Drahtlose Sensordatenübertragung mit ESP32: Von ESP-NOW bis zur Browserdarstellung über BLE

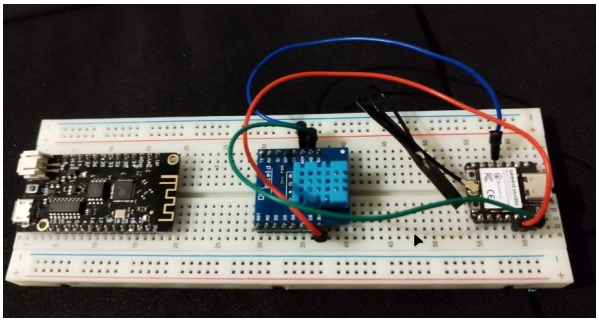
Einleitung

1.1 Im Rahmen dieses Projekts wird die Kommunikation zwischen zwei Mikrocontrollern der ESP32-Familie untersucht und praktisch umgesetzt. Ein ESP32-C3 liest Messwerte eines Sensors aus, dem DHT11 und sendet Sensordaten per ESP-NOW an einen ESP32. Dieser verarbeitet die empfangenen Daten als ESP-NOW-Empfänger und leitet sie über Bluetooth Low Energy (BLE) weiter als BLE-Sender.
Mit Hilfe der Web Bluetooth API von Google Chrome können die Daten schließlich im Browser empfangen, ausgelesen und dargestellt werden.

Ziel des Projekts ist es, einen durchgängigen Datenfluss vom Sensor bis hin zur plattformunabhängigen Darstellung im Browser zu realisieren.

1.2 Zur Umsetzung des Projekts wird folgende Hardware und Software verwendet:- Steckbrett 1x

- USB-C-Kabel und Micro-USB-Kabel 1x
- Kabel für Steckverbindungen 3x
- Seeed XIAO ESP32-C3 1x
- DHT11
- ESP32-Dev-Kit
- Arduino IDE 2.3.6
- Microsoft Visual Studio Code
- Google Chrome Web Bluetooth API



Technische Grundlagen

2.1 Der ESP32 ist ein von Espressif Systems entwickelter Mikrocontroller. Er basiert auf einer Dual-Core-Tensilica-LX6-Architektur und verfügt über zahlreiche Peripherieschnittstellen sowie analoge Ein- und Ausgänge. Diese Eigenschaften machen ihn besonders geeignet für Anwendungen im Bereich der IoT.
Der ESP32-C3 stellt eine Weiterentwicklung dar, die auf einer energieeffizienten RISC-V-Architektur basiert. Im Vergleich zum klassischen ESP32 hat er eine reduzierte Rechenleistung, dafür jedoch einen niedrigeren Energieverbrauch und eine kostengünstigere Herstellung. Aufgrund dieser Eigenschaften eignet sich der ESP32-C3 vor allem für batteriebetriebene Sensorknoten, während der ESP32 typischerweise als zentrales Gateway in einer verteilten IoT-Architektur eingesetzt wird.

2.2 ESP-NOW wurde von Espressiv Systems entwickelt und ist ein Kommunikationsprotokoll für diverse ESP Geräte, welches eine direkte und Stromsparende Peer-to-Peer Kommunikation ermöglicht, ohne auf einen Router oder eine WiFi-Verbindung angewiesen zu sein.
ESP-NOW ermöglicht eine direkte Verbindung von bis zu 20 Geräten auf einer Reichweite von über 200 Metern.

ESP-NOW ist eine Verbindungslose Kommunikation, bei der sich Geräte direkt über ihre MAC-Adresse austauschen. Es wird kein Netzwerk benötigt, die Geräte müssen lediglich einmal „pairen“, und können sicher ohne Handshake kommunizieren. Zudem ist eine Verschlüsselung mit der AES-128-Methode und dem CCMP-Verfahren möglich, was die Sicherheit in der Kommunikation gewährleistet.

Bluetooth Low Energy (BLE) hingegen ist ein international standardisiertes Funkprotokoll, das speziell für den energiesparenden Datenaustausch entwickelt wurde. Es eignet sich insbesondere für kurze Übertragungen und bietet zugleich eine breite Unterstützung in mobilen Endgeräten und Computern. Im vorliegenden Projekt dient BLE als Schnittstelle zwischen dem ESP32 und einem Webbrowser, wodurch die Sensordaten benutzerfreundlich zugänglich gemacht werden können.

2.3 Die Web Bluetooth API ist eine von Google entwickelte Schnittstelle, die in modernen Browsern wie z.B dem Google Chrome-Browser oder Microsoft Edge implementiert ist. Sie ermöglicht es, über einen einfache html-Anwendung direkt mit Bluetooth-Low-Energy-Geräten zu kommunizieren. Dadurch entfällt die Notwendigkeit, eine plattformspezifische Software oder mobile Applikation zu entwickeln.
Im Rahmen des Projekts wird diese Schnittstelle mit einer einfachen html-Seite verwendet.

Im Folgenden wird das Vorgehen beim Konfigurieren der einzelnen Komponenten gezeigt.

Die demonstrierten Sketche (C3_DHT11.ino, C3_ESPNOW.ino, ESP32_ESPNOW.ino, ESP32_BLESERVER.ino und ESP32_BLE_ESPNOW.ino) sowie der Code für die html-Seite befinden dich in meinem Gitlab-Repository und stehen zur freien Verfügung.

https://git-ce.rwth-aachen.de/hendrik-soenke.voss/sensoreaktoren_fdai7723

ESP32-C3

3.1 Der ESP32-C3 muss zuerst die Messwerte des DHT11 auslesen, bevor sie verschickt werden. Dazu eignet sich folgender einfacher Sketch (C3_DHT11.ino).

Die „includes“ Stellen eine Schnittstelle zur Verfügung, für den Umgang mit Sensoren und dem DHT11.

```
1  #include <Adafruit_Sensor.h>
2  #include <DHT_U.h>
3  #include <DHT.h>
4
5
```

Der DHT11 wird mit dem GPIO5 des C3 verbunden und in *setup()* gestartet.

```
6  // GPIO auf dem C3 für den DHT11
7  #define DHTPIN 5
8  // DHT-Typ definieren
9  #define DHTTYPE DHT11
10 // dht.-Objekt erzeugen mit defines
11 DHT dht(DHTPIN, DHTTYPE);
12
13 void setup() {
14   Serial.begin(115200);
15   delay(2000);
16   Serial.println("Starte DHT-Sensor...");
17   dht.begin();
18 }
19
```

Im Loop werden die Messwerte ausgelesen und in float-Variablen gespeichert. In der Fehlerbehandlung wird geprüft, ob es sich bei den empfangenen Daten wirklich um Zahlen handelt. So lässt sich auch prüfen, ob alle Bibliotheken auch richtig inkludiert wurden. Bei gültigen Daten, sollen diese im Seriellen Monitor ausgegeben werden, als Debugging-Information. Der Delay ist wichtig, damit genügend Zeit bleibt um den Bitstrom des DHT11 auszulesen.

```
20  void loop() {
21    // Messwerte auslesen
22    float t = dht.readTemperature(); // Temperature
23    float h = dht.readHumidity(); // Luftfeuchtigkeit
24
25    // --- Fehlerbehandlung ---
26    if (isnan(t) || isnan(h)) {
27      Serial.println("Fehler: Keine gültigen Daten vom DHT-Sensor!");
28    } else {
29      // Ausgabe
30      Serial.print("Temperatur: ");
31      Serial.print(t);
32      Serial.println(" °C");
33
34      Serial.print("Luftfeuchtigkeit: ");
35      Serial.print(h);
36      Serial.println(" %");
37    }
38
39    delay(2000);
40  }
41
```

3.2 Nachdem der ESP32-C3 die Messwerte des DHT11 zuverlässig auslesen kann, sollen diese per ESP-NOW versendet werden. Dazu muss ESP-NOW zum Senden auf dem C3 eingerichtet werden. Damit ESP-NOW auf dem C3 verwendet werden kann, müssen die Bibliotheken *WiFi.h* (Standard-WLAN-Bibliothek für den ESP32), *esp_now.h* (Implementiert ESP-NOW**)** *und esp_wifi.h* (Low-Level WLAN-API) inkludiert werden. Es muss eine Struktur für die Nachricht definiert werden, mit Variablen für Messwerte, die versendet werden sollen. Eine Variable vom Typ *message_struct* wird deklariert mit dem Namen *dhtData*, in der währende der Laufzeit die Messwerte gespeichert werden können (C3_ESPNOW.ino).

```
#include <Adafruit_Sensor.h>
#include <DHT_U.h>
#include <DHT.h>
// ESP-NOW
#include <WiFi.h>
#include <esp_now.h>
#include <esp_wifi.h>

#define DHTPIN 5
#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);

// struct defnieren für Messwerte
typedef struct struct_message {
  float temperature;
  float humidity;
};

// Variable vom Typ struct_message deklarieren
struct_message dhtData;

```

Die MAC-Adresse des Empfängers wird benötigt, da die Kommunikation von ESP-NOW über die MAC-Adresse läuft.

```
// MAC-Addresse des Empfängers festlegen
uint8_t broadcastAddress[] = {0x24, 0x6F, 0x28, 0x0A, 0xFD, 0x5C};

```

Die Callback-Funktion gibt ein Feedback, ob die Nachricht erfolgreich versendet wurde. Dabei wird *onDataSent* ein Zeiger auf die MAC-Adresse des Empfängers übergeben und eine Variable vom Typ *esp_now_send_status_t*, in der der Sendestatus gespeichert wird. Zur Debugging-Information wird der Status im Seriellen Monitor ausgegeben.

```
// Callback, wenn Daten gesendet wurden
void OnDataSent(const wifi_tx_info_t *mac_addr, esp_now_send_status_t status) {
  Serial.print("Sendestatus: ");
  Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Erfolgreich" : "Fehlgeschlagen");
}


```

In *setup()* muss WiFi im Station-Mode gestartet werden und ein Kanal definiert werden. Der Empfänger muss den gleichen Kanal verwenden. 1 ist der primäre Kanal. WIFI_SECOND_CHAN_NONE bedeutet, dass kein sekundärer Kanal benutzt wird.

```
33  void setup() {
34      Serial.begin(115200);
35      delay(2000);
36      Serial.println("Starte DHT-Sensor...");
37      dht.begin();
38      // WLAN im Station-Mode starten
39      WiFi.mode(WIFI_STA);
40      // Kanal auf 1 setzen
41      esp_wifi_set_channel(1, WIFI_SECOND_CHAN_NONE);
```

ESP-NOW initialisieren und per Statuscode (ESP_OK) prüfen, ob die Initialisierung erfolgreich war.

```
42      // ESP-NOW starten und per Statuscode prüfen
43      if (esp_now_init() != ESP_OK)
44      {
45          Serial.println("Fehler beim Initialisieren von ESP-NOW");
46          return;
47      }
```

Den Callback zu registrieren, teilt dem C3 mit, die *OnDataSent*-Funktion aufzurufen, nachdem einen Nachricht verschickt wurde.

```
48      // Callback registrieren
49      esp_now_register_send_cb(OnDataSent);
```

Die struct-Variable *peerInfo* vom Typ *esp_now_peer_info_t* definiert die Kommunikation mit dem Empfänger. Die MAC-Adresse des Empfängers wird mit *memcopy* in *peerInfo* ergänzt, sowie der zu verwendende Kanal. Ebenfalls kann hier auch eine verschlüsselte Kommunikation eingerichtet werden mit *encrypt*. Über *esp_now_add_peer* lässt sich über Statuscodes prüfen, ob der Empfänger-Struct *peerInfo* richtig initialisiert wurde und hinzugefügt werden kann.

```
50      // Empfänger hinzufügen
51      esp_now_peer_info_t peerInfo = {};
52      memcpy(peerInfo.peer_addr, broadcastAddress, 6);
53      peerInfo.channel = 1;
54      peerInfo.encrypt = false;
55      if (esp_now_add_peer(&peerInfo) != ESP_OK)
56      {
57          Serial.println("Fehler beim Hinzufügen des Peers");
58          return;
59      }
60  }
```

In *loop()* werden die Messwerte des DHT11 in den zuvor deklarierten *struct dhtData* gespeichert. Über *esp_now_send* kann diese Struktur nun versendet werden, indem man der Funktion die Adresse des Empfängers, sowie die Struktur und die Größe der Struktur übergibt. Die Funktion liefert das ESP-OK als Statuscode bei korrektem verwenden.

```
64  void loop() {
65    // --- Messwerte auslesen ---
66    float t = dht.readTemperature();
67    float h = dht.readHumidity();
68
69    if (isnan(t) || isnan(h)) {
70      Serial.println("Fehler: Keine gültigen Daten vom DHT-Sensor!");
71    } else {
72      // In die Struktur speichern
73      dhtData.temperature = t;
74      dhtData.humidity = h;
75
76      // Senden
77      esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &dhtData, sizeof(dhtData));
78
79      if (result == ESP_OK) {
80        Serial.println("Daten gesendet:");
81        Serial.print("  Temperatur: "); Serial.print(t); Serial.println(" °C");
82        Serial.print("  Feuchtigkeit: "); Serial.print(h); Serial.println(" %");
83      } else {
84        Serial.println("Fehler beim Senden der Daten");
85      }
86    }
87
```

ESP32

4.1 Als nächstes muss des ESP32 so konfiguriert werden, dass er die versendeten Nachrichten des C3 empfangen kann. Dafür muss auf der Empfängerseite ESP-NOW konfiguriert werden. Damit die ESP-NOW-Kommunikation verfiziert werden kann, wird zunächst nur ESP-NOW auf dem ESP32 eingerichtet und als Debugging-Unterstützung die empfangene Nachricht, sowie die MAC-Adresse des Senders im seriellen Monitor der Arduino IDE ausgegeben.

Zuerst werden die beiden Bibliotheken *esp_now.h* und *WiFi.h* benötigt, damit ESP-NOW verwendet werden kann und eine Programmierschnittstelle zur Verfügung steht (ESP32_ESPNOW).

```
#include <esp_now.h>
#include <WiFi.h>

```

Als nächstes wird wieder ein *struct f*ür die Nachricht definiert, mit zwei Variablen für die Messwerte. Dieser muss einheitlich zum Sender sein. Jetzt wird eine Variable vom Typ dieser Struktur deklariert werden, *incomingData*, in der dann später die empfangenen Messwerte gespeichert werden können.

```
// Beispielstruktur für empfangene Daten
typedef struct struct_message {
  float temperatur;
  float feuchtigkeit;
} struct_message;

// Empfangene Daten speichern
struct_message incomingData;

```

Auch der Empfänger benötigt eine Callback-Funktion. Der Funktion *OnDataRecv* wird *esp_now_recv_info_t *mac* übergeben, was die MAC-Infos des Absender enthält, die übertragenen Bytes in *incomingDataBytes* und die Länge der Nachricht mit *len*. Zur Kontrolle werden in der Funktion alle empfangenen Informationen ausgelesen und im seriellen Monitor ausgegeben. Bei korrekter Ausführung wird nun periodisch die Temperatur, Feuchtigkeit und MAC-Adresse des Senders im seriellen Monitor ausgegeben.

```
// Callback-Funktion bei Datenempfang
void OnDataRecv(const esp_now_recv_info_t *mac, const uint8_t *incomingDataBytes, int len) {
  memcpy(&incomingData, incomingDataBytes, sizeof(incomingData));
  // Mac des Absenders auslesen
  Serial.print("Empfangen von: ");
  char macStr[18];
  snprintf(macStr, sizeof(macStr), "b0:81:84:04:62:28",
           mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
  Serial.println(macStr);


  Serial.print("Temperatur: ");
  Serial.println(incomingData.temperatur);
  Serial.print("Feuchtigkeit: ");
  Serial.println(incomingData.feuchtigkeit);
  Serial.println("-------------------------");
}

```

In *setup()* wird wie gewohnt eine Baudrate initialisiert und der WiFi-Mode gesetzt. Bevor ESP-NOW initialisiert wird, wird über *WiFi.disconnect* jede Verbindung geschlossen, sodass der Funkkanal für ESP-NOW frei ist. Die Callback-Funktion wird registriert, damit wird dem ESP32 mitgeteilt, welche Funktion aufgerufen werden soll, sobald eine Nachricht eingeht. Eine kleine Debug-Info zeigt, ob das Setup komplett durchlaufen wird und teilt mit, ob der Empfänger bereit ist.

```
31  void setup() {
32    Serial.begin(115200);
33
34    // WiFi auf "Station Mode"
35    WiFi.mode(WIFI_STA);
36    WiFi.disconnect(); // Sicherstellen, dass keine Verbindung besteht
37    delay(100);
38
39    // ESP-NOW initialisieren
40    if (esp_now_init() != ESP_OK) {
41      Serial.println("Fehler beim Initialisieren von ESP-NOW");
42      return;
43    }
44
45    // Callback registrieren
46    esp_now_register_recv_cb(OnDataRecv);
47    // Debug-Info
48    Serial.println("ESP-NOW Empfänger bereit");
49  }
50
51  void loop() {
52    // nichts tun – warten auf Daten
53  }
```

In *loop()* passiert hier nichts, alles ist über Callbacks regelbar. ESP-NOW funktioniert ohne Loop der permanent laufen muss, was für seine Energieeffizienz spricht.

4.2 Nun soll ein lauffähiger BLE-Sender auf dem ESP32 konfiguriert werden. Dazu werden die Bibliotheken *BLEDevice.h* (Initialisierung und Verwaltung des BLE-Geräts), *BLEServer.h* (Serverfunktionen bereitstellen), *BLEUtils.h* (Hilfsfunktionen) und *BLE2902.h* (Standard-Descriptor, über den Clients Notifications ein- oder ausschalten können) benötigt (ESP32_BLE.ino).

```
1  #include <BLEDevice.h>
2  #include <BLEServer.h>
3  #include <BLEUtils.h>
4  #include <BLE2902.h>
5
```

Der Pointer *pCharacteristic* vom Typ *BLECharacteristic* ist ein Zeiger auf die BLE Charakteristik. Die Charakteristik von BLE ist das Zentrale Element des Datenaustausch zwischen Server und Client. Sie beinhaltet eine eindeutige Kennung (UUID = Universally Unique identifier), die Nutzdaten (*value*) und definiert wie die Clients auf die Daten des Servers zugreifen dürfen.

Der Descriptor **p2902Descriptor* wird später im Code dazu verwendet, um abzufragen, ob der Server dem Client Daten per *notify()* oder *indicate()* senden darf. Nur wenn auf Clientseite der Descriptor 0x2902 explizit auf *true* gesetzt wird, darf er Nachrichten empfangen. Der Client hat in BLE somit volle Kontrolle, ob er Daten erhält oder nicht.

```
6   BLECharacteristic *pCharacteristic;
7   BLE2902 *p2902Descriptor;   // Descriptor, um zu prüfen ob Notifications aktiv sind
8   bool deviceConnected = false;
9
```

Zudem wird eine Flag benötigt um dem Server mitzuteilen, ob ein gerät verbunden ist. Dafür wird *deviceConnected* zunächst mit *false i*nitialisiert, da bei Serverstart noch kein Gerät verbunden ist.

Jetzt werden die ersten Charakteristika von BLE definiert, die SERVICE_UUID und die CHARACTERISTIC_UUID. Die SERVICE_UUID ist eine Kennnummer, für einen Dienst, den der Server anbietet. Der Dienst ist in diesem Fall, die Messwerte des Sensors in regelmäßigen Abständen an den Client zu übermitteln.

```
10  #define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
11  #define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"
12
```

Die CHARACTERISTIK_UUID identifiziert eine einzelne Datenvariable im Service. Der Client sieht, dass es im Service eine Charakteristik mit dieser UUID gibt, die er lesen bzw. abonnieren kann.

Für die Callbacks wird eine Klasse erzeugt, die von der Klasse BLEServerCallbacks, aus der *BLEServer.h*-Bibliothek, erbt. Die Funktionen o*nConnect* wird so verändert, dass wenn sich ein Gerät/Client verbindet man eine Debug-Information erhält und der Status von deviceConnected auf true gesetzt wird. Die Funktion *onDisconnect*, die ebenfalls aus der Klasse *BLEServerCallbacks* vererbt wurde, wird aufgerufen, wenn sich ein Gerät vom Server trennt. Der Status von *deviceConnected* wird dann wieder auf f*alse* gesetzt und der Server startet wieder mit dem Werben(*Advertising*) für neue Clients.

```
13  class MyServerCallbacks : public BLEServerCallbacks {
14    void onConnect(BLEServer* pServer) {
15      deviceConnected = true;
16      Serial.println("Gerät verbunden.");
17    }
18
19    void onDisconnect(BLEServer* pServer) {
20      deviceConnected = false;
21      Serial.println("Gerät getrennt.");
22      delay(500);  // kurze Pause für Browser
23      pServer->getAdvertising()->start();
24      Serial.println("Werbung neu gestartet.");
25    }
26  };
27
```

Auch hier gibt es eine kleine Ausgabe im seriellen Monitor der IDE (Debugging).

In *setup()* erhält der ESP32 beim Initialisieren den Gerätenamen „*ESP32_BLE_Device*“. BLEServer **pServer* erzeugt einen Pointer auf einen Server, der auf dem zuvor initialisierten BLEDevice mit dem Namen „*ESP32_BLE_Device*“ gestartet wird. Über den Pointer **pServer* kann nun auf den

erstellten Server zugegriffen werden, z.B. um ein *Advertising (pServer->getAdvertising->start()*) zu starten etc.

```
28  void setup() {
29    Serial.begin(115200);
30    delay(1000);
31
32    Serial.println("Starte BLE-Server...");
33
34    // BLE initialisieren
35    BLEDevice::init("ESP32_BLE_Device");
36    // BLE Server erzeugen
37    BLEServer *pServer = BLEDevice::createServer();
38    //Callback-Klasse zuweisen
39    pServer->setCallbacks(new MyServerCallbacks());
40
```

Hier wird der Pointer *pServer* dazu verwendet, die zuvor erstellte Callback-Klasse *MyServerCallbacks* dem Server zuzuweisen. Sobald sich nun der Verbindungsstatus eines Gerätes zum Server ändert, wird eine der beiden überschriebenen Funktionen *onConnect* bzw *onDisconnect aufgerufen.*

Jetzt kann dem Server auch die zuvor definierte SERVICE_UUID zugewiesen werden. Dazu wird einem Pointer vom Typ BLEService initialisiert, der auf den Service zeigt, der über den „Server-Zugriffs-Pointer“ **pServer* mit der Funktion *createService* und der zuvor angelegten SERVICE_UUID erstellt wird. **pServic*e kann für den Zugriff auf den vom Server angebotenen Service verwendet werden.

```
41    // BLE Service mit UUID erstellen
42    BLEService *pService = pServer->createService(SERVICE_UUID);
43
```

Die UUIDs sind hierarchisch aufgebaut. Die obere Ebene ist die SERVICE_UUID die definiert, welcher Service angeboten wird (Temperatur und Luftfeuchtigkeit). Die CHARACTERISTIC_UUID ist die untere Ebene, die angibt wo die Werte gespeichert bzw. abgerufen werden können. Sie beschreibt den konkreten Datenpunkt innerhalb des Services (z. B. die aktuelle Temperatur- und Luftfeuchtigkeitswerte), während die SERVICE_UUID nur den Rahmen vorgibt. Also kann der **pService* Pointer, dem zuvor ein Service zugeteilt wurde, verwendet werden, diesen Service durch Charakteristiken (den Datenpunkt, in dem die Werte des Services zu finden sind) zu spezifizieren. **pCharacteristic* zeigt nun auf diese Charakteristik und stellt den Zugriff auf die Charkteristiken des Servers dar. Über die Funktion c*reateCharacteristic* kann nicht nur die CHARACTERISTIK_UUID dem Service zugeordnet werden, sondern auch die Art, wie der Client auf die Charakteristiks zugreifen kann.

```
44  // Charakteristik mit READ + NOTIFY
45  pCharacteristic = pService->createCharacteristic(
46    CHARACTERISTIC_UUID,
47    BLECharacteristic::PROPERTY_READ | // Client kann den Wert diekt abfragen
48    BLECharacteristic::PROPERTY_NOTIFY // Server kann Änderunge an Clients pushen
49  );
50
```

Der p2902Descriptor legt fest ob Nachrichten an den Client geschickt werden dürfen. Die Funktion, Nachrichten (*Notifications)* an den Client zu pushen ist zunächst auf *false* gesetzt, der Client muss den Zustand im Code selbst explizit auf t*rue* setzen.

```
51  // Descriptor hinzufügen (BLE Standart)
52  p2902Descriptor = new BLE2902();
53  p2902Descriptor->setNotifications(false);
54  pCharacteristic->addDescriptor(p2902Descriptor);
55
```

Jetzt werden die Startwerte der Charakteristik auf 0.0 gesetzt.

```
56  // Startwert setzen
57  pCharacteristic->setValue("0.0,0.0");
58
```

Nachdem alle Charakteristiken erstellt sind, kann der Service gestartet werden.

```
59  // Service starten
60  pService->start();
```

Damit der Server für Clients sichtbar und abonnierbar ist, muss der Server die Werbung (Advertising) starten. Der Pointer **pAdvertising* vom Typ BLEAdvertising zeigt auf das Advertising des Servers, das über den „Server-Zugriffs-Pointer“ **pServer* mit dem BLE-Server verbunden wird. Dem Advertising kann nun die SERVICE_UUID zugeordnet werden.

```
62  // Advertising konfigurieren
63  BLEAdvertising *pAdvertising = pServer->getAdvertising();
64  pAdvertising->addServiceUUID(SERVICE_UUID);
65
```

Jetzt wird der Takt festgelegt, in dem Client und Server miteinander kommunizieren, also wie oft pro Sekunde Pakete ausgetauscht werden. Desto größer das Intervall ist, zwischen dem Austausch von Nachrichten, desto stromsparender wird der Betrieb der BLE-Kommunikation. Kurze Taktfrequenzen haben einen höheren Stromverbrauch

```
66    // Stabileres Connection-Interval
67    pAdvertising->setMinPreferred(0x06);  // 30 ms
68    pAdvertising->setMaxPreferred(0x12);  // 90 ms
69
```

Da **pAdvertising* auf den Service zeigt, den der Server anbietet, lässt sich darüber auch der Service starten, sodass der Server nun mit dem Service wirbt, Temperatur- und Feuchtigkeitswerte zu versenden. Clients können nun sehen, dass dieser Service angeboten wird.

```
70    pAdvertising->start();
71    Serial.println("BLE Server gestartet – Werbung läuft...");
72    Serial.println("Warte auf Bluetooth-Verbindung...");
73  }
74
```

In *loop()* wird geprüft, ob ein Gerät mit dem Server Verbunden ist und **p2902Descriptor* prüft ob setNotifications von Client auf *true* gesetzt wurde. „Falls ein Client verbunden ist und der Client Nachrichten vom Server erlaubt hat“, werden für Temperatur und Luftfeuchtigkeit zunächst zwei *random* Werte erzeugt die verschickt werden sollen. Dies dient zur Kontrolle, ob die eingerichtete Verbindung auch wirklich funktioniert.

```
75  void loop() {
76    if (deviceConnected && p2902Descriptor->getNotifications()) {
77      // Beispielwerte (Temp. & Luftfeuchtigkeit)
78      float temp = random(200, 300) / 10.0;  // 20.0 - 30.0 °C
79      float hum  = random(400, 700) / 10.0;  // 40.0 - 70.0 %
80
81      char buffer[32];
82      snprintf(buffer, sizeof(buffer), "%.1f,%.1f", temp, hum);
83
84      // Wert setzen und senden
85      pCharacteristic->setValue(buffer);
86      pCharacteristic->notify();
87
88      // Seriell ausgeben
89      Serial.print("Gesendet: ");
90      Serial.println(buffer);
91
92      delay(2000);
93    }
94  }
```

Für die Messwerte wird ein Buffer deklariert, in dem die Werte für Temperatur und Luftfeuchtigkeit gespeichert werden (Charakteristik des Services).

Da es sich bei den Werten die verschickt werden um eine Charakteristik handelt, wird auf diese auch mit dem dafür zuständigen Pointer **pCharacteristik* zugegriffen,. Die Charakteristik die im Buffer gespeichert ist, wird an g*etValue*() übergeben und über *notify()* an den verbundenen Client gesendet.

4.3 Der BLE Server soll mit der Google Chrome Bluetooth Web API kommunizieren, einer Browser-Erweiterung für moderne Browser wie Google Chrome und Edge, die es ermöglicht sich mit Bluetooth-Geräten zu verbinden und Daten über den Browser zu empfangen.

Dazu genügt eine einfache html-Seite, die die Möglichkeit bietet, nach aktiven Bluetooth-Geräten in der Umgebung zu suchen und sich mit diesen zu verbinden, sowie zwei Feldern, die bei erfolgreicher Verbindung die Werte für Temperatur und Feuchtigkeit anzeigen.

Im Client müssen die gleichen UUIDs definiert werden, wie beim Server, damit der Client den Service der Servers abonnieren kann und die Charakteristiken empfangen kann

Der Client muss die Notifications auf true setzen, um Nachrichten des Servers zu empfangen.

Der Client sollte sich immer wieder automatisch verbinden, sodass ein konstanter Datenstrom vorliegt und die Verbindung nicht abbricht, wenn im nächsten Schritt ESP-NOW und BLE sich zusammen den Funkkanal teilen müssen.

Die Funktion *handleData()* decodiert die Rohdaten und formatiert sie so, dass der Wert für die Temperatur in dem Temperaturfeld der html-Seite angezeigt wird und der Wert für die Luftfeuchtigkeit im dafür vorgesehenen Feld für die Luftfeuchtigkeit.

Damit überhaupt nach Bluetooth-Verbindungen vom Browser aus gesucht werden kann, muss dem Google Chrome Browser beim Start mitgeteilt werden, Bluetooth-Verbindungen zu erlauben. Dazu muss Chrome über die TTY mit folgendem Befehl gestartet werden:

google-chrome --enable-experimental-web-platform-features –enable-bluetooth

Damit die html-Seite auch im Browser aufgerufen werden kann, muss im TTY ein kleiner Webserver im Zielverzeichnis der html-Seite gestartet werden.
In den Ordner wechseln, in dem die html-Seite gespeichert ist :

cd /Pfad/zur/html-Seite

Webserver starten :

python3 -m http.server 8080

Da der Webserver im gleichen Verzeichnis gestartet wurde, in der auch die html-Seite hinterlegt ist, kann diese nun über den Browser mit *localhost* und dem Port 8080 erreicht werden. Die für das zur Umsetzung des Projekts verwendete html-Datei heißt „bluetoothWebAPI.html“.

Dementsprechend lässt sich die Seite nun mit http://localhost:8080/bluetoothWebApi.html öffnen.

Jetzt ist der Browser in der Lage sich mit dem ESP32 über BLE zu verbinden und die Messwerte des Servers zu empfangen und die anzuzeigen.

Wenn die auf dem ESP32 zufällig generierten Werte nun im Browser dargestellt werden, wurde erfolgreich eine BLE-Verbindung zwischen Browser und ESP32 aufgebaut.

Der Code für den html-Server befindet sich in den beigefügten Codes.

4.3 Abschließend soll neben BLE auch ESP-NOW zum empfangen von Nachrichten auf dem ESP32 konfiguriert werden, da schließlich die Messwerte des DHT11 im Browser angezeigt werden sollen und keine zufällig generierten Werte.

ESP-NOW und BLE wurden jeweils zum laufen gebracht, jedoch lassen sich diese beiden Sketche nicht ohne weiteres miteinander kombinieren, da ESP-NOW und BLE sich den gleichen Funkkanal auf dem ESP32 teilen müssen. Also muss noch geregelt werden, wann welches Protokoll den Funkkanal verwendet. Der Zugriff auf den Funkkanal lässt sich mit Hilfe der Bibliothek *esp_wifi.h* regeln. Die Bibliothek ermöglicht es, den beiden Protokollen einen festen Kanal zuzuweisen, sodass sie sich nicht in die Quere kommen beim Zugriff auf die Funkhardware des ESP32.

Neu ist zudem eine Variable in der die empfangenen Messwerte zwischengespeichert werden können, damit BLE immer auf konsistente Daten zugreifen kann (ESP32_BLE_ESPNOW.ino).

```
#include <esp_now.h>
#include <WiFi.h>
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <esp_wifi.h>

// UUIDs für BLE
#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

// BLE Variablen
BLECharacteristic *pCharacteristic;
BLE2902 *p2902Descriptor;
bool deviceConnected = false;

// Struktur für empfangene Sensordaten
typedef struct struct_message {
  float temperature;
  float humidity;
} struct_message;

struct_message cachedData;   // Zwischengespeicherte Daten
volatile bool newDataAvailable = false;
```

Zudem wird jetzt eine *boolsche* Variable benötigt, die angibt, ob neue Daten vorhanden sind (*true*) oder nicht (*false*). Beim Start ist dieser Wert logischerweise auf *false* gesetzt, da zu diesem Zeitpunkt noch keine Daten vorliegen.

Die Callback-Funktion bleibt nahezu gleich, außer, dass die über ESP-NOW empfangenen Daten in dem *message_struct cachedData* zwischengespeichert werden und der Zustand von *newDataAvailable* auf *true* gesetzt wird, da die Callback-Funktion nur aufgerufen wird, wenn neue Daten empfangen wurden und somit vorliegen.

```
27  // ESP-NOW Callback
28  void OnDataRecv(const esp_now_recv_info_t *info, const uint8_t *incomingDataBytes, int len) {
29    if (len != sizeof(cachedData)) return;
30
31    memcpy(&cachedData, incomingDataBytes, sizeof(cachedData));
32    newDataAvailable = true;
33
34    char macStr[18];
35    snprintf(macStr, sizeof(macStr), "%02X:%02X:%02X:%02X:%02X:%02X",
36             info->src_addr[0], info->src_addr[1], info->src_addr[2],
37             info->src_addr[3], info->src_addr[4], info->src_addr[5]);
38
39    Serial.printf("Von %s empfangen: T=%.1f°C H=%.1f%%\n",
40                  macStr, cachedData.temperature, cachedData.humidity);
41  }
42
```

Die BLE-Callbacks sind jetzt nur noch dafür zuständig, den Status von *deviceConnected* entweder auf *true* oder *false* zu setzen, je nachdem ob ein Gerät verbunden (*true*) ist oder nicht (*false*)

```
43  // BLE Server Callbacks
44  class MyServerCallbacks : public BLEServerCallbacks {
45    void onConnect(BLEServer *pServer) {
46      deviceConnected = true;
47      Serial.println("BLE Gerät verbunden");
48    }
49    void onDisconnect(BLEServer *pServer) {
50      deviceConnected = false;
51      Serial.println("BLE Gerät getrennt");
52    }
53  };
54
```

Zum Starten von ESP-NOW wird nun eine eigene Funktion definiert, die immer wieder aufgerufen werden kann um ESP-NOW richtig zu starten. Darin wird wie gehabt der WiFi-Mode auf *stationary* gesetzt, ESP-NOW initialisiert und die Callback-Funktion registriert. Neu dazugekommen ist die Funktion e*sp_wifi_set_channel*(6, WIFI_SECOND_CHAN_ONE) aus der *esp_wifi.h* Bibliothek. ESP-NOW wird dabei gezwungen auf Kanal 6 zu arbeiten und das untere Hälfte des Frequenzbandes des Funkkanals dafür zu verwenden.

```
55  //ESP-NOW Start
56  void startESPNOW() {
57    WiFi.mode(WIFI_STA);
58    if (esp_now_init() != ESP_OK) {
59      Serial.println("Fehler bei ESPNOW Init");
60      return;
61    }
62    esp_now_register_recv_cb(OnDataRecv);
63
64    // Fester Kanal, damit BLE/ESP-NOW stabil laufen
65    esp_wifi_set_channel(6, WIFI_SECOND_CHAN_NONE);
66
67    Serial.println("ESPNOW gestartet und Kanal gesetzt");
68  }
69
```

Wenn s*tartBLE()* aufgerufen wird, wird wie gehabt BLE initialisiert und gestartet.

Dem ESP32 wird ein sichtbarer Gerätename gegeben, unter dem er zu finden ist.

```
70  // BLE Start
71  void startBLE() {
72    Serial.println("Starte BLE...");
73
74    BLEDevice::init("ESP32_Sensor");
```

**pServer* = Zeiger auf dieses Server-Objekt, über das Services und Charakteristiken angelegt werden.

```
75    BLEServer *pServer = BLEDevice::createServer();
76    pServer->setCallbacks(new MyServerCallbacks());
77
```

Service erstellen, **pService* = Zeiger auf den Services und Charakteristik im Service anlegen

```
78    BLEService *pService = pServer->createService(SERVICE_UUID);
79    pCharacteristic = pService->createCharacteristic(
80                        CHARACTERISTIC_UUID,
81                        BLECharacteristic::PROPERTY_NOTIFY
82    );
```

Descriptor2902 hinzufügen, zum Prüfen, ob der Client Nachrichten empfangen kann.

```
83    p2902Descriptor = new BLE2902();
84    p2902Descriptor->setNotifications(true);
85    pCharacteristic->addDescriptor(p2902Descriptor);
86
87    pService->start();
88
```

Den Service starten.

**pAdvertising*, zeigt auf die Werbung des Servers, Der Server hat nun die Advertising Funktion

```
89    BLEAdvertising *pAdvertising = pServer->getAdvertising();
90    pAdvertising->addServiceUUID(SERVICE_UUID);
91    pAdvertising->setScanResponse(true);
92    pAdvertising->setMinPreferred(0x06);
93    pAdvertising->setMaxPreferred(0x12);
94    pAdvertising->start();
95
96    Serial.println("BLE Advertising gestartet, warte auf Verbindung...");
97  }
```

SERVICE_UUID dem Advertising hinzufügen und zusätzliche Scan Response aktivieren, dabei wird ein zweites Datenpaket beim Scannen verschickt. Hilfreich, wenn man mehr Infos schicken will(z.B Gerätenamen).

Verbindungsintervalle setzen auf:

0x06→ ca. 7,5 ms

0x12 → ca. 22,5 ms

Das Advertising starten. ESP32 ist jetzt im BLE-Scan sichtbar.

Wenn ein Client die Verbindung abbricht, wird der Status von *deviceConnected* auf *false* gesetzt und der Server startet wieder das *Advertising*, damit der Server wieder sichtbar wird für Clients.

In *setup*() wird als erstes Funktion *startBLE*() aufgerufen. Die zuvor beschriebene Funktion erzeugt einen BLE-Server und diesem wird die Callback-Klasse zugewiesen. Über die Service-UUID wird ein BLE-Service erzeugt. Clients denen die Service-UUID bekannt ist, können den BLE-Server abonnieren. In *pCharacteristic* wird die Charakteristik des Services festgelegt. In diesem Fall, Clients dürfen lesen und und müssen empfangen.

Nachdem ein Startwert für Temperatur und Luftfeuchtigkeit gesetzt wurde, wird der Service *pService* gestartet mit *start*() und ist somit verfügbar.

Als nächstes wird die Werbung eingerichtet, damit teilt der ESP32 mit, dass er einen Service mit der angegebenen UUID anbietet. *pAdvertising→start*() startet das Werben. Zur Kontrolle wird wieder eine Ausgabe im seriellen Monitor erzeugt.

In *setup*() wird hier zuerst BLE gestartet, damit sich Clients verbinden können. Danach wird ESP-NOW mit der Funktion s*trartESPNOW()* gestartet, zum empfangen von Nachrichten.

```
 99  void setup() {
100    Serial.begin(115200);
101    delay(2000);
102
103    Serial.println("Starte ESP-NOW + BLE Gateway...");
104
105    // Zuerst BLE starten
106    startBLE();
107    delay(500); // kurze Pause für BLE-Stack
108
109    // Danach ESP-NOW aktivieren
110    startESPNOW();
111  }
```

Im *loop*() wird geprüft ob ein Gerät mit dem ESP32 verbunden ist und ob neue Daten zur Verfügung stehen. Wenn beides der Fall ist, werden die im *cachedData* zwischengespeicherten Daten in der Buffer geschrieben.

```
113  void loop() {
114    // Wenn neue Daten vorhanden sind und ein BLE-Client verbunden ist → senden
115    if (newDataAvailable && deviceConnected) {
116      char buffer[32];
117      snprintf(buffer, sizeof(buffer), "%.1f,%.1f", cachedData.temperature, cachedData.humidity);
```

Der Charakteristik wird der Inhalt/Wert des Buffers übergeben

```
118        pCharacteristic->setValue(buffer);
```

Charakteristik wird dem Client der verbunden ist und den Service abonniert hat mitgeteilt.

```
119        pCharacteristic->notify();
```

Nachdem die Daten ausgelesen wurden, wird das Flag von *newDataAvailable* wieder auf *false* gesetzt, bis wieder neue Daten empfangen und in *cachedData* zwischengespeichert wurden.

```
120
121      Serial.print("BLE gesendet: ");
122      Serial.println(buffer);
123
124      newDataAvailable = false; // Flag zurücksetzen
125    }
126
127    delay(1); // wichtig: CPU-Zeit für BLE/WiFi Tasks freigeben
128  }
129
```

Der Datenfluss vom DHT11 bis zum Browser über ESP-NOW und BLE wurde erfolgreich umgesetzt.

Beide Protokolle können jetzt koexistieren, da ESP-NOW dazu gezwungen wird, nur einen Kanal zu verwenden (6). So gibt es nur noch diesen einen Kanal auf dem sich ESP-NOW und BLE in die Quere kommen können, da BLE auf den Kanälen 1-37 Daten sendet. Ohne Kanalbindung würde ESP-NOW die Kanäle 1-13 zum Datenaustausch verwenden, was den kompletten Datenverkehr stören würde, da ESP-NOW und BLE dann zu oft im gleichen Frequenzbereich funken würden. Die Kanalzuweisung ist daher zwingend erforderlich, damit beide Funkprotokolle zeitgleich arbeiten können. Der Zugriff auf den Funkkanal kann in diesem Fall noch von der Firmware des ESP32 intern geregelt werden, sodass Kollisionen vermieden werden.

Der ESP32 ist nun in der Lage, Daten per ESP-NOW zu empfangen, zwischenzuspeichern und über BLE an einen Web-Browser weiter zu leiten. Nachdem sich der Server mit dem Browser verbunden hat, werden die Temperatur- und Feuchtigkeitswerte konstant vom Server an den Browser geschickt und aktualisiert.

Das Projekt lässt sich mit kleinen Anpassungen mit nahezu jedem Sensor verwenden. Die Kommunikation vom Browser zum C3 ist mit diesem Konzept auch realisierbar. Die Funktion Daten per ESP-NOW zu empfangen und zu senden sind Teil der Umsetzung des Projekts.

Eine weitere Optimierung wäre es, den C3 immer wieder in einen Deep-Sleep zu versetzen, wenn keine Daten versendet werden. ESP-NOW ermöglicht es extrem energieeffizient zu arbeiten. In den beiliegenden Sketchen befindet sich auch eine Deep-Sleep Version des ESP32-C3.

Quellen

Espressif Systems. (2019). *ESP32 Series Datasheet*. Abgerufen von https://www.espressif.com/en/products/socs/esp32

Espressif Systems. (2020). *ESP32-C3 Series Datasheet*. Abgerufen von https://www.espressif.com/en/products/socs/esp32-c3

Espressif Systems. (o. J.). *ESP-NOW User Guide*. Abgerufen von https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html

Espressif Systems. (o. J.). *ESP-IDF API Reference – esp_wifi.h*. Abgerufen von https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_wifi.html

Espressif Systems. (o. J.). *ESP-IDF API Reference – Bluetooth Low Energy*. Abgerufen von https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/bluetooth/index.html

Arduino. (o. J.). *DHT11 Temperature and Humidity Sensor with Arduino*. Abgerufen von https://docs.arduino.cc/tutorials/generic/dht

Santos, R. (o. J.). *ESP-NOW with ESP32: Send Data (one-to-many, broadcast)*. Random Nerd Tutorials. Abgerufen von https://randomnerdtutorials.com/esp-now-esp32-arduino-ide/

Santos, R. (o. J.). *ESP32 BLE Server and Client (Arduino IDE)*. Random Nerd Tutorials. Abgerufen von https://randomnerdtutorials.com/esp32-ble-server-client/

Giraldo, C. (2019). *Introduction to Bluetooth Low Energy (BLE)*. Medium. Abgerufen von https://medium.com/@carlosgiraldoa/introduction-to-bluetooth-low-energy-ble-19c56c70df1e

Google Developers. (o. J.). *Web Bluetooth API*. Abgerufen von https://developer.chrome.com/articles/web-bluetooth/

Mozilla Developer Network (MDN). (o. J.). *Web Bluetooth API*. Abgerufen von https://developer.mozilla.org/en-US/docs/Web/API/Web_Bluetooth_API

Infineon Technologies. (2021). *Wi-Fi and Bluetooth Coexistence*. Abgerufen von https://community.infineon.com/t5/Blogs/Wi-Fi-and-Bluetooth-Coexistence/ba-p/249063