 mit leichtem Rauschen auf industrielle Schaltnetzteile zur Versorgung des Boards reagiert. Daher werden die Messwerte von Temperatur und Luftfeuchtigkeit jetzt über den Simple Moving Average Filter aus dem Projekt <*SimpleMovingAverage*> über die Anzahl *APP_SMA_DHT_SAMPLE_WINDOW_SIZE* bzw. *APP_SMA_DHT_SAMPLE_WINDOW_SIZE* gemittelt ("geglättet“).

Projekt-Überblick

Beim Systemstart verbindet sich das *ESP32SmartBoard* mit einem MQTT-Broker. Anschließend sendet ("published") das Board zyklisch die Werte des Temperatur- und Luftfeuchtesensors (DHT22) sowie des CO2-Sensors (MH-Z19). Die beiden Taster KEY0 und KEY1 werden event-gesteuert übertragen. Per "Subscribe" empfängt das Board Daten vom Broker und lässt sich so zur Laufzeit steuern und konfigurieren. Ausführliche Details beschreibt der Abschnitt *"MQTT-Messages des ESP32SmartBoard"* weiter unten.

Die Konfiguration des Boards (WLAN-Setup, MQTT-Broker, Laufzeit-Konfiguration) erfolgt über Bluetooth mit Hilfe des grafischen Konfigurations-Tools *"Esp32ConfigUwp"* aus dem Projekt *<ESP32BleConfig>* (siehe Abschnitt *"Board-Konfiguration"* weiter unten).

In einem typischen Szenario sind mehrere *ESP32SmartBoards* in verschiedenen Räumen einer Wohnung bzw. in mehreren Büros oder Klassenräumen installiert. Alle diese Boards übertragen ihre Daten per MQTT an einen gemeinsamen Broker, der alle Sensorwerte in einer zentralen Instanz bereitstellt. Hierauf greifen dann weitere Clients zu, die beispielsweise in Form von Dashboards die Momentanwerte von Temperatur oder CO2-Level verschiedener Räume anzeigen (z.B. Node-RED) oder die Sensorwerte in einer Datenbank sammeln (z.B. InfluxDB). Die als Zeitreihen gespeicherten Messdaten lassen sich dann wiederum als zeitliche Verläufe der Sensorwerte in Charts grafisch darstellen und auswerten (z.B. Grafana).



[Project Overview]

Das Projekt *ESP32SmartBoard_MqttSensors* enthält von dem beschriebenen Setup nur die MQTT-basierte Firmware für das *ESP32SmartBoard*. Broker, Datenbank, Dashboards usw. sind nicht Bestandteil dieses Projektes.

Ergänzend zu diesem Arduino-Projekt setzt *<ESP32SmartBoard_NodeRED>* ein Node-RED basiertes Dashboard zur Anzeige der Sensordaten und zur Laufzeitkonfiguration des Boards um.

Zur Inbetriebnahme und Diagnose eignet sich das Open-Source Tool 'MQTT Explorer' (https://mqtt-explorer.com/).

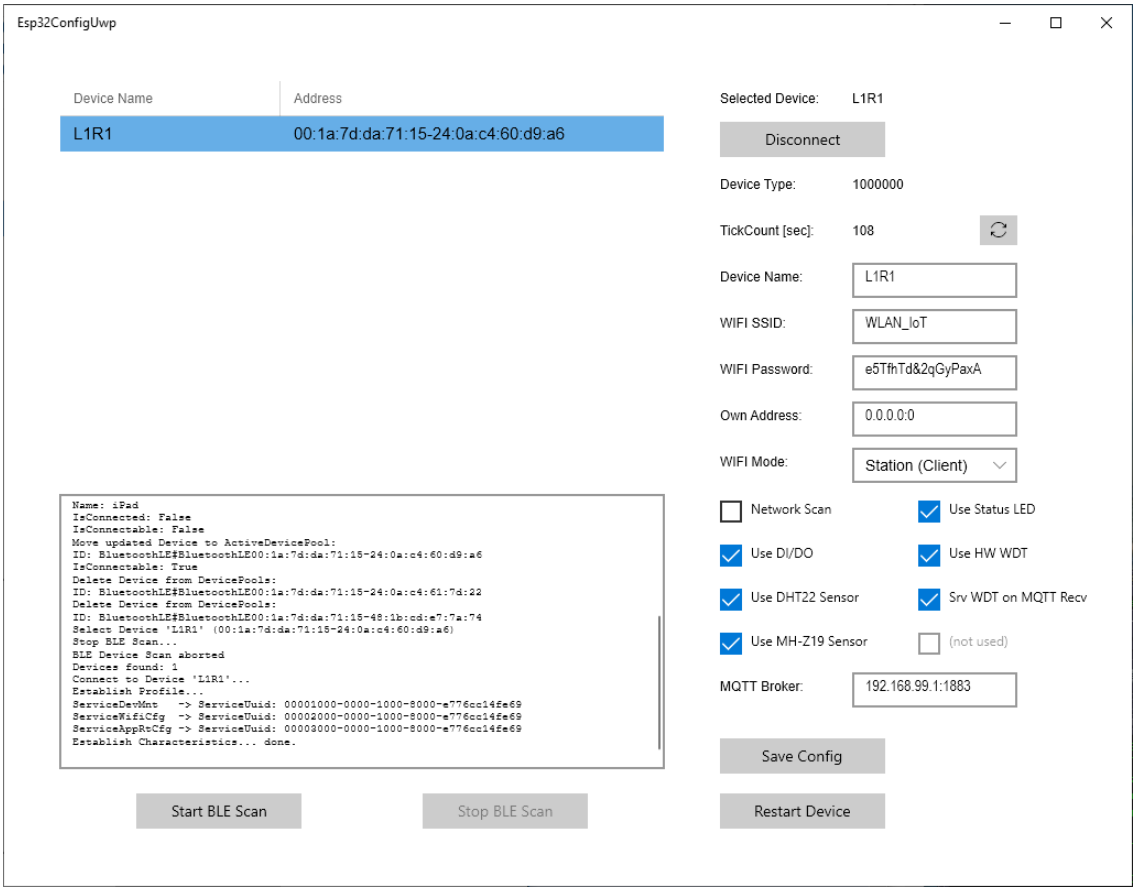
Board-Konfiguration

Die Konfiguration des Boards (WLAN-Setup, MQTT-Broker, Laufzeit-Konfiguration) erfolgt über Bluetooth mit Hilfe des grafischen Konfigurations-Tools *"Esp32ConfigUwp"* aus dem Projekt *<ESP32BleConfig>*.

Um den Bluetooth-Konfigurationsmodus auf dem *ESP32SmartBoard* zu aktivieren, sind folgende Schritte notwendig:

1. BLE_CFG auf dem *ESP32SmartBoard* drücken und durchgehend bis Schritt 2 gedrückt halten
2. Reset am ESP32DevKit drücken und wieder loslassen
3. BLE_CFG loslassen

Anschließend kann das *ESP32SmartBoard* mit dem Konfigurations-Tool *"Esp32ConfigUwp"* verbunden und von diesem konfiguriert werden. Die dazu notwendigen Schritte beschreibt *<https://github.com/ronaldsieber/ESP32BleConfig/blob/main/README.md>* im Detail.



[Esp32ConfigUwp]

Applikations-Konfiguration Sektion

Am Anfang des Sketches befindet sich folgender Konfigurations-Abschnitt mit Standardeinstellungen:

```
const int  DEFAULT_CFG_ENABLE_NETWORK_SCAN  = 1;
const int  DEFAULT_CFG_ENABLE_DI_DO         = 1;
const int  DEFAULT_CFG_ENABLE_DHT_SENSOR    = 1;
const int  DEFAULT_CFG_ENABLE_MHZ_SENSOR    = 1;
const int  DEFAULT_CFG_ENABLE_STATUS_LED    = 1;
const int  DEFAULT_CFG_ENABLE_HW_WDT        = 1;
const int  DEFAULT_CFG_SRV_WDT_ON_MQTT_RECV = 1;
```

Hierüber lässt sich zur Laufzeit die Ausführung der zugehörigen Code Abschnitte freischalten (=1) bzw. sperren (=0). Das vermeidet das Auftreten von Laufzeitfehlern bei Boards, auf denen nicht alle Komponenten bestückt sind (insbesondere, wenn der DHT22 oder der MH-Z19 nicht vorhanden sind).

Diese Standardeinstellungen können über die oben beschriebene Bluetooth-basierte Konfiguration beliebig überschrieben und angepasst werden.

Bootup Datenblock

Der Bootup Datenblock wird einmalig nach dem Systemstart gesendet und enthält grundlegende Informationen zum System:

```
{
  "PacketNum": 0,
  "FirmwareVer": "3.00",
  "BootReason": 3,
  "IP": "192.168.69.251",
  "ChipID": "240AC460D9A4",
  "DataPackCycleTm": 60,
  "CfgNetworkScan": 0,
  "CfgDiDo": 1,
  "CfgTmpHumSensor": 1,
  "CfgCO2Sensor": 1,
  "CfgStatusLed": 1,
  "CfgHwWdt": 1,
  "CfgSrvWdtOnMqtt": 1
}
```

PacketNum *(Integer)*: fortlaufende Paketnummer seit Systemstart (0 = Bootup)
FirmwareVer *(String)*: Firmware Versionsnummer im Format "Ver.Rev"
BootReason *(Integer)*: Ursache des letzten Systemstarts (Returnwert von `esp_reset_reason()`)
1 = *ESP_RST_POWERON* (Power-on)
3 = *ESP_RST_SW* (Software Reset durch Aufruf der Funktion `esp_restart()`)
4 = *ESP_RST_PANIC* (Software Reset infolge Exception/Panic)
7 = *ESP_RST_WDT* (Watchdog Reset)
9 = ESP_RST_BROWNOUT (Brownout Reset)
(vollständige Übersicht: siehe `esp_reset_reason_t` in *"esp_system.h"*)
IP *(String)*: Die vom DHCP-Server beim WLAN Connect zugewiesene IP-Adresse
ChipID *(String)*: Chip-ID des ESP32 (basierend auf Returnwert von `ESP.getEfuseMac()`)
DataPackCycleTm *(Integer)*: konfiguriertes Sendeintervall für Sensor Datenblock in [sec]
CfgNetworkScan *(Integer)*: Konfigurations-Option *CFG_ENABLE_NETWORK_SCAN*
CfgDiDo *(Integer)*: Konfigurations-Option *CFG_ENABLE_DI_DO*
CfgTmpHumSensor *(Integer)*: Konfigurations-Option *CFG_ENABLE_DHT_SENSOR*
CfgCO2Sensor *(Integer)*: Konfigurations-Option *CFG_ENABLE_MHZ_SENSOR*
CfgStatusLed *(Integer)*: Konfigurations-Option *CFG_ENABLE_STATUS_LED*
CfgHwWdt *(Integer)*: Konfigurations-Option *CFG_ENABLE_HW_WDT*
CfgSrvWdtOnMqtt *(Integer)*: Konfigurations-Option *CFG_SRV_WDT_ON_MQTT_RECV*

Sensor Datenblock

Das Paket wird zyklisch mit dem zur Laufzeit durch die Variable `ui32DataPackCycleTm_g` definierten Sendeintervall übertragen (Default-Wert: `DEFAULT_DATA_PACKET_PUB_CYCLE_TIME`). Das Intervall für das zyklische Senden kann zur Laufzeit über das Topic *"SmBrd/<DevID>/Settings/DataPackPubCycleTime"* modifiziert werden. Außerdem wird das Paket event-gesteuert bei Änderung eines der beiden Taster KEY0 und KEY1 übertragen.

```
{
  "PacketNum": 168,
  "MainLoopCycle": 19779,
  "Uptime": 10356,
  "NetErrorLevel": 0,
  "Key0": 0,
  "Key0Chng": 0,
  "Key1": 0,
  "Key1Chng": 0,
  "Temperature": 21.1,
  "Humidity": 51.1,
  "Co2Value": 826,
  "Co2SensTemp": 26
}
```

PacketNum *(Integer)*: fortlaufende Paketnummer seit Systemstart (0 = Bootup)
MainLoopCycle *(Integer)*: Anzahl der `main()` Loops seit Systemstart
Uptime *(Integer)*: Uptime des Systems in [sec]
NetErrorLevel *(Integer)*: Diagnose, Fehlerwert für MQTT-Übertragung
Bei einem MQTT-Übertragungsfehler (`PubSubClient_g.publish() != true`) wird der Fehlerwert um den Betrag 2 erhöht, nach einer fehlerfreien MQTT-Übertragungsfehler wird der Wert wieder um den Betrag 1 vermindert
Key0 *(Integer)*: [0|1] - aktueller Zustand des Taster KEY0
Key0Chng *(Integer)*: [0|1] - Zustand des Taster KEY0 geändert bzw. unverändert
Key1 *(Integer)*: [0|1] - aktueller Zustand des Taster KEY1
Key1Chng *(Integer)*: [0|1] - Zustand des Taster KEY1 geändert bzw. unverändert
Temperature *(float)*: Umgebungstemperatur in [°C]
Humidity *(float)*: Relative Luftfeuchtigkeit der Umgebung in [%]
Co2Value *(Integer)*: CO2-Messwert der Umgebung in [ppm]
Co2SensTemp *(Integer)*: Temperatur innerhalb des CO2-Sensors

MQTT Client Basics

Der *ESP32SmartBoard* Sketch nutz die Library "*Arduino Client for MQTT"* (https://github.com/knolleary/pubsubclient) für die MQTT-Kommunikation mit dem Broker. Diese Bibliothek kapselt die gesamte Protokoll-Implementierung ist daher für den Anwender sehr einfach und komfortabel nutzbar:

Das Laufzeit-Objekt wird direkt über den Konstruktor erstellt:

```
PubSubClient  PubSubClient(WiFiClient);
```

Um dem *ESP32SmartBoard* den zu nutzenden Broker bekannt zu geben, sind die IP-Adresse und Port des Servers zu setzen:

```
PubSubClient.setServer(pszMqttServer, iMqttPort);
```

Damit der Client auf dem *ESP32SmartBoard* Nachrichten via Subscribe vom Broker empfangen kann, muss ein entsprechender Callback Handler registriert werden:

```
PubSubClient.setCallback(pfnMqttSubCallback);
```

Um den lokalen Client auf dem *ESP32SmartBoard* mit dem Broker zu verbinden, sind mindestens eine eindeutige ClientID sowie die Anmeldedaten in Form von Username und Passwort erforderlich. Die ClientID wird dabei von der MAC-Adresse des Boards abgeleitet, wodurch sichergestellt ist, dass jedes Board eine eindeutige ID verwendet. Username und Passwort sind auf Client-Seite anzugeben, ob diese aber tatsächlich ausgewertet werden, hängt von der Konfiguration des Brokers ab. Viele Broker werden offen betrieben, so dass die verwendeten Anmeldedaten beliebig gewählt werden können. Bei Brokern mit aktiver Authentifizierung sind die vom Server-Betreiber zugewiesenen Nutzerdaten zu verwenden.

Optional können beim Verbindungsaufbau Topic und Payload einer "LastWill" Message definiert werden. Der Broker speichert diese Nachricht zunächst und published sie dann später in dem Moment, in dem die Verbindung zum Client abbricht. Damit kann der Client beim Verbindungsaufbau selber festlegen, mit welcher Message der Broker alle anderen Teilnehmer über den Abbruch der Verbindung zwischen Client und Broker informiert:

```
PubSubClient.connect(strClientId.c_str(),           // ClientID
                     pszMqttUser,                   // Username
                     pszMqttPassword,               // Password
                     pszSupervisorTopic,            // LastWillTopic
                     0,                             // LastWillQoS
                     1,                             // LastWillRetain
                     strPayloadMsgGotLost.c_str(),  // LastWillMessage
                     true);                         // CleanSession
```

Zum Publishen einer Nachricht sind deren Topic und Payload anzugeben. Eine als "Retain" gekennzeichnete Nachricht wird vom Broker zwischengespeichert. Sie kann damit auch dann noch einem Client zugestellt werden, wenn dieser das betreffende Topic erst nach dem Publizieren der Nachricht subscribed. Damit können andere Clients bei ihrer Anmeldung am Broker auf den jeweils aktuellen Status ihrer Partner synchronisiert werden:

```
PubSubClient.publish(pszSupervisorTopic,                    // Topic
                     strPayloadMsgConnect.c_str(),          // Payload
                     1);                                    // Retain
```

Da die Bibliothek keinen eigenen Scheduler verwendet, ist sie darauf angewiesen, regelmäßig mit CPU-Zeit versorgt zu werden:

```
PubSubClient.loop();
```

Der aktuelle Verbindungsstatus zum Broker kann zur Laufzeit abgefragt werden:

```
PubSubClient.state();
```

Individualisierung der MQTT-Topics zur Laufzeit

In einem typischen Szenario verbinden sich mehrere *ESP32SmartBoards* zu einem gemeinsamen Broker. Um die Daten den einzelnen Boards zuordnen zu können, verwendet jedes Board individualisierte Topics. Diese Individualisierung erfolgt erst zur Laufzeit, so dass für alle Boards derselbe Sketch Source verwendet werden kann.

Die generischen Topic-Templates sind in statischen String-Arrays definiert:

```
const char*  MQTT_SUBSCRIBE_TOPIC_LIST_TEMPLATE[] =
{
    "SmBrd/<%>/Settings/Heartbeat",             // "<%>" will be replaced by DevID
    "SmBrd/<%>/Settings/LedBarIndicator",       // "<%>" will be replaced by DevID
    "SmBrd/<%>/Settings/DataPackPubCycleTime", // "<%>" will be replaced by DevID
    "SmBrd/<%>/Settings/PrintSensorVal",        // "<%>" will be replaced by DevID
    "SmBrd/<%>/Settings/PrintMqttDataProc",     // "<%>" will be replaced by DevID
    "SmBrd/<%>/OutData/LedBar",                 // "<%>" will be replaced by DevID
    "SmBrd/<%>/OutData/LedBarInv",              // "<%>" will be replaced by DevID
    "SmBrd/<%>/OutData/Led"                     // "<%>" will be replaced by DevID
    "SmBrd/<%>/Ack/PacketNum"                   // "<%>" will be replaced by DevID
};

const char*  MQTT_PUBLISH_TOPIC_LIST_TEMPLATE[] =
{
    "SmBrd/<%>/Data/Bootup",                    // "<%>" will be replaced by DevID
    "SmBrd/<%>/Data/StData",                    // "<%>" will be replaced by DevID
};
```

Die board-spezifische Individualisierung realisiert die Funktion `MqttBuildTopicFromTemplate()`. Die individualisierten Stings werden folgenden beiden Arrays zugewiesen:

```
String  astrMqttSubscribeTopicList_g[ARRAY_SIZE(MQTT_SUBSCRIBE_TOPIC_LIST_TEMPLATE)];
String  astrMqttPublishTopicList_g[ARRAY_SIZE(MQTT_PUBLISH_TOPIC_LIST_TEMPLATE)];
```

Beispielsweise besitzt das Topic zum Publishen des Sensor-Datenblocks folgendes generisches Format:

```
"SmBrd/<DevID>/Data/StData"
```

Zur Individualisierung wird der Substring `"<DevID>"` board-spezifisch ersetzt. Primär ist dies der über die oben beschriebene Bluetooth-basierte Konfiguration festgelegte Gerätename (Feld *"Device Name"*), der zur Laufzeit in `AppCfgData_g.m_szDevMntDevName` steht, z.B.:

```
"SmBrd/Bedroom/Data/StData"
```

Falls über das Konfigurations-Tools kein Gerätename spezifiziert wurde (leerer String), wird stattdessen der mittels Konstante `MQTT_DEVICE_ID` definierte Name verwendet. Ist diese auf NULL gesetzt, wird schlussendlich die aus der MAC-Adresse abgeleitete individuelle Board-ID genutzt, z.B.:

```
"SmBrd/246F2822A4B8/Data/StData"
```

MQTT-Messages des ESP32SmartBoard

Das *ESP32SmartBoards* kommuniziert über die im Folgenden beschriebene MQTT-Messages mit dem Broker. Sowohl für Topics als auch für Payload werden ausschließlich Strings als Datentyp verwendet.

Typ: Publish
Topic: SmBrd_Supervisor
Payload: "CI=<ClientId>, IP=<LocalIP>, RC=[N][I][T][C][L][W][A], ST=[Connected|GotLost]"

Diese Message ist eine Diagnose-Nachricht. Sie wird einmalig als erste Nachricht nach dem erfolgreichen Verbindungsaufbau zum Broker gesendet. Sie beinhaltet die ClientID, die IP-Adresse sowie die Laufzeitkonfiguration des Boards. Dieselbe Nachricht wird auch als "LastWill" verwendet. Der Status "Connected" bzw. "GotLost" zeigt Verbindungsaufbau bzw. -abbruch an.

Der zur Beschreibung der aktuellen Laufzeitkonfiguration ("RC=") verwendete Term enthält je ein Symbol für jeden zur Laufzeit aktivierten Code-Abschnitt:

```
N :   CFG_ENABLE_NETWORK_SCAN  == 1   // (N)etworkScan
I :   CFG_ENABLE_DI_DO         == 1   // (I)/O
T :   CFG_ENABLE_DHT_SENSOR    == 1   // (T)emperature+Humidity
C :   CFG_ENABLE_MHZ_SENSOR    == 1   // (C)O2
L :   CFG_ENABLE_STATUS_LED    == 1   // Status(L)ed
W :   CFG_ENABLE_HW_WDT        == 1   // (W)atchdog
A :   CFG_SRV_WDT_ON_MQTT_RECV == 1   // Serve Watchdog on (A)CK
```

Typ: Publish
Topic: SmBrd/<DevID>/Data/Bootup
Payload: <JsonBootupPacket>

Das *JsonBootupPacket* ist im Abschnitt *"Bootup Datenblock"* beschrieben. Das Paket wird einmalig nach dem Systemstart übertragen.

Typ: Publish
Topic: SmBrd/<DevID>/Data/StData
Payload: <JsonDataPacket>

Das *JsonDataPacket* ist im Abschnitt *"Sensor Datenblock"* beschrieben. Das Paket wird zyklisch mit dem zur Laufzeit durch die Variable `ui32DataPackCycleTm_g` definierten Sendeintervall übertragen (Default-Wert: `DEFAULT_DATA_PACKET_PUB_CYCLE_TIME`). Das Sendeintervall kann zur Laufzeit über das Topic *"SmBrd/<DevID>/Settings/DataPackPubCycleTime"* modifiziert werden (siehe unten).

Typ: Subscribe
Topic: SmBrd/<DevID>/Settings/Heartbeat
Payload: Payload=['0'|'1']

Setzt den Heartbeat-Modus (Blinken der blauen LED auf dem ESP32DevKit):
0 = off
1 = on

Den Default-Wert definiert die Konstante `DEFAULT_STATUS_LED_HEARTBEAT` im Sketch.

Typ: Subscribe
Topic: SmBrd/<DevID>/Settings/LedBarIndicator
Payload: Payload=['0'|'1'|'2'|'3']

Wählt die Datenquelle für den LED Bar Indikator aus:
0 = kLedBarNone
1 = kLedBarDht22Temperature
2 = kLedBarDht22Humidity
3 = kLedBarMhz19Co2Level

Den Default-Wert definiert die Konstante `DEFAULT_LED_BAR_INDICATOR` im Sketch.

Typ: Subscribe
Topic: SmBrd/<DevID>/Settings/DataPackPubCycleTime
Payload: Payload=<INT32>

Legt das Sendeintervall für den Sensor Datenblock fest (siehe Topic *"SmBrd/<DevID>/Data/StData"* weiter oben):
> 0 = Sendeintervall in Sekunden (Limit: `MAX_SET_DATA_PACKET_PUB_CYCLE_TIME`)
  0 = einmaliges Senden des Sensor Datenblock ohne Änderung des Sendeintervalls
< 0 = Rücksetzen auf den Default-Wert (`DEFAULT_DATA_PACKET_PUB_CYCLE_TIME)`

Den Default-Wert definiert die Konstante `DEFAULT_DATA_PACKET_PUB_CYCLE_TIME` im Sketch.

Typ: Subscribe
Topic: SmBrd/<DevID>/Settings/PrintSensorVal
Payload: Payload=['0'|'1']

Legt fest, ob die von den Sensoren gelesenen Werte auch im seriellen Terminal-Fenster (115200Bd) angezeigt werden
0 = off
1 = on

Den Default-Wert definiert die Konstante *DEFAULT_PRINT_SENSOR_VALUES* im Sketch.

Typ: Subscribe
Topic: SmBrd/<DevID>/Settings/PrintMqttDataProc
Payload: Payload=['0'|'1']

Legt fest, ob die gesendeten und empfangenen MQTT Messages auch im seriellen Terminal-Fenster (115200Bd) angezeigt werden
0 = off
1 = on

Den Default-Wert definiert die Konstante *DEFAULT_PRINT_MQTT_DATA_PROC* im Sketch.

Typ: Subscribe
Topic: SmBrd/<DevID>/OutData/LedBar
Payload: Payload=['0'-'9']

Zeigt den angegebenen Wert am LED Bar Indikator dar. Dabei wird die Datenquelle für den LED Bar Indikator auf 0 = *kLedBarNone* gesetzt.

Typ: Subscribe
Topic: SmBrd/<DevID>/OutData/LedBarInv
Payload: Payload=['0'-'9']

Zeigt den angegebenen Wert invers am LED Bar Indikator dar. Dabei wird die Datenquelle für den LED Bar Indikator auf 0 = *kLedBarNone* gesetzt.

Typ: Subscribe
Topic: SmBrd/<DevID>/OutData/Led
Payload: Payload=['0'-'9']=['0'|'1']

Schaltet eine einzelne LED am LED Bar Indikator ein oder aus. Dabei wird die Datenquelle für den LED Bar Indikator auf 0 = *kLedBarNone* gesetzt.
0 = off
1 = on

Typ: Subscribe
Topic: SmBrd/<DevID>/Ack/PacketNum
Payload: Payload=<INT32>

Wird als optionale Trigger-Bedingung für den Watchdog Monitor verwendet.
Ein Server quittiert dem *ESP32SmartBoards* hierüber die Paketnummer des letzten empfangenen *"Sensor Datenblock"* (Variable *PacketNum* in *JsonDataPacket*).

Bei aktiver Konfigurations-Option *CFG_SRV_WDT_ON_MQTT_RECV* wird der Watchdog Monitor abhängig von der letzten quittierte Paketnummer bedient (siehe folgender Abschnitt *"Watchdog Monitor"*).

Watchdog Monitor

Der Watchdog Monitor überwacht kontinuierlich den zeitlichen Programmablauf des *ESP32SmartBoards*. Wenn das System für eine vorgegebene Zeitspanne den Watchdog Timer nicht mehr bedient, löst dieser einen Reset aus und startet so das System neu.

Watchdog Timer:

Der Watchdog Timer ist nur aktiv, wenn er über die Konfigurations-Option *CFG_ENABLE_HW_WDT* freigegeben wurde. Diese Einstellung kann über Bluetooth mit Hilfe des grafischen Konfigurations-Tools *"Esp32ConfigUwp"* gesetzt werden (siehe Abschnitt *"Board Konfiguration"* oben). Wenn der Watchdog Timer aktiviert wurde, muss er innerhalb des durch *APP_WDT_TIMEOUT* definierten Zeitintervalls getriggert werden. Das erfolgt während des zyklischen Programmablaufes innerhalb der Hauptfunktion `loop()`.

Acknowledge Paket

Bei aktiver Konfigurations-Option *CFG_SRV_WDT_ON_MQTT_RECV* wird der Watchdog Monitor abhängig von der letzten quittierte Paketnummer bedient. Diese Einstellung kann über Bluetooth mit Hilfe des grafischen Konfigurations-Tools *"Esp32ConfigUwp"* gesetzt werden (siehe Abschnitt *"Board Konfiguration"* oben). Der Server, der die vom *ESP32SmartBoards* gesendeten Daten verarbeitet, quittiert über das Topic *"SmBrd/<DevID>/Ack/PacketNum"* die Paketnummer des letzten empfangenen *"Sensor Datenblock"* (Variable *PacketNum* in *JsonDataPacket*). Liegt die letzte quittierte Paketnummer um mehr als *APP_WINSIZE_NON_ACK_MQTT_PACKETS* Pakete gegenüber dem letzten gesendeten Paket zurück, wird dies als Verlust der WLAN-Verbindung gedeutet und der Watchdog Timer nicht mehr bedient. Das führt zu einem Reset und damit einem Neustart des *ESP32SmartBoards* mit anschließendem Neuaufbau der Verbindung.

Kalibrierung des CO2 Sensors MH-Z19

Eine falsch ausgeführte Kalibrierung kann dazu führen, dass der Sensor unbrauchbar wird. Es ist daher wichtig, die Funktionsweise der Kalibrierung zu verstehen.

Der Sensor ist für einen Einsatz im 24/7 Dauerbetrieb konzipiert. Er unterstützt die Modi AutoCalibration, Kalibrierung per Hardware-Signal (manuell ausgelöst) und Kalibrierun per Software-Kommando (ebenfalls manuell ausgelöst). Das *ESP32SmartBoard* nutzt davon die Modi AutoCalibration und manuelle Kalibrierun per Software-Kommando. Unabhängig von der jeweiligen Methode wird durch die Kalibrierung sensor-intern der Referenzwert von 400 ppm CO2 gesetzt. Eine Konzentration von 400 ppm gilt als CO2-Normalwert in der Erdatmosphäre, also als typischer Wert der Außenluft im ländlichen Raum.

(1) AutoCalibration:

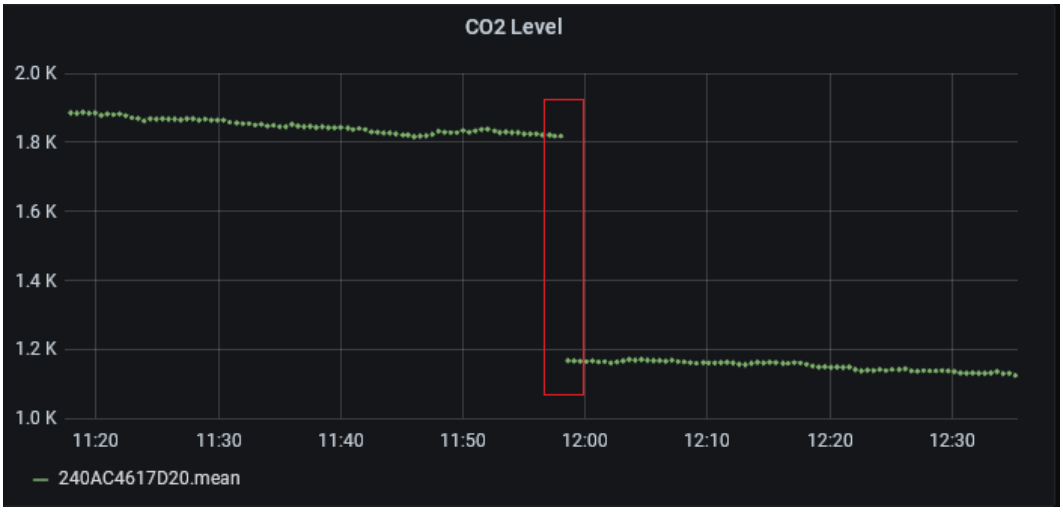
Bei der AutoCalibration überwacht der Sensor permanent die gemessenen CO2 Werte. Der jeweils niedrigste innerhalb von 24 Stunden gemessene Wert wird dabei als Referenzwert von 400 ppm CO2 interpretiert. Diese Methode ist vom Sensor-Hersteller empfohlen für den Einsatz des Sensors in normalen Wohn- oder Büro-Räumen, die regelmäßig gut gelüftet werden. Dabei wird implizit davon ausgegangen, dass beim Lüften die Raumluft komplett getauscht wird und somit die CO2-Konzentration im Raum bis auf den Normalwert der Erdatmosphäre / Außenluft abfällt.

In seinem Datasheet weist der Sensor-Hersteller jedoch explizit darauf hin, dass die AutoCalibration Methode nicht für den Einsatz in landwirtschaftlichen Gewächshäusern, Ställen, Kühlschränken usw. genutzt werden kann. Hier sollte die AutoCalibration disabled werden.

Im *ESP32SmartBoard* Sketch dient die Konstante `DEFAULT_MHZ19_AUTO_CALIBRATION` zum aktivieren (*true*) bzw. deaktivieren (*false*) der AutoCalibration Methode. Die AutoCalibration Funktion wird einmalig beim Systemstart gesetzt. Bei Bedarf kann die AutoCalibration Methode per Taster invertiert werden. Dazu sind folgende Schritte notwendig:

1. KEY0 drücken und durchgehend bis Schritt 3 gedrückt halten
2. Reset am ESP32DevKit drücken und wieder loslassen
3. KEY0 noch weitere 2 Sekunden gedrückt halten
4. KEY0 loslassen

Hinweis: Der AutoCalibration Modus kann insbesondere in den ersten Tagen nach Inbetriebnahme des Sensors zu stark sprunghaften Veränderungen des CO2 Wertes führen. Nach einiger Zeit im 24/7 Dauerbetrieb nimmt dieser Effekt immer mehr ab.



Die durch den AutoCalibration Modus verursachten spontanen Unstetigkeiten lassen sich durch eine manuelle Kalibrierung des Sensors vermeiden.

(2) Manuelle Kalibrierung:

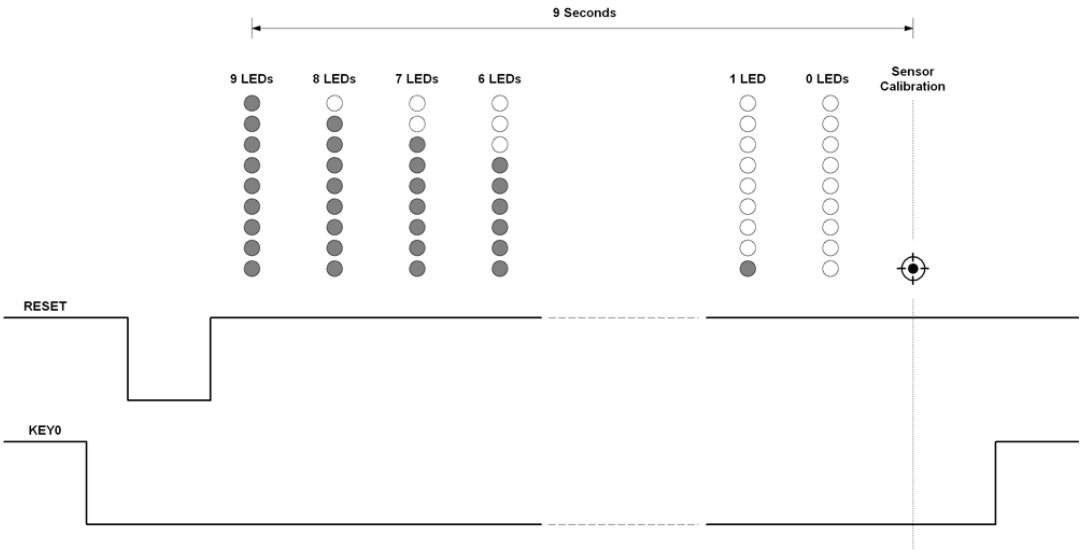
Vor einer manuellen Kalibrierung muss der Sensor für mindestens 20 Minuten in einer stabilen Referenzumgebung mit 400 ppm CO2 betrieben werden. Diese Forderung ist im Amateur- und Hobby-Bereich ohne definierte Kalibrierumgebung nur näherungsweise realisierbar. Dazu kann das *ESP32SmartBoard* im Freien an einem schattigen Platz oder in einem Raum in der Nähe eines geöffneten Fensters ebenfalls im Schatten betrieben werden. In dieser Umgebung muss das *ESP32SmartBoard* für mindestens 20 Minuten arbeiten, bevor die Kalibrierung ausgelöst werden kann.

(2a) Direkte Kalibrierung:

Arbeitet das *ESP32SmartBoard* bereits seit mindestens 20 Minuten in der 400 ppm Referenzumgebung (im Freien, am geöffneten Fenster), kann der Sensor direkt kalibriert werden. Dazu sind folgende Schritte notwendig:

1. KEY0 drücken und durchgehend bis Schritt 4 gedrückt halten
2. Reset am ESP32DevKit drücken und wieder loslassen
3. das *ESP32SmartBoard* startet einen Countdown von 9 Sekunden, dabei blinkt die LED-Bar im Sekundentakt und zeigt die noch verbleibende Zeit in Sekunden an
4. KEY0 die gesamte Zeit über gedrückt halten bis der Countdown abgeschlossen ist und die LED-Bar die abgeschlossene Kalibrierung mit 3x schnellen Flashen quittiert
5. KEY0 loslassen

Wird KEY0 während des Countdowns losgelassen, bricht das *ESP32SmartBoard* die Prozedur ab, ohne den Sensor zu kalibrieren.

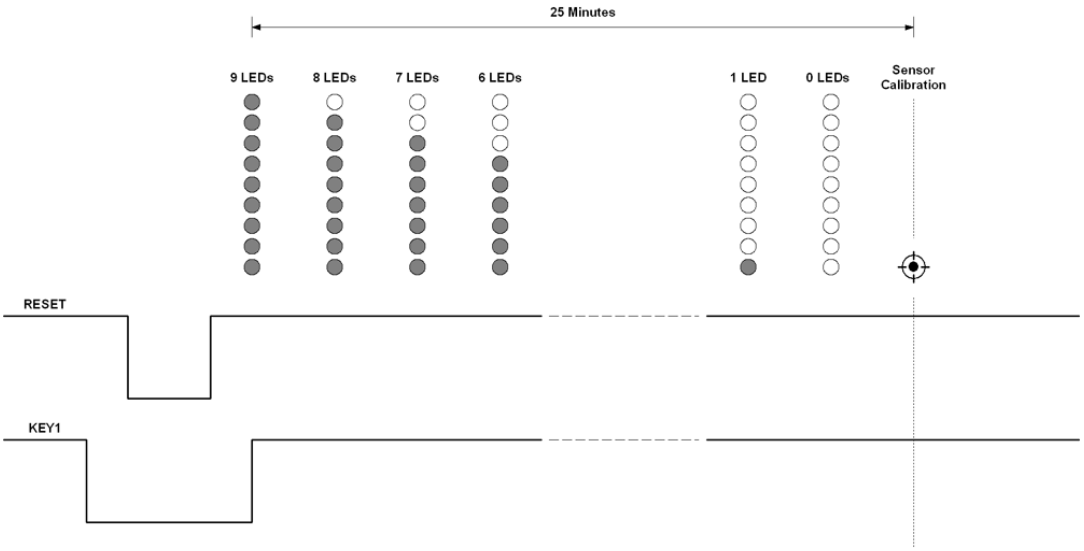


(2b) Unbeaufsichtigte, zeitverzögerte Kalibrierung:

Nachdem das *ESP32SmartBoard* in der 400 ppm Referenzumgebung (im Freien, am geöffneten Fenster) platziert wurde, kann die Kalibrierung unbeaufsichtigt und zeitverzögert durchgeführt werden. Dazu sind folgende Schritte notwendig:

1. KEY1 drücken und durchgehend bis Schritt 2 gedrückt halten
2. Reset am ESP32DevKit drücken und wieder loslassen
3. KEY1 loslassen

Das *ESP32SmartBoard* startet einen Countdown von 25 Minuten. Während des Countdowns blinkt die LED-Bar im Sekundentakt und zeigt die noch verbleibende Zeit an (25 Minuten / 9 LEDs = 2:46 Minuten / LED). Nach Ablauf des Countdowns wird der Kalibrievorgang ausgelöst und mit 3x schnellen Flashen der LED-Bar quittiert. Anschließend wird das *ESP32SmartBoard* per Software-Reset neu gestartet.



Laufzeitausgaben im seriellen Terminal-Fenster

Zur Laufzeit werden alle relevanten Informationen im seriellen Terminal-Fenster (115200Bd) ausgegeben. Insbesondere während des Systemstarts (Sketch Funktion *setup()*) werden hier auch Fehlermeldungen angezeigt, die ggf. auf eine fehlerhafte Softwarekonfiguration zurückzuführen sind. Diese Meldungen sollten insbesondere während der Erstinbetriebnahme auf jeden Fall beachtet werden.

In der Hauptschleife des Programms (Sketch Funktion *main()*) werden zyklisch die Werte des DHT22 (Temperatur und Luftfeuchtigkeit) und des MH-Z19 (CO2 Level und Sensor Temperatur) angezeigt. Darüber hinaus informieren auch hier wieder Meldungen über evtl. Probleme beim Zugriff auf die Sensoren.

Durch aktivieren der Zeile *#define DEBUG* am Anfang des Sketches werden weitere, sehr detaillierte Laufzeitmeldungen angezeigt. Diese sind insbesondere während der Programmentwicklung bzw. zur Fehlersuche sehr hilfreich. Durch auskommentieren der Zeile *#define DEBUG* werden die Ausgaben wieder unterdrückt.

Verwendete Drittanbieter Komponenten

1. **MQTT Library**

   Für die MQTT-Kommunikation wird die Bibliothek *Arduino Client for MQTT* verwendet:
   https://github.com/knolleary/pubsubclient

   Die Installation erfolgt mit dem Library Manager der Arduino IDE.

2. **MH-Z19 CO2 Sensor**

   Für den MH-Z19 CO2 Sensor wird folgende Treiberbibliothek verwendet:
   https://www.arduino.cc/reference/en/libraries/mh-z19/

   Die Installation erfolgt mit dem Library Manager der Arduino IDE.

3. **DHT Sensor**

   Für den DHT Sensor (Temperatur, Luftfeuchtigkeit) wird die Treiberbibliothek von Adafruit verwendet. Die Installation erfolgt mit dem Library Manager der Arduino IDE.

4. **ArduinoJson**

   Zur Erstellung der JSON-Records wird die Bibliothek *ArduinoJson* verwendet: https://github.com/bblanchon/ArduinoJson

   Die Installation erfolgt mit dem Library Manager der Arduino IDE.

Für das Bluetooth Konfigurations-Framework aus dem Projekt *<ESP32BleConfig>* werden keine Drittanbieter Komponenten verwendet. Sowohl die Unterstützung für BLE als auch für den EEPROM werden zusammen mit dem Arduino ESP32 Add-on installiert.