Projekt: https://github.com/ronaldsieber/ESP32BleConfig
Lizenz: MIT
Autor: Ronald Sieber

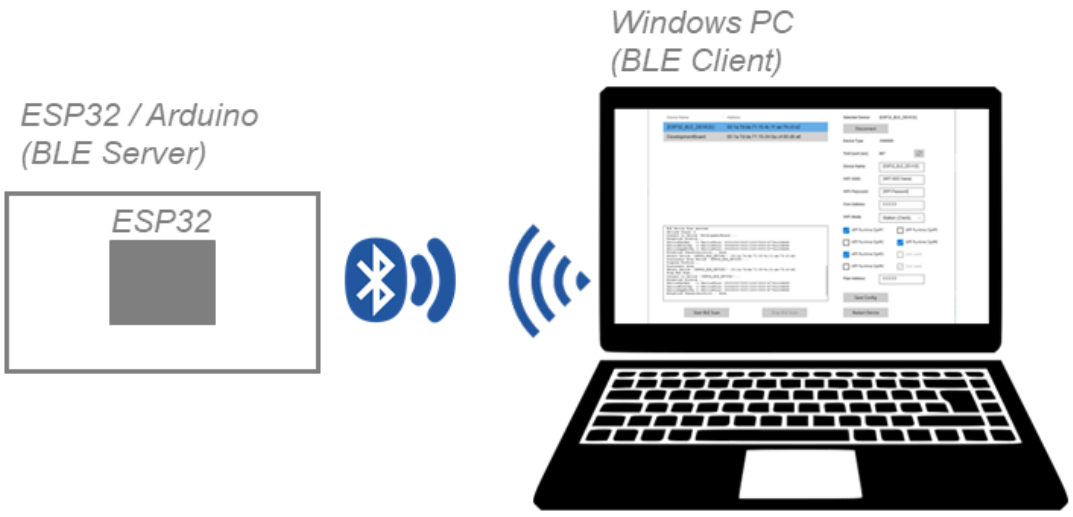
ESP32BleConfig

Dieses Framework stellt Funktionalitäten bereit, um eine ESP32/Arduino Applikation zur Laufzeit über Bluetooth zu konfigurieren. Dadurch entfallen die sonst typischen Anpassungen im Sourcecode, wie beispielsweise das Eintragen der eigenen WLAN-Konfigurationsdaten oder das feste Kodieren von Zieladressen für MQTT-Broker direkt im C Quelltext des Sketches. Zum einen können damit Arduino-Projekte leichter in öffentlichen Repositories verwaltet werden, ohne zuvor persönliche Daten entfernen zu müssen. Zum anderen kann für mehrere Boards ein und dasselbe Binary verwendet werden, da die Personalisierung verschiedener Parameter erst zur Laufzeit über Bluetooth erfolgt.

Das Framework besteht aus einem Code-Template, das in das jeweilige ESP32/Arduino-Projekt integriert wird und einem Konfigurations-Tool mit grafischer Oberfläche (derzeit für Windows, evtl. später auch als Android-App). Der ESP32/Arduino ist dabei der Bluetooth Server, das grafische Konfigurations-Tool fungiert als Client. Folgende Parameter können damit konfiguriert werden:

- WLAN-Modus (Access Point, Station/Client)
- Konfiguration der WLAN-Zugangsdaten (SSID, Passwort)
- eigene IP-Adresse des Boards (DHCP oder statische IP)
- individueller Device Name des Boards
- Adresse eines Kommunikationspartners (z.B. MQTT Broker)
- bis zu 8 frei nutzbare Laufzeitoptionen (enable/disable)

Die Konfigurationsdaten werden via EEPROM-Simulation persistent im Flash des ESP32 gespeichert und bleiben somit auch nach einem Neustart erhalten.



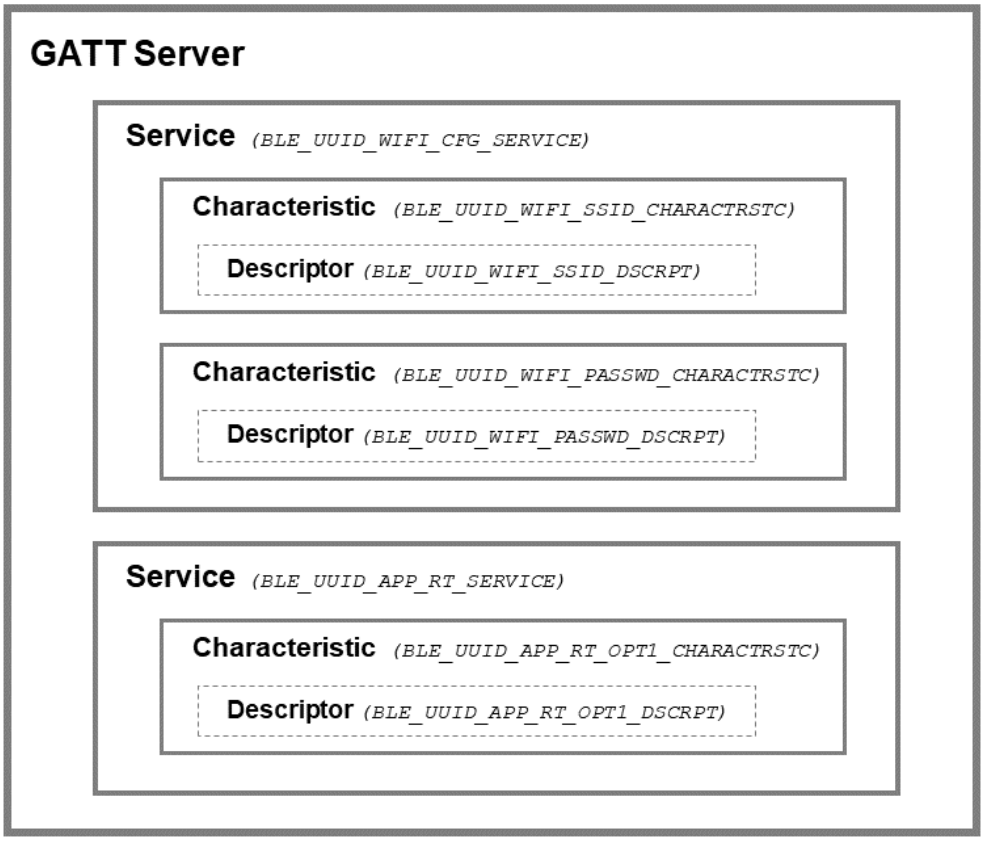
Die Integration des Frameworks in eine reale ESP32/Arduino-Anwendung veranschaulicht das Projekt <*ESP32SmartBoard_MqttSensors_BleCfg*>.

Bluetooth Device Profile

Das Konfigurations-Framework basiert auf einem privaten Bluetooth Device Profile (Generic Attribute Profile, GATT). Dieses umfasst die 3 Services:

- Device Management
- WIFI Config
- App Runtime Options

Jeder Service beinhaltet mehrere Charakteristiken. Diese repräsentieren die internen Zustände eines BLE Gerätes, wie beispielsweise die aktuellen Werte von Variablen (z.B. den SSID Namen, das WLAN-Passwort oder auch die Uptime des Gerätes). Weiterhin kann ein Lese- oder Schreibzugriff auf eine Charakteristik auch eine Aktion triggern, wie z.B. das Sichern der Konfigurationsdaten in den EEPROM oder das Auslösen eines Restarts. Jede Charakteristik wiederum kann ein oder auch mehrere Deskriptoren enthalten. Die Deskriptoren dienen zur Beschreibung der Charakteristik durch entsprechende Metadaten (z.B. textuelle Beschreibung dessen, was die Charakteristik repräsentiert). Die Properties einer Charakteristik definieren, welche Operationen für die betreffende Charakteristik erlaubt sind (Lesen, Schreiben, Benachrichtigen usw.). Der Value einer Charakteristik repräsentiert letztlich die aktuellen Daten der betreffenden Charakteristik.



Alles Services, Charakteristiken und Deskriptoren besitzen jeweils ihre eigene, individuelle UUID. Das vollständige Bluetooth Device Profile ist in <BleProfileDefinition.txt> beschrieben.

ESP32/Arduino Part des Frameworks

Der ESP32/Arduino Part des Frameworks implementiert das für die Konfiguration erforderliche Bluetooth Device Profile (`class ESP32BleCfgProfile`) und übernimmt die persistente Speicherung der Konfigurationsdaten im EEPROM (`class ESP32BleAppCfgData`). Das Sketch-Template `ESP32BleConfig.ino` zeigt die Nutzung des Frameworks in eigenen Applikationen.

Die im Sketch-Template *ESP32BleConfig.ino* definierte Struktur *tAppCfgData AppCfgData_g* beinhaltet zunächst die Standardwerte für die Konfiguration ("Werkseinstellungen"):

```
static tAppCfgData  AppCfgData_g =
{

    APP_CFGDATA_MAGIC_ID,                           // .m_ui32MagicID

    APP_DEFAULT_DEVICE_NAME,                        // .m_szDevMntDevName

    APP_DEFAULT_WIFI_SSID,                          // .m_szWifiSSID
    APP_DEFAULT_WIFI_PASSWD,                        // .m_szWifiPasswd
    APP_DEFAULT_WIFI_OWNADDR,                       // .m_szWifiOwnAddr
    APP_DEFAULT_WIFI_OWNMODE,                       // .m_ui8WifiOwnMode

    APP_DEFAULT_APP_RT_OPT1,                        // .m_fAppRtOpt1 : 1
    APP_DEFAULT_APP_RT_OPT2,                        // .m_fAppRtOpt2 : 1
    APP_DEFAULT_APP_RT_OPT3,                        // .m_fAppRtOpt3 : 1
    APP_DEFAULT_APP_RT_OPT4,                        // .m_fAppRtOpt4 : 1
    APP_DEFAULT_APP_RT_OPT5,                        // .m_fAppRtOpt5 : 1
    APP_DEFAULT_APP_RT_OPT6,                        // .m_fAppRtOpt6 : 1
    APP_DEFAULT_APP_RT_OPT7,                        // .m_fAppRtOpt7 : 1
    APP_DEFAULT_APP_RT_OPT8,                        // .m_fAppRtOpt8 : 1
    APP_DEFAULT_APP_RT_PEERADDR                     // .m_szAppRtPeerAddr

};
```

Im ersten Schritt wird der *setup()* Funktion des Sketches diese Struktur als In/Out Parameter an die Methode *ESP32BleAppCfgData_g.LoadAppCfgDataFromEeprom()* übergeben:

```
iResult = ESP32BleAppCfgData_g.LoadAppCfgDataFromEeprom(&AppCfgData_g);
if (iResult == 1)
{
    Serial.print("-> Use saved Data read from EEPROM");
}
else if (iResult == 0)
{
    Serial.println("-> Keep default data untouched");
}
else
{
    Serial.print("-> ERROR: Access to EEPROM failed!");
}
```

Die Methode *LoadAppCfgDataFromEeprom()* prüft, ob sich im EEPROM bereits gültige Konfigurationsdaten befinden. Ist das der Fall (gültige Signatur und CRC), werden diese Daten in die Struktur *AppCfgData_g* übernommen und an die Applikation zurückgegeben. Wurde noch keine Konfiguration durchgeführt (keine gültigen Daten im EEPROM), bleiben die Standardwerte ("Werkseinstellungen") erhalten.

Im zweiten Schritt prüft die *setup()* Funktion des Sketches, ob der Konfigurationsmodus gestartet werden soll (hier gesteuert durch das Flag *fStateBleCfg_g*). Ist dies der Fall, erstellt die Methode *ESP32BleCfgProfile_g.ProfileSetup()* das entsprechende Bluetooth Device Profile und startet den GATT Service.

```
if ( fStateBleCfg_g )
{
    ESP32BleCfgProfile_g.ProfileSetup(APP_DEVICE_TYPE,
                                      &AppCfgData_g, &AppDescriptData_g,
                                      AppCbHdlrSaveConfig,
                                      AppCbHdlrRestartDev,
                                      AppCbHdlrConStatChg);
}
```

Die an *ProfileSetup()* übergebenen Callback-Handler werden unter folgenden Bedingungen aufgerufen:

Callback Handler	**Bedeutung**
AppCbHdlrSaveConfig()	Über das Bluetooth Device Profile wurde die Aktion *"Konfiguration Schreiben"* ausgelöst (Schreibzugriff auf die Charakteristik *"BLE_UUID_DEVMNT_SAVE_CFG_CHARACTRSTC"*
AppCbHdlrRestartDev()	Über das Bluetooth Device Profile wurde die Aktion *"Restart Device"* ausgelöst (Schreibzugriff auf die Charakteristik *"BLE_UUID_DEVMNT_RST_DEV_CHARACTRSTC"*
AppCbHdlrConStatChg()	Der Connection Status eines Clients (Konfigurations-Tool, GUI) hat sich geändert (connect/disconnect)

In der *loop()* Funktion des Sketches stellt die Methode *ESP32BleCfgProfile_g.ProfileLoop()* die zyklische Zuteilung von CPU Zeit an den GATT Service sicher:

```
if ( fStateBleCfg_g )
{
    ESP32BleCfgProfile_g.ProfileLoop();
}
```

Integration des Frameworks in eigene ESP32/Arduino Applikationen

Um den ESP32/Arduino Part des Frameworks in eigene Applikationen zu integrieren, sind folgende Schritte erforderlich:

1. Kopieren folgender Source Code Dateien in das ESP32/Arduino Projekt:
   - ESP32BleAppCfgData.h
   - ESP32BleAppCfgData.cpp
   - ESP32BleCfgProfile.h
   - ESP32BleCfgProfile.cpp

   Wenn im *ESP32BleCfgProfile.cpp* die Zeile *#define DEBUG* aktiv ist, werden ebenfalls noch folgende beiden Source Code Dateien im ESP32/Arduino Projekt benötigt:
   - Trace.h
   - Trace.cpp

2. Einbinden folgender Header Dateien in das Sketch File:
   ```
   #include "ESP32BleCfgProfile.h"
   #include "ESP32BleAppCfgData.h"
   ```

3. Übernahme des Inhaltes des Sketch-Template *ESP32BleConfig.ino* in die eigene Applikation
   - oder -
   Nutzen des Sketch-Template *ESP32BleConfig.ino* als Ausgangsbasis für die eigene Applikation

Für die Konfiguration via Bluetooth wird eine explizite Aktivierung benötigt. Im Sketch-Template *ESP32BleConfig.ino* dient hierzu das Flag *fStateBleCfg_g*. In einem realen Sketch kann das Flag z.B. durch abfragen eines Tasters beim Systemstart gesetzt werden. Auf dem *ESP32SmartBoard* (siehe Hardware Projekt *<ESP32SmartBoard_PCB>*) ist hierfür der Taster *BLE_CFG* vorgesehen:

```
const int  PIN_KEY_BLE_CFG = 36;                // GPIO36 -> Pin02

pinMode(PIN_KEY_BLE_CFG, INPUT);
fStateBleCfg_g = !digitalRead(PIN_KEY_BLE_CFG);  // Keys are inverted (1=off, 0=on)
if ( fStateBleCfg_g )
{
    ESP32BleCfgProfile_g.ProfileSetup(APP_DEVICE_TYPE,
                                      &AppCfgData_g, &AppDescriptData_g,
                                      AppCbHdlrSaveConfig,
                                      AppCbHdlrRestartDev,
                                      AppCbHdlrConStatChg);
}
```

Um den Bluetooth-Konfigurationsmodus für das gezeigte Beispiel 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

Das Sketch-Template *`ESP32BleConfig.ino`* beinhaltet Code, um den Bluetooth Konfigurations- und Verbindungsstatus durch Blinken der blauen LED auf dem ESP32DevKit zu signalisieren. Enabled werden die Code-Sektionen durch den Konfigurations-Abschnitt am Anfang des Sketches:

```
const int CFG_ENABLE_STATUS_LED = 1;
```

Sind die Code-Sektionen enabled, zeigt die blauen LED des ESP32DevKit durch langsames Blinken an, dass sich das Device im Konfigurations-Modus befindet (*`fStateBleCfg_g == TRUE`*). Nach dem Connect eines Clients (grafische Konfigurations-Tool) wechselt das Device in ein schnelles Blinken.

Grafisches Konfigurations-Tool

Das grafische Konfigurations-Tool fungiert als Client und verbindet sich via Bluetooth mit dem als Server operierenden ESP32/Arduino. Das Tool ist als UWP Applikation (Universal Windows Platform) in Visual Studio 2019 implementiert. Ein natives, in .NET integriertes Bluetooth Subsystem existiert nur für UWP. Dagegen sind klassische WindowsForm Applikationen bei Bluetooth auf 3rd Party Komponenten angewiesen.

Eine empfehlenswerte Knowledge Base zur Nutzung des BLE Subsystem in .NET/UWP Applikationen ist das Microsoft *"Bluetooth Low Energy sample"*:

https://docs.microsoft.com/en-us/samples/microsoft/windows-universal-samples/bluetoothle/

Der für das Konfigurations-Tool genutzte PC muss über eine geeignete Bluetooth Schnittstelle verfügen. Bei Laptops ist eine solche Schnittstelle typischerweise integriert. Für Desktop PC eignet sich ein USB Bluetooth Adapter Dongle (z.B. TP-Link UB400 Nano).

Die Implementierung des grafischen Konfigurations-Tools untergliedert sich im Wesentlichen in folgende Source Code Files:

- Esp32ConfigUwp\MainPage.xaml.cs:
  Funktionalitäten der grafischen Oberfläche sowie deren Anbindung an das Backend

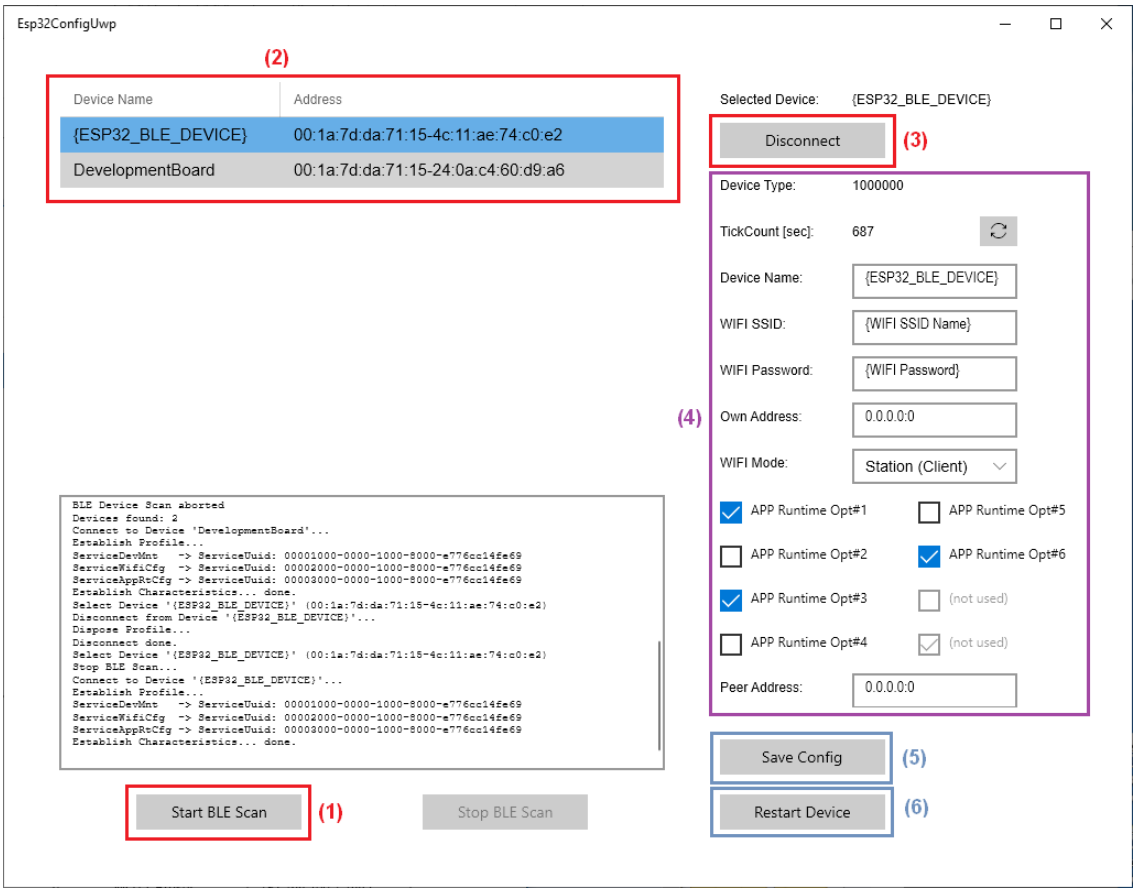
- Esp32ConfigUwp\BleDeviceManagement.cs:
  BLE Device Management Funktionalitäten wie beispielsweise Scannen verfügbarer Geräte, Connect und Disconnect

- Esp32ConfigUwp\Esp32BleGattServices.cs:
  Implementierung des spezifischen Bluetooth Device Profiles (`class Esp32BleAppCfgProfile`) mit den 3 Services "Device Management" (`class BleGattServiceDevMnt`), "WIFI Config" (`class BleGattServiceWifi`) und "App Runtime Options" (`class BleGattServiceAppRt`)

Deployment für UWP Applikationen

Auf dem Entwicklungs-PC, auf dem eine UWP Applikation mit Visual Studio erstellt wurde, lässt sich das generierte Executeable genauso einfach starten, wie jede andere klassischer Windows Applikation auch. Im Gegensatz dazu gestaltet sich die Verteilung von UWP Applikationen auf andere Geräte etwas komplexer als die Weitergabe klassischer Windows Applikationen. Dazu ist die UWP Applikation in einen Installer zu verpacken *("Project -> Publish -> Create App Packages… -> Sideloading"*, Tutorial siehe https://docs.microsoft.com/en-us/windows/msix/package/packaging-uwp-apps). Dieser Schritt erfordert ein geeignetes Zertifikat, im einfachsten Fall ein mit Hilfe des Assistenten von Visual Studio selbst generiertes temporäres Zertifikat. Da ein solches Zertifikat von keiner Root Certificate Authority (CA) auf Echtheit geprüft werden kann, muss das Zertifikat (**.pfx* Datei) manuell auf dem jeweiligen Zielrechner in den Windows Certificate Store in den Zweig *"Vertrauenswürdige Personen"* importiert werden (unter Verwendung des Windows Tools *"certlm"*).

Konfiguration eines ESP32/Arduino mittels grafischem Konfigurations-Tool



Zunächst ist der ESP32/Arduino in den Bluetooth-Konfigurationsmodus zu versetzen (für *ESP32SmartBoard* siehe Beispiel oben). Anschließend sind im grafischen Konfigurations-Tool die nachstehend beschriebenen Bedienschritte auszuführen:

1. BLE Scan mit Hilfe des Buttons *"Start BLE Scan"* (1) starten. Im GridView (2) werden alle gefunden Devices aufgelistet.

2. Markieren des zu konfigurierenden Devices im GridView (2)

3. Klicken des Buttons *"Connect"* (3), um das Konfigurations-Tool (Client) mit dem zu konfigurierenden ESP32/Arduino (Server) zu verbunden. Dabei werden die aktuellen Konfigurationsdaten vom Arduino gelesen und in der GUI angezeigt (4).

4. Durch entsprechendes bearbeiten der Einträge in den Konfigurations-Elementen (4) können die einzelnen Parameter individuell angepasst werden.

5. Anschließend kann die modifizierte Konfiguration mit Hilfe des Buttons *"Save Config"* (5) auf den Arduino zurückgeschrieben werden.

6. Mit Hilfe des Buttons *"Restart Device"* (6) kann ein Restart des ESP32/Arduino mit den neuen Konfigurationsdaten ausgelöst werden.

Customizing des Bluetooth Device Profils

Durch das Konfigurations-Tool werden neben den Konfigurationsparametern selbst (Value einer Charakteristik) für die Konfigurationselemente im Service *"App Runtime Options"* (`BLE_UUID_APP_RT_SERVICE`) auch die Deskriptoren vom ESP32/Arduino gelesen und angezeigt. Dadurch lassen sich die Bezeichner der zugehörigen Elemente in der grafischen Oberfläche durch entsprechende Strings im Sourcecode des ESP32/Arduino anpassen (Customizing). Die den Deskriptoren zugeordneten Bezeichner enthält die im Sketch-Template `ESP32BleConfig.ino` definierte Struktur `tAppDescriptData AppDescriptData_g`:

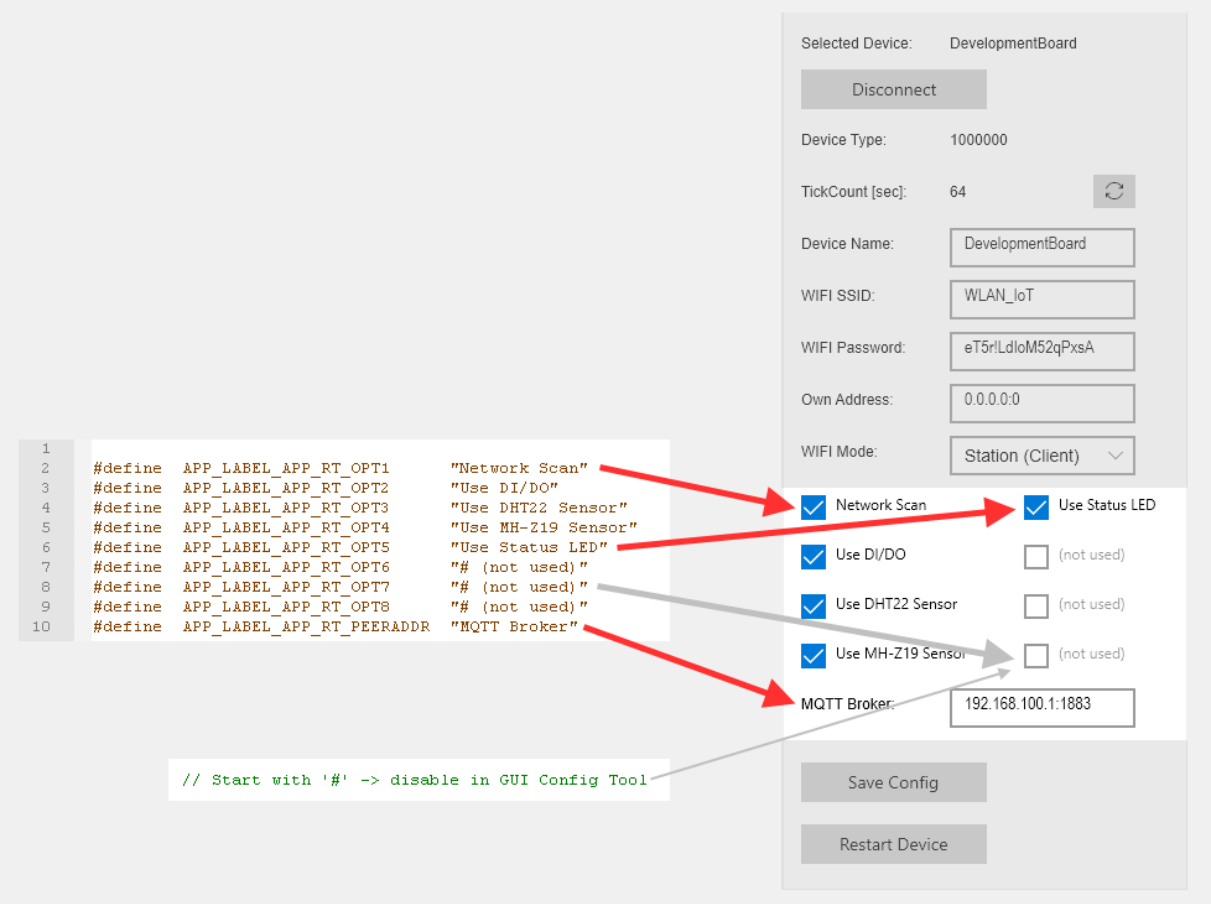
```
static tAppDescriptData  AppDescriptData_g =
{

    APP_DESCRPT_WIFI_OWNMODE_FEATLIST,              // .m_ui16OwnModeFeatList

    APP_LABEL_APP_RT_OPT1,                          // .m_pszLabelOpt1
    APP_LABEL_APP_RT_OPT2,                          // .m_pszLabelOpt2
    APP_LABEL_APP_RT_OPT3,                          // .m_pszLabelOpt3
    APP_LABEL_APP_RT_OPT4,                          // .m_pszLabelOpt4
    APP_LABEL_APP_RT_OPT5,                          // .m_pszLabelOpt5
    APP_LABEL_APP_RT_OPT6,                          // .m_pszLabelOpt6
    APP_LABEL_APP_RT_OPT7,                          // .m_pszLabelOpt7
    APP_LABEL_APP_RT_OPT8,                          // .m_pszLabelOpt8

    APP_LABEL_APP_RT_PEERADDR                       // .m_pszLabelPeerAddr

};
```

Das Feld `m_ui16OwnModeFeatList` ist eine Bitmaske, die festlegt, welche WIFI Modi der ESP32/Arduino unterstützen kann (`WIFI_MODE_STA` = Station/Client, `WIFI_MODE_AP` = Access Point):

```
#define  APP_DESCRPT_WIFI_OWNMODE_FEATLIST  (WIFI_MODE_STA | WIFI_MODE_AP)
```

Beginnt ein Bezeichner im ESP32/Arduino-Sketch mit einem *'#'* Zeichen, führt das im grafischen Konfigurations-Tool zum Disablen des betreffenden Elements.

```
#define  APP_LABEL_APP_RT_OPTx  "# (not used)"  // Start with '#' -> disable in GUI
```



Verwendete Drittanbieter Komponenten

Es werden keine Drittanbieter Komponenten verwendet. Sowohl die Unterstützung für BLE als auch für den EEPROM werden zusammen mit dem Arduino ESP32 Add-on installiert.

Praxis Hinweise

Bei Änderungen, Erweiterungen oder Anpassungen des Bluetooth Device Profiles sind in er Regel sowohl Server- als auch Client-Seite betroffen. In der Praxis hat es sich bewährt, zunächst die Server-Implementierung des ESP32/Arduino zu modifizieren und zu testen. Für den Test eignet sich als universeller Bluetooth Client die freie Android App *"nRFConnect"* von Nordic Semiconductor:

https://github.com/NordicSemiconductor/Android-nRF-Connect