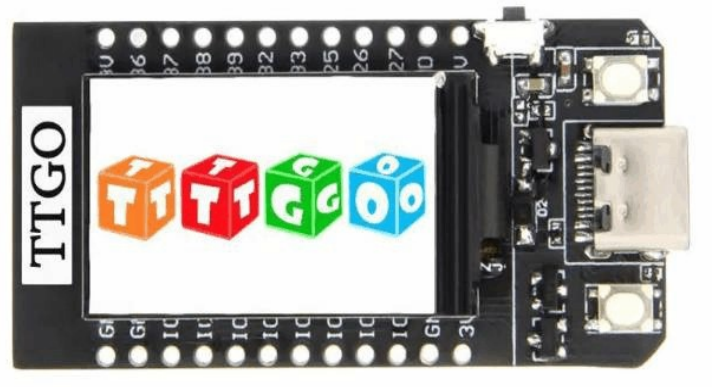
Bluetooth Low Energy

Eine praktische Einführung mit Micropython und dem ESP32-Board TTGO T-Display



von

G. Heinrichs

Mönchengladbach, 08.03.2022

Vorwort

Im Internet findet man zum Thema **B**luetooth **L**ow **E**nergy (kurz: BLE) eine ganze Reihe von fertigen - und teilweise auch ordentlich kommentierten - Programmen für den ESP32. Meist sind sie aber auf spezielle Aspekte ausgerichtet; so ist es für Einsteiger oft nicht einfach, einen Überblick über verschiedene Anwendungen zu erhalten und gleichzeitig auch grundlegende Einsichten in die Funktionsweise von BLE zu erlangen.

Dieses Skript möchte nun zweierlei: Zum Einen sollen hier - nach Schwierigkeitsgrad gestaffelt - eine Reihe von einfachen, aber ausbaufähigen Beispielen zur Anwendung von BLE mit dem ESP32 vorgestellt werden; zum Anderen sollen aber auch wichtige Begriffe und Konzepte (Advertiser, Scanner, BLE-Datenpakete, Central und Peripheral, Connecting, GATT und GAP, um einige zu nennen) so dargelegt werden, wie wir sie für ein Verständnis der BLE-Programmierung erforderlich halten. Ziel ist es, dem Leser damit grundlegende Kenntnisse zu vermitteln und bei ihm eine tragfähige Vorstellung zu entwickeln, die es ihm erlauben, die vorgestellten Programme weiter auszubauen und neue Anwendungen zu realisieren.

Ganz bewusst habe ich in vielen Fällen die englischen Fachausdrücke benutzt und auf den Versuch einer Übersetzung ins Deutsche verzichtet. Zwei Gründe dafür möchte ich nennen: Zum Einen gibt es häufig gar keine entsprechenden deutschen Ausdrücke. Zum Anderen sind die meisten Bücher und Internet-Beiträge zu unserem Thema in englischer Sprache geschrieben; da fällt das Lesen leichter, wenn man gleich die englischen Fachausdrücke kennen gelernt hat.

Als Programmiersprache werden wir **Micropython** verwenden. Hierbei handelt es sich um eine Variante der Programmiersprache **Python**, die speziell für Mikrocontroller konzipiert wurde: Micropython benutzt dieselbe Syntax wie Python, aber im Kern besitzt sie nur eine Teilmenge der Python-Befehle; hinzu kommt allerdings noch eine Reihe von Mikrocontroller-spezifischen Befehlen.

Python wurde ursprüngliche zu Lehrzwecken konzipiert; entsprechend einfach ist die Syntax gehalten und entsprechend wenig Schlüsselwörter sind zu lernen. Inzwischen wird Python aber auch professionell eingesetzt.

Welche Voraussetzungen sollten Sie, lieber Leser, erfüllen? Nun, zunächst sollten Sie ein ESP32-Board und ein wenig Zubehör besitzen. Das werden Sie benötigen, um eigene Erfahrungen mit der BLE-Programmierung zu sammeln. Zum Board selbst und dem benötigten Zubehör werde ich im nächsten Kapitel (Vorbereitung) noch mehr sagen. Sodann setze ich voraus, dass Sie zur Programmierung des ESP32 einen Rechner mit Windows 10 zur Verfügung haben. Die benutzte Software **Thonny** (kostenlos erhältlich, s. nächstes Kapitel) gibt es zwar auch für andere Betriebssysteme, auf diese werde ich aber nicht näher eingehen.

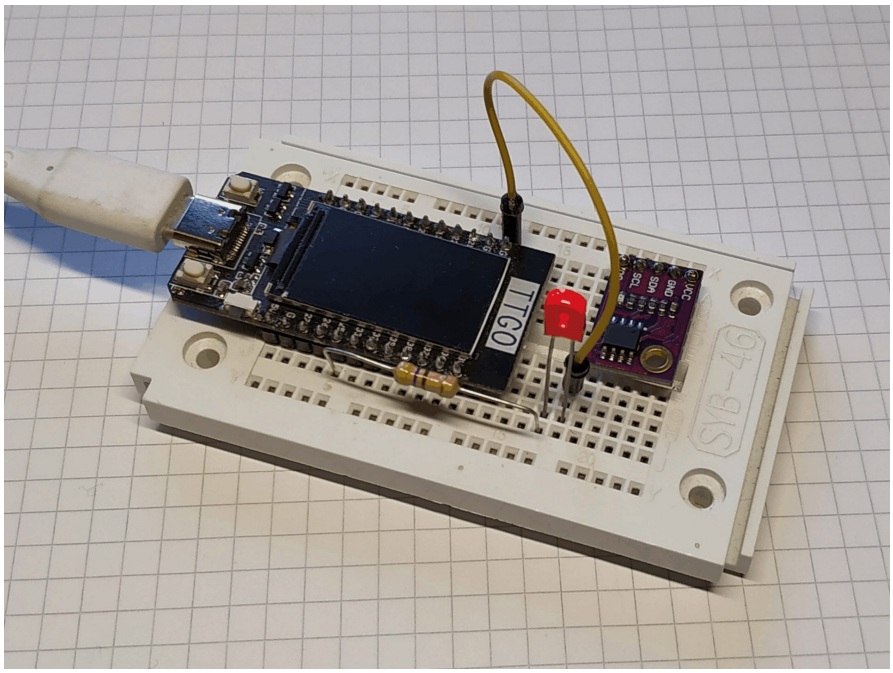


Abb. 1: ESP32-Board (TTGO T-Display) und einige zusätzliche Bauteile

Die einzelnen Kapitel dieses Skripts bauen auf einander auf. Sie sollten also **die einzelnen Kapitel der Reihe nach durcharbeiten**. Wenn Sie sich im Laufe der Lektüre an den einen oder anderen Fachausdruck nicht mehr genau erinnern können, schauen Sie einfach im **Stichwortverzeichnis** nach; dort wird diejenige Seite angegeben, auf der Fachausdrücke und Abkürzungen eingeführt worden sind.

Gut wäre es, wenn Sie schon etwas Erfahrung im Programmieren von Mikrocontrollern mitbringen. In diesem Fall sollte unsere kurze Micropython-Einführung im übernächsten Kapitel ausreichen, um die anschließenden Kapitel zur BLE-Programmierung verstehen zu können.

Sollten Sie keinerlei Programmier-Erfahrung besitzen, empfehle ich Ihnen, zunächst die ersten zehn Kapitels des **Buches "Python 3" von Michael Bonacina** zu lesen. Jeder Befehl, jede Struktur, insbesondere aber auch die Grundlagen der Programmierung werden anschaulich und gut verständlich erklärt. Zahlreiche Beispiele und Übungen unterstützen den Lernvorgang. In diesem Buch wird zwar nicht mit Micropython, sondern mit Python 3 gearbeitet. Die behandelten Beispiele lassen sich aber (mit ganz wenigen Ausnahmen) auch mit der Thonny-Entwicklungsumgebung und Micropython studieren. Aus diesem Grunde ist es auch nicht erforderlich, die im Buch benutzte Python-Software zu installieren.

Ein herzlicher Dank geht an dieser Stelle an meinen Sohn Michael: Er hatte mich dazu gebracht, mich mit dem TTGO T-Display zu beschäftigen, indem er mir ein solches Board zum Geburtstag schenkte. Zudem hat er mich unermüdlich immer wieder beim Schreiben dieses Skripts unterstützt, sei es durch kritische Bemerkungen, sei es durch wertvolle Hinweise oder Anregungen.

Inhaltsverzeichnis

V Vorbereitung

Hardware

Es gibt zahlreiche ESP32-Boards. Sie unterscheiden sich hauptsächlich darin, welche Anschlüsse des ESP32 nach außen geführt sind und welche Module zusätzlich auf dem Board montiert sind. Wir werden hier mit der Variante "TTGO T-Display" arbeiten. Sie ist preislich erschwinglich (ab ca. 10 Euro) und bietet ein fest verdrahtetes Farb-Display mit einer Auflösung von 240 x 135 Pixel. Zudem gibt es für dieses Board eine spezielle Micropython-Firmware, welche dieses Display gut unterstützt. Insbesondere bei Stand-Alone-Lösungen hat sich dieses Display als eine wertvolle Ergänzung erwiesen.

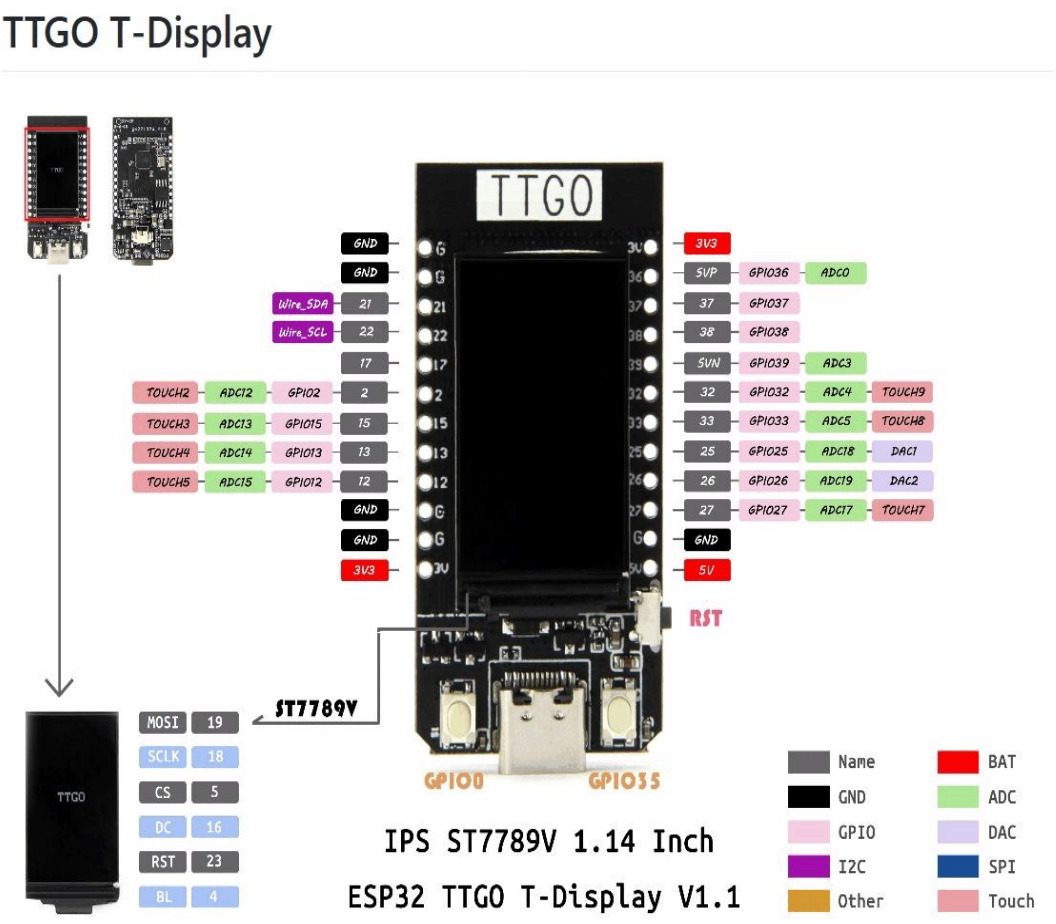


Abb. 1

Für die technischen Daten des Boards und des Displays verweise ich hier auf die Datei `ESP32-Materialien.zip`, siehe http://www.g-heinrichs.de/wordpress/index.php/informatik/TTGO/. Was für die Programmierung wichtig ist, werde ich jeweils im entsprechenden Kontext angeben.

Wenn Sie sich mit diesem Skript in die Python-Programmierung des ESP32 ***einarbeiten*** wollen, empfehle ich, sich genau diese Board-Variante anzuschaffen. Dann können Sie bei Bedarf auf die für dieses Board von mir bereitgestellten Programme zurückgreifen (`ESP-Materialien.zip`, s. o.). Natürlich lassen sich die behandelten Projekte auch mit anderen ESP32-Boards realisieren; allerdings kann es in dem einen oder anderen Fall erforderlich sein, Anpassungen vorzunehmen. Meist wird es darum gehen, andere Anschlüsse zu benutzen; bei dem Einsatz eines anderen Displays können die nötigen Änderungen aber auch schon gravierender sein.

Für die meisten Experimente ist ein Handy erforderlich. Nur bei wenigen Projekten werden Sie weitere Teile benötigen: einen BME/BMP280-Temperatursensor, 1 LED (rot oder grün) mit einem Vorwiderstand (zwischen 220 Ω und 470 Ω), ein Breadboard, etwa zehn Verbindungskabel (männlich-männlich). Unverzichtbar ist ein USB-Kabel zum Anschluss des Boards an Ihren Rechner. Auf der ESP32-Seite muss dieses Kabel einen Stecker vom Typ C besitzen.

Bitte beachten Sie: Häufig werden die ESP32-Boards nicht fertig montiert geliefert; es bleibt dem Käufer überlassen, die beiden Steckerleisten selbst an das ESP32-Board zu löten. Mit anderen Worten: Sie benötigen also noch einen Elektronik-Lötkolben und müssen damit umgehen können, oder sie kennen jemanden, der… ☺

Software

Zur Programmierung des ESP32 werden wir hier die **Thonny-IDE** benutzen. Sie wird auf der Webseite `www.Thonny.org` vorgestellt. Dort findet man direkt rechts neben der Überschrift Links zum kostenlosen Download (Windows, LINUX und Mac).

Es gibt eine Reihe von Installationsanleitungen zur Thonny-IDE im Internet. Die meisten sind in Englisch verfasst. Die Installation des Programms ist eigentlich recht einfach, aber bei der Konfiguration des Programms und auch bei der Installation des Treibers für die USB-Schnittstelle des ESP32-Boards kann der eine oder andere Hinweis gute Dienste leisten. Deswegen möchte ich hier eine möglichst detaillierte Anleitung geben. Diese bezieht sich auf die Windows-Version.

1. Schritt: Thonny installieren

Wir laden das aktuelle Installationsprogramm `thonny-3.x.x.exe` herunter und starten es. Es erscheint das Formular aus Abb. 2. Dort klicken wir auf die Schaltfläche ***Next>*** und starten damit die Installation. Wie üblich müssen wir die Lizenz-Vereinbarung akzeptieren, bevor wir fortfahren können (vgl. Abb. 3).

Abb. 2

Anschließend bestätigen wir den vorgeschlagen Installationsordner oder ändern ihn ab. Im nächsten Schritt aktivieren wir das Kontrollkästchen zur Er-

zeugung eines Thonny-Icons auf dem Desktop. Das Icon können wir bei Bedarf später entfernen oder in einen anderen Ordner verschieben. Wir klicken wieder auf *Next>* und bekommen die Mitteilung "Fertig zum Installieren", vgl. Abb. 6. Wenn wir wollen, können wir an dieser Stelle über die Schaltfläche *<Back* noch Korrekturen vornehmen. Die Installation beginnt, nachdem wir die Schaltfläche *Install* betätigt haben. Sie dauert eine Weile; ihr Ende wird mit dem Fenster aus Abb. 7 angezeigt. Hier betätigen wir die Schaltfläche *Finish*.

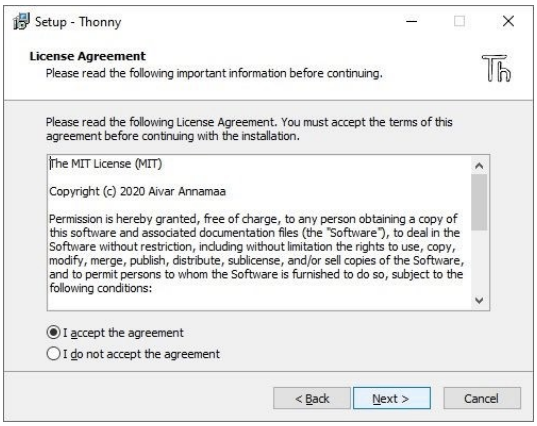


Abb. 3

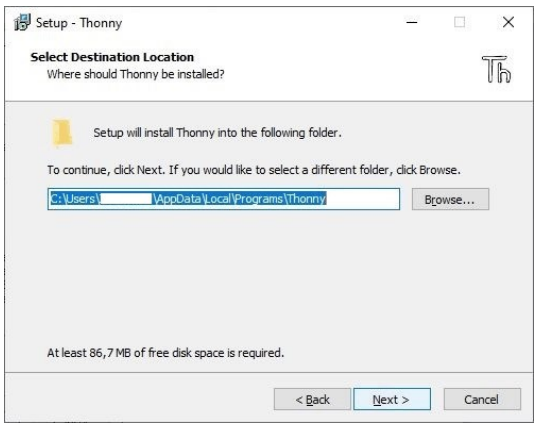


Abb. 4

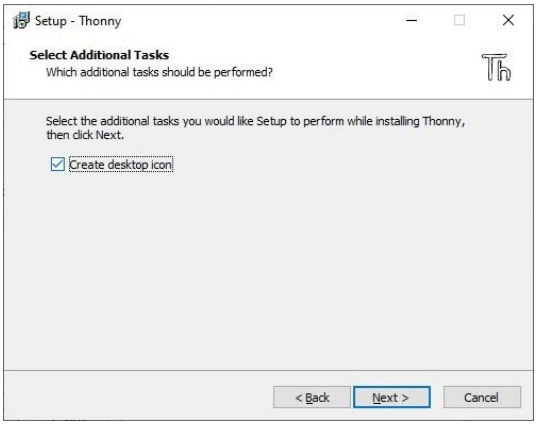


Abb. 5

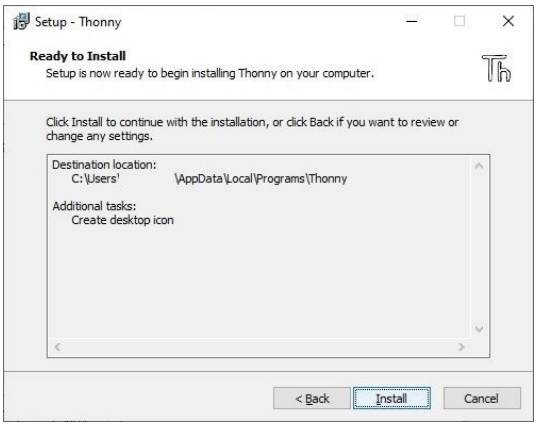


Abb. 6

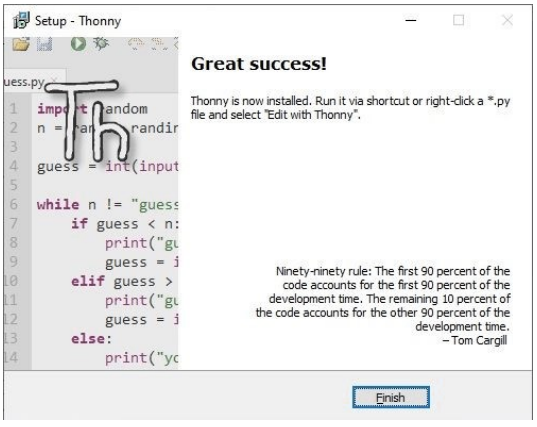


Abb. 7

2. Schritt: Thonny konfigurieren

Wir starten die Thonny-IDE. Zunächst erscheint das Fenster aus Abb. 8. Hier haben Sie die Möglichkeit, die Sprache einzustellen. Die Sprache Deutsch finden Sie in der Auswahlliste oben an erster Stelle. Sie werden aber sehen, dass in dieser ALPHA-Sprachversion die Übersetzung ins Deutsche leider noch nicht vollständig ist.

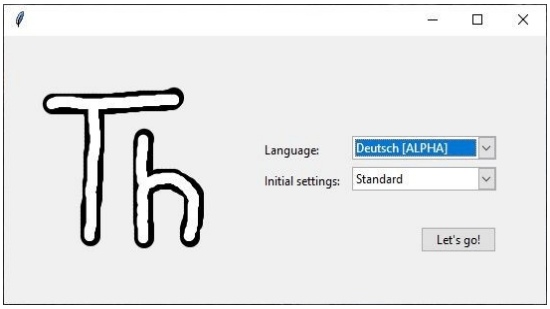


Abb. 8

Nach Anklicken der Schaltfläche ***Let's go*** öffnet sich das Fenster aus Abb. 9. Hier erkennen Sie zwei Fenster: Das obere Fenster (**Editor**) dient zur Eingabe und Bearbeitung von Python-Programmen. Das untere Fenster (**Shell** bzw. **Terminal**) hat mehrere Funktionen: So können hier z. B. Status-Meldungen vom Python-System angezeigt werden oder auch Ein- und Ausgaben eines Python-Programms erfolgen. In diesem Fall wird uns hier mitgeteilt, dass die Thonny-IDE hier auf die Python-Version 3.7.7 zurückgreift. Dies ist eine Version von Python für PCs, nicht für Microcontroller.

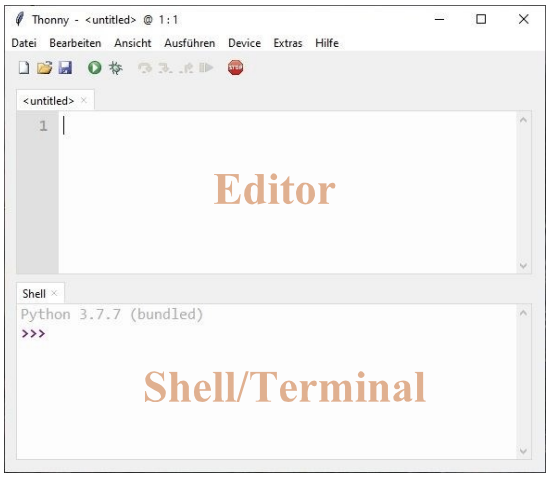


Abb. 9

Wir wollen hier mit Micropython arbeiten; wesentlich ist dabei, dass der Micropython-Interpreter nicht auf dem PC, sondern auf dem Microcontroller selbst liegen muss; dadurch können Python-Programme auch stand-alone auf dem ESP32 laufen.

Dazu müssen bei der Thonny-IDE einige Einstellungen vorgenommen werden; dies wollen wir jetzt schrittweise durchführen.

A: Treiber für das ESP32-Board installieren

Damit Thonny mit dem ESP32-Board kommunizieren kann, muss auf dem PC ein passender Treiber installiert sein. Um dies zu testen, schließen wird das Board an den PC an. Wir öffnen den Gerätemanager und kontrollieren, ob sich in der Rubrik ***Anschlüsse COM & LPT*** ein Gerät mit der Bezeichnung ***CP210x*** befindet. Ist das so, können Sie die folgenden Anweisungen überspringen und direkt mit Schritt B weiter machen.

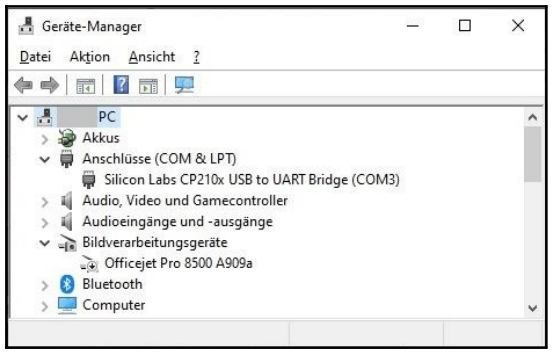


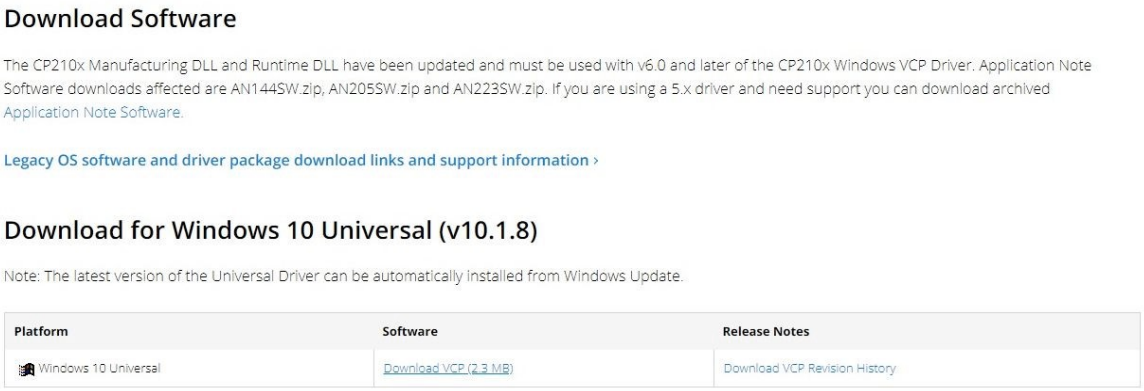
Abb. 10

Entfernen Sie zunächst wieder die Verbindung zwischen PC und ESP32-Modul.

Laden Sie nun den passenden Treiber herunter. Sie finden ihn unter:

https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

oder in dem bereits erwähnten Materialien-Ordner.



Download Software

The CP210x Manufacturing DLL and Runtime DLL have been updated and must be used with v6.0 and later of the CP210x Windows VCP Driver. Application Note Software downloads affected are AN144SW.zip, AN205SW.zip and AN223SW.zip. If you are using a 5.x driver and need support you can download archived Application Note Software.

Legacy OS software and driver package download links and support information ›

Download for Windows 10 Universal (v10.1.8)

Note: The latest version of the Universal Driver can be automatically installed from Windows Update.

Platform	Software	Release Notes
Windows 10 Universal	Download VCP (2.3 MB)	Download VCP Revision History

Abb. 11

Mit Hilfe der rechten Maustaste laden Sie die Software in einen temporären Ordner. Hier entpacken Sie die herunter geladene zip-Datei.

Starten Sie nun das Treiber-Installations-Programm; wählen Sie die exe-Datei aus, die zu Ihrem Betriebssystem passt.

Nach dieser Installation können Sie das ESP32-Board wieder an Ihren Rechner anschließen. Nun sollte das CP210x-Gerät im Gerätemanager (Abb. 10) auftauchen.

B: Download der Micropython-Firmware

Laden Sie über meine Webseite http://www.g-heinrichs.de/wordpress/index.php/informatik/TTGO/ den Ordner `ESP32-Materialien.zip herunter` und entpacken Sie ihn in ein Verzeichnis ihrer Wahl. Diesen Ordner benötigen wir für den nächsten Schritt.

C: Thonny-IDE einrichten

Wählen Sie im Menüpunkt **Extras** den Punkt **Optionen**. Es erscheint das Formular “Thonny options”; hier wählen Sie die Lasche “Interpreter”, vgl. Abb. 12. Wählen Sie zunächst als Interpreter/Gerät aus: MicroPython (ESP32), so wie es in der Abbildung zu sehen ist.

Klicken Sie nun auf die Schaltfläche ***Open the dialog for installing...*** Es erscheint jetzt ein neues Fenster (vgl. unterer Bereich von Abb. 12). Wählen Sie hier Ihren Anschluss (Port) aus und fügen Sie die den Dateinamen (inkl. Pfad) der Firmware aus Schritt 2 ein; am einfachsten geht das über die Schaltfläche ***Browse...***. **Ganz wichtig: Aktivieren Sie unbedingt das Kontrollkästchen zum Löschen des Flash-Speichers!**

Benutzen Sie als Firmware **unbedingt** die Datei mit der Bezeichnung

`firmware_micropython_russ_hughes_v_1_14a.bin`,

und nicht die in der Abbildung 12 angezeigte ältere Firmware-Version. Nur mit dieser Firmware wurden die im Script vorgestellten Programme getestet. Da die Firmware laufend weiterentwickelt wird, ist nicht garantiert, dass die Programme auch mit einer älteren Firmware-Version korrekt arbeiten.

Wenn Sie nun die Schaltfläche ***Install*** anklicken, wird die Firmware auf den ESP32 übertragen; dies nimmt etwas Zeit in Anspruch. Nach Abschluss der Firmware-Installation können wir das Optionen-Fenster schließen.

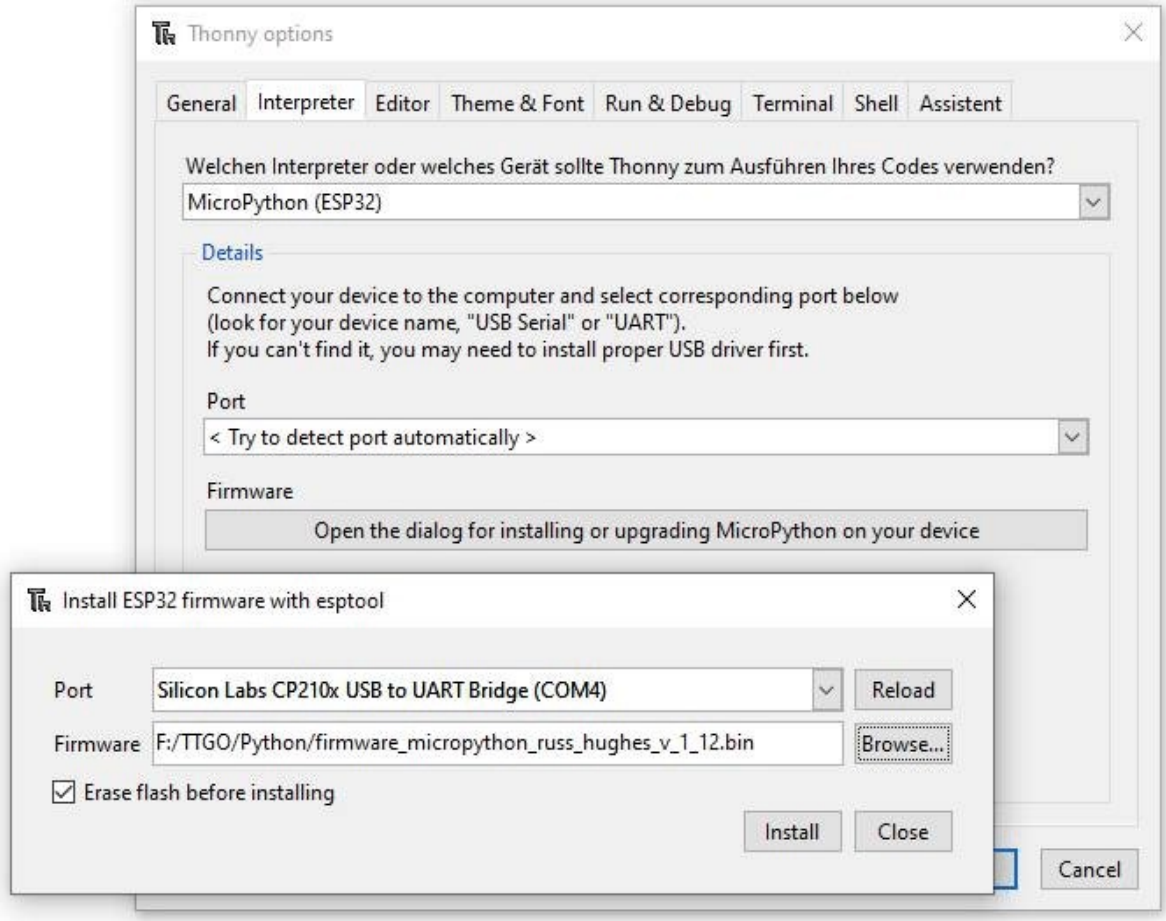


Abb. 12

D: Endkontrolle

Wir lassen das ESP32-Modul angeschlossen und beenden die Thonny-IDE. Anschließend öffnen wir die Thonny-IDE wieder; jetzt zeigt die Shell eine **MicroPython-Version v1.14** für den ESP32 an.

Abb. 13: Meldung von Micropython für eine ältere Version

Folgen Sie einmal dem Vorschlag in der zweiten Zeile und tippen Sie hinter das **Python-Prompt-Zeichen >>>** den Befehl

```
help()
```

ein; schließen Sie die Eingabe mit der Enter-Taste ab. Alsbald erscheinen im Terminal ein paar Informationen zum Umgang mit Micropython - ein weiteres Zeichen dafür, dass die Thonny-IDE und die Firmware auf dem ESP32 korrekt arbeiten. Damit ist die Thonny-Konfiguration abgeschlossen.

Sollten noch irgendwelche Probleme bestehen, finden Sie vielleicht hier eine Lösung:

https://randomnerdtutorials.com/getting-started-thonny-micropython-python-ide-esp32-esp8266/

E Einführung

E.1 Erste Erfahrungen mit Zahlen und Zeichenketten

Wir schließen unser ESP32-Board an den Rechner an und starten die Thonny-IDE. In dem Terminal-Fenster (s. Kapitel V, Abb. 9) wird die Micropython-Version angegeben; darunter erscheint das Python-Prompt: >>>

Hinter das Prompt-Zeichen geben wir nun ein:

```
23+8
```

Dann betätigen wir die Enter-Taste. Daraufhin sieht das Terminalfenster so aus:

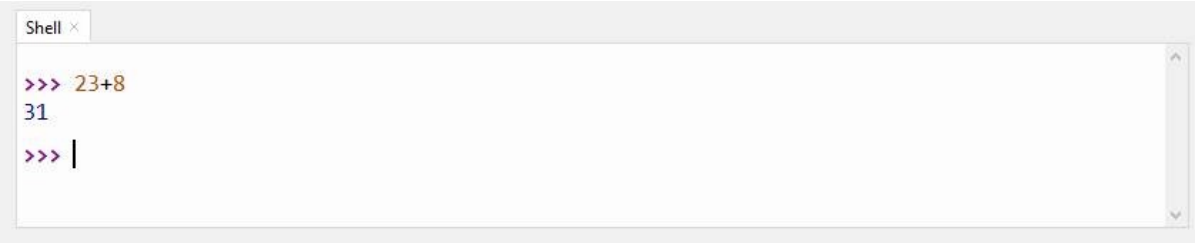


Abb. 1

Unter der Rechenaufgabe steht das Ergebnis, darunter eine Leerzeile, und darunter ist wieder das Prompt-Zeichen zu sehen: Micropython wartet auf eine neue Eingabe.

Ein Blick hinter die Kulissen: Tatsächlich war es nicht das Terminal, das hier gerechnet hat. Wie schon im Kapitel V dargelegt, hat das Terminal die eingegebene Aufgabe an das Micropython-System auf dem Mikrocontroller weitergeleitet. Und dieses Micropython-System hat die Aufgabe bearbeitet und das Ergebnis an das Terminal gesendet. Anschließend wird auf eine neue Eingabe gewartet... Diese Schleife wird auch als Read-Evaluate-Print-Loop (kurz: **REPL**) bezeichnet.

Betrachten wir eine weitere Aufgabe:

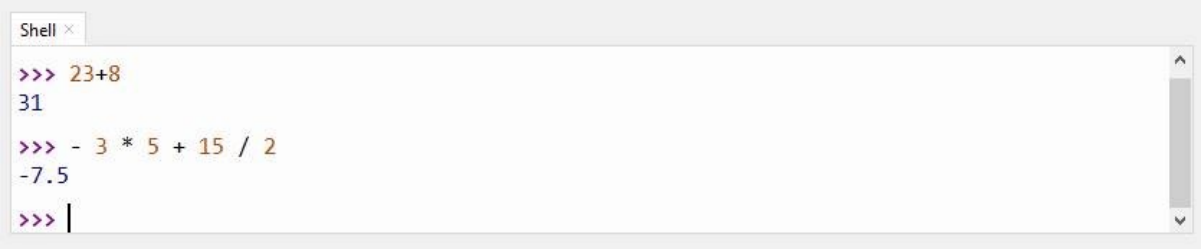


Abb. 2

Wir schließen:

- In den Term können wir Leerzeichen einfügen. Micropython ignoriert sie.
- Micropython kann auch mit negativen Zahlen rechnen.
- Micropython beherrscht die Regel "Punktrechnung vor Strichrechnung".
- Micropython kann auch mit Dezimalzahlen umgehen.

Prüfen Sie nun selbst einmal nach, ob Micropython auch mit (runden) Klammern umgehen kann.

Zahlen können in **Variablen** gespeichert werden; dazu schreibt man hinter den Variablennamen ein Gleichheitszeichen, gefolgt von dem zu speichernden Wert. Man sagt auch: Mit `a = 22` wird **der Variablen a der Wert 22 zugewiesen**. Variablennamen können aus mehreren Zeichen bestehen. Allerdings sind nur folgende Zeichen zulässig: Groß- und Kleinbuchstaben (ohne deutsche Sonderzeichen wie ä, Ä, ..., ß), Ziffern und der Unterstrich (_). Ein Variablenname darf nicht mit einer Ziffer beginnen. Testen Sie selbst einige Zeichenfolgen aus.

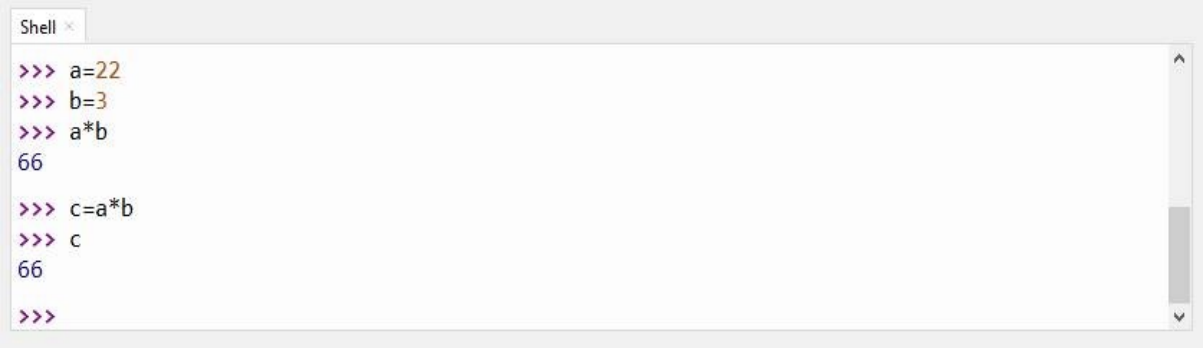


Abb. 3

Geben wir hinter dem Prompt eine Variable ein, wird nach dem Betätigen der Enter-Taste der Wert der Variable ausgegeben. Wie reagiert Micropython, wenn Sie auf diese Weise versuchen, den Wert einer Variablen auszugeben, der wir zuvor noch keinen Wert zugewiesen haben? Versuchen Sie es selbst aus!

In der Zeile `c = a*b` wird der Variablen `c` nicht die Zeichenfolge "`a*b`" zugewiesen. Vielmehr wird zunächst der Ausdruck `a*b` aus den zugewiesenen Werten berechnet und anschließend das Ergebnis dieser Rechnung in der Variablen `c` gespeichert.

Können auch Dezimalzahlen auf die gleiche Weise in Variablen gespeichert werden? Was geschieht mit den Variablen, wenn die Stopp-Schaltfläche (unterhalb der Menü-Zeile) oder `Ausführen > Soft-Reboot` angeklickt wird? Testen Sie es selbst aus!

Wenn Sie in Abb. 3 `c = A * b` statt `c = a * b` geschrieben hätten, wäre es zu einer Fehlermeldung gekommen. Micropython unterscheidet nämlich zwischen Groß- und Klein-Buchstaben. Der Fachman sagt: **Micropython ist case-sensitive**. Dies gilt nicht nur für Variablennamen, sondern generell!

Auch **Zeichenketten** (englisch: **Strings**) können in Variablen gespeichert werden. Damit eine Zeichenkette von Micropython nicht als Variablenname gedeutet wird, muss sie unbedingt mit Anführungszeichen markiert werden. Dabei ist es gleichgültig, ob Sie dafür ‘ oder “ benutzen. Allerdings muss am Ende der Zeichenfolge dasselbe Anführungszeichen stehen wie am Anfang. Testen Sie selbst aus:

```
>>> s1 = 'Hallo'
>>> s2 = "Welt"
>>> s1
'Hallo'
>>> s1+s2
'HalloWelt'
```

Überlegen Sie einmal: Wie ist hier das Plus-Zeichen zwischen `s1` und `s2` zu deuten? Wie kann man dafür sorgen, dass zwischen den Wörtern Hallo und Welt in der letzten Zeile ein Leerzeichen steht?

Die Bedeutung des Plus-Zeichens hängt davon ab, von welchem Typ der Inhalt der Variablen ist, zwischen denen es steht. Bei vielen Programmiersprachen muss der Programmierer vor der Benutzung einer Variablen eine so genannte **Variablendeklaration** vornehmen, bei der der Typ der Variablen festgelegt wird. Eine solche Deklaration ist bei Micropython nicht erforderlich. Am Beispiel des Pluszeichens haben wir gesehen, dass Micropython hier selbstständig den Typ erkennt.

E.2 Mit Objekten arbeiten: LEDs ein- und ausschalten

An unser ESP32-Board schließen wir eine LED an Pin 25 wie in Abb. 4 dargestellt an. Es ist übrigens nicht gleichgültig, an welchen Pin sie angeschlossen wird, weil nicht alle Pins als Ausgänge benutzt werden können. Achten Sie auch bitte auf die Polung der LED!

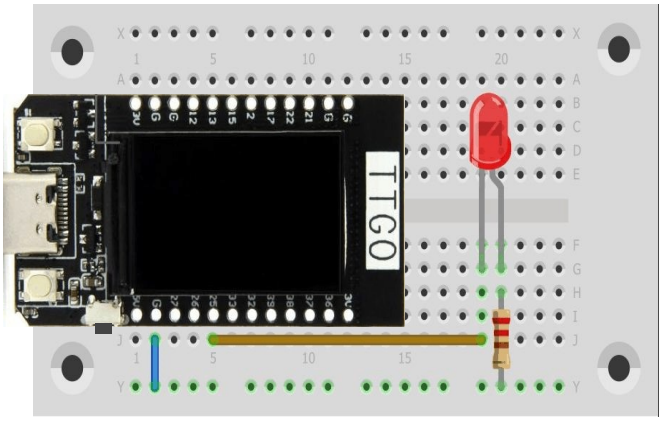


Abb. 4

Über das Terminal soll die LED nun ein- und ausgeschaltet werden. Im Internet finden wir die folgende Zeile zum Einschalten:

```
led.value(1)
```

Sie kommt uns etwas suspekt vor, weil nirgendwo eine Pin-Nummer auftaucht. Wir geben trotzdem den Befehl einmal im Terminal ein. Das Ergebnis ist in Abb. 5 zu sehen. Es macht klar: Die Angelegenheit ist wohl nicht ganz so einfach. Tatsächlich werden wir hier etwas weiter ausholen müssen; dabei werden wir mit wichtigen Konzepten bekannt gemacht: Module, Klassen, Objekte mit ihren Methoden und Eigenschaften. Das klingt erst einmal recht abstrakt. An unserem LED-Beispiel lässt sich aber gut verdeutlichen, wie man damit umgeht. Das wollen wir jetzt zeigen.

Abb. 5

Ganz wichtig ist zunächst: Standardmäßig hält Micropython nur eine relativ geringe Anzahl von Befehlen bereit. Ähnliche Situationen gibt es im alltäglichen Leben: In Ihrem Haus wird sich nicht zu jedem Thema ein passendes Buch finden. Was machen Sie? Sie besorgen sich aus einer Bücherei oder dem Internet ein oder mehrere "Dokumente" (Bücher, Zeitschriften, Webseiten…) mit den gewünschten Themen. Für alle Eventualitäten alle erdenklichen Dokumente immer zu Hause vorzuhalten wäre einfach unpraktisch.

Ähnlich verfahren wir bei Python: Es gibt sehr wohl Befehle, mit denen man LEDs kontrollieren kann. Man muss sie sich allerdings besorgen. Micropython besitzt eine Sammlung von Modulen; solch eine Sammlung wird auch eine Bibliothek (englisch: **library**) genannt. In der Firmware unseres ESP32 befindet sich bereits eine solche Bibliothek; je nach benutzter Firmware können die einzelnen Module sich in Umfang und Funktionsweise geringfügig unterscheiden.

Jedes **Modul** liefert uns nun Hilfsmittel zu einem bestimmten Thema. Das Modul, welches uns bei dem LED-Problem helfen kann, hat den Namen `machine`. Dieses Modul müssen wir uns aus der Bibliothek holen (**importieren**). Dies geschieht mit dem Kommando

```
import machine
```

Um uns einen Überblick über den Inhalt dieses Moduls zu verschaffen, lassen wir und das Inhaltsverzeichnis (**directory**) anzeigen:

```
dir(machine)
```

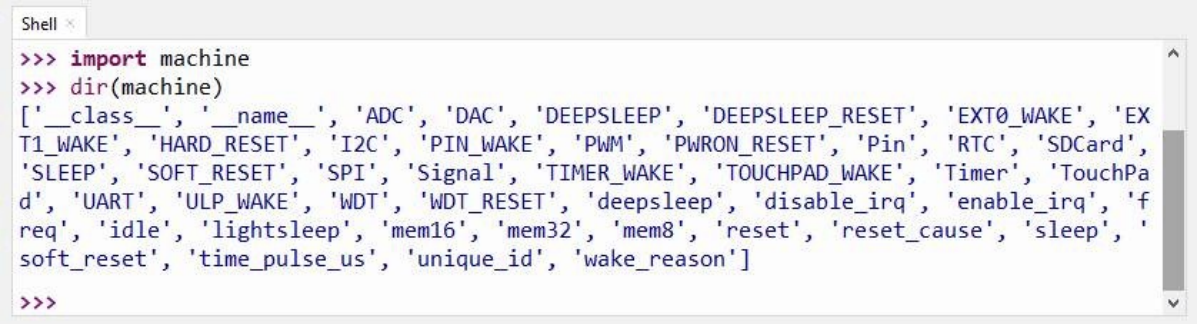


Abb. 6

Einige "Inhalte" wie ADC, I2C, PWM und UART kommen uns vielleicht bekannt vor. Am ehesten für unsere Zwecke geeignet scheint aber `Pin` zu sein; immerhin wollen wir ja unsere LED über einen Pin des Mikrocontrollers ansteuern. Was hat es aber damit auf sich?

Kann uns eine Internet-Recherche weiterhelfen? Mit den Stichworten "Micropython" und "pin" erhalten wir als ersten Link:

`https://docs.micropython.org/en/latest/library/machine.Pin.html`

Dort erfahren wir, dass es sich hier um eine Klasse handelt, die zur Kontrolle der I/O-Pins dient. Eine **Klasse** kann man sich als einen Bauplan vorstellen. Nach diesem Bauplan können **Objekte** (oft auch als **Instanzen** der Klasse bezeichnet) erzeugt werden. Beispielsweise wird durch die Befehlszeile

```
r1 = Rechteck(laenge, breite, ... )
```

eine Instanz der Klasse `Rechteck` mit den entsprechenden Werten für die Eigenschaften `laenge` bzw. `breite` erzeugt (vorausgesetzt, dass diese Klasse bereits existiert) und weist sie der Variablen `r1` zu.

Da in unserem Fall das Pin-Objekt zur Steuerung unserer LED dienen soll, wählen wir als Variablennamen `led`.

```
led = machine.Pin(25, machine.Pin.OUT)
```

Der erste Parameter legt hier die Nummer des Pins (GPIO-Nr) fest, der zweite gibt an, dass dieser Pin als Ausgang konfiguriert wird (`OUT` ist eine Eigenschaft der Klasse `Pin`). Damit Micropython "weiß", in welchem Modul die Klasse `Pin` zu finden ist, wird dem Klassennamen der Modulname vorangestellt, getrennt durch einen Punkt.

Die für uns wichtige **Methode** heißt `value`: Mit `led.value(1)` wird der Ausgang von Pin25 auf High gelegt; die LED geht dann an. Ausgeschaltet wird mit `led.value(0)`.

Das sollten Sie jetzt unbedingt austesten!

Wer möchte, importiert nicht das komplette `machine`-Modul, sondern nur die Klasse `Pin` aus diesem Modul. Das erreicht man mit der Anweisung:

```
from machine import Pin
```

Das hat u. A. den Vorteil, dass die Anweisung zum Erzeugen unseres Objekts `led` jetzt kürzer ausfällt:

```
led = Pin(25, Pin.OUT)
```

E.3 Programmieren

Für diesen Abschnitt stecken wir uns das folgende Ziel: Unsere LED aus dem letzten Abschnitt soll blinken. Und das ohne, dass wir selbst am Terminal dafür aktiv eingreifen. Dazu muss die LED selbstständig vom Mikrocontroller ein- und ausgeschaltet werden. Genauer gesagt muss der Mikrocontroller folgende "Befehle" der Reihe nach durchführen.

```
einschalten
warten
ausschalten
warten
einschalten
warten
ausschalten
warten
u.s.w
```

Eine derartige Abfolge von Befehlen bezeichnet man als **Programm**.

Micropython-Programme werden im Editor (vgl. Abb. 9 aus Kapitel V) eingegeben. Die benötigten Micropython-Befehle kennen wir zum großen Teil schon; was uns noch fehlt, ist ein Befehl, der den Mikrocontroller warten lässt. Dies geht mit der `sleep`-Funktion aus dem Modul `time`. Mit

```
sleep(1.5)
```

lassen wir den Mikrocontroller z. B. 1,5 s lang warten. Ohne solche Warte-Befehle würde so rasch ein- und ausgeschaltet, dass das Auge dies nicht mehr wahrnehmen kann.

Das Programm sieht nun so aus wie in Abb. 7. Wir öffnen ein neues Editor-Fenster und tippen es dort ein; dabei wird der Programmtext automatisch formatiert: Schlüsselwörter wie `from` und `import` werden z. B. fett angezeigt.

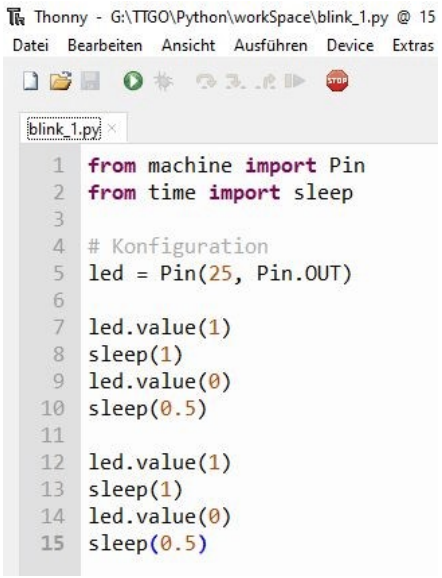


Abb. 7

Mit `Datei > Save as...` wollen wir es unter dem Namen `blink_1.py` abspeichern. Als Speicherort werden uns zwei Optionen angeboten: `This computer` oder `Micropython device`. Wir wählen die erste Möglichkeit und speichern dann das Python-Programm in einem Ordner unserer Wahl ab. Die Thonny-IDE fügt dabei automatisch die Extension `.py` an den Dateinamen an. Außerdem wird der benutzte Dateiname beim Speichern in den Reiter des Editierfeldes eingetragen.

Das Programm können wir mit `Ausführen > Run current script` starten. Einfacher geht das allerdings mit der Funktionstaste F5 oder mit der Schaltfläche  . Die LED blinkt nun zweimal auf!

> *Ein Blick hinter die Kulissen:* Bei den meisten Programmierumgebungen wird das in einer Hochsprache wie z. B. Basic, C oder Pascal geschriebene Programm zunächst **kompiliert**; das dabei erzeugte **Maschinenprogramm** wird anschließend auf den Mikrocontroller geladen und kann dann dort gestartet werden. Bei unsere Thonny-Micropython-Umgebung läuft es hingegen so ab: Das in Micropython geschriebene Programm wird durch die REPL (s. o.) an das Micropython-System auf dem ESP32 übertragen und dort **interpretiert**; d. h. von dem Micropython-Quelltext wird ein erster Abschnitt in Bytecode übersetzt und gleich ausgeführt, anschließend wird auf dieselbe Weise mit dem nächsten Abschnitt verfahren u.s.w. Daran würde sich auch nichts wesentlich ändern, wenn wir den Micropython-Quelltext auf dem ESP32 gespeichert hätten.

Jetzt wollen Sie sicherlich die LED häufiger blinken lassen. Natürlich kann man die vier Zeilen für das Ein- und Ausschalten (inkl. Pausen) noch einige Male unten im Programm anfügen. Es geht aber auch eleganter mit einer **Endlos-Schleife**:

```
while True:
      led.value(1)
      sleep(1)
      led.value(0)
      sleep(0.5)
```

Beachten Sie: Sobald Sie hinter dem Doppelpunkt die Enter-Taste gedrückt haben, springt der Cursor in die nächste Zeile und rückt dabei ein. Alle vier Zeilen des Schleifen-Körpers müssen auf die gleiche Weise **eingerückt** werden. Später werden wir auf die Bedeutung dieses Einrückens (engl.: **indenting**) noch ausführlich eingehen.

Speichern Sie das neue Programm unter dem Namen `blink_2.py` ab und starten Sie es. Es lässt die LED solange blinken, bis Sie das Programm unterbrechen, z. B. indem Sie die Stopp-Schaltfläche betätigen.

Nun wollen wir, dass der ESP32 das Blink-Programm auch ohne Rechner ausführt. Dafür speichern wir das Programm auf dem ESP32 unter dem Dateinamen `main.py` ab. Der Grund dafür ist folgender: Nach einem Hard-Reset (z. B. RST-Taste betätigt oder den ESP32 gerade an elektrische Quelle angeschlossen) wird automatisch die Datei `boot.py` und anschließend die Datei `main.py` auf dem ESP32 ausgeführt.

Nachdem wir unser Blink-Programm unter dem Namen `main.py` auf dem ESP32 gespeichert haben, lassen wir uns mit `Ansicht > Dateien` die Dateien auf dem ESP32 und auch auf dem Rechner in getrennten Fenstern anzeigen (vgl. Abb. 8). Diese beiden Fenster besitzen zwar keine Drag&Drop-Funktionen; über die rechte Maustaste lässt sich aber jeweils ein Kontext-Menü öffnen. Mit diesem kann man z. B. auch Dateien vom Rechner auf den ESP32 übertragen.

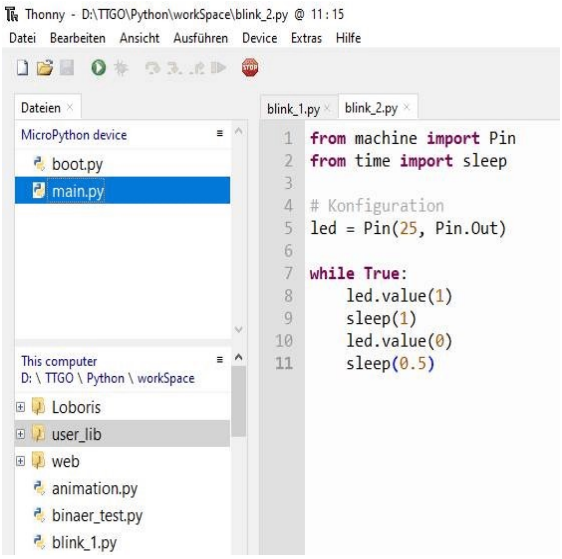


Abb. 8

Nun können wir einmal den Reset-Knopf an dem Board betätigen. Im Terminal erkennen wir einige Aktionen, die dabei ausgeführt werden. Anschließend beginnt die LED zu blinken, ohne dass wir das Programm explizit starten mussten.

Wer eine Powerbank zur Verfügung hat, trennt einmal das ESP32-Board vom Rechner und schließt es stattdessen an die Powerbank an. In dem Augenblick, in dem das Board wieder mit Strom versorgt wird, werden automatisch auch `boot.py` und `main.py` ausgeführt und unsere LED blinkt - diesmal auch ohne Kontakt mit dem Rechner.

E.4 Vergleichen und Verzweigen

Wir bleiben bei unserer LED: Diesmal soll sie nicht selbstständig blinken, sondern mit einem der Taster unseres Boards geschaltet werden. Wir wählen den linken Taster in der Abb. 1 aus dem Kapitel V. Ganz bewusst benutzen wir den Ausdruck ***Taster*** und nicht *Schalter*; denn wie bei einer Türklingel soll die LED nicht dauerhaft ein- oder ausgeschaltet werden; vielmehr soll die LED dann und nur dann leuchten, wenn der Taster gedrückt ist.

In Abb. 9 sehen wir, wie dieser Taster mit dem Pin `GPIO0` verbunden ist. Wegen des Pullup-Widerstands liegt am Punkt X ein hohes Potential (high) an, wenn der Taster nicht gedrückt ist. Solange er gedrückt ist, liegt dort ein niedriges Potential (low) an. Ob hohes oder niedriges Potential vorliegt, kann über den Pin `GPIO0` ermittelt werden. Dazu muss dieser Pin jetzt als Eingang konfiguriert werden.

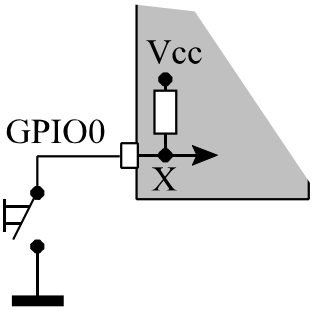


Abb. 9

Beim Erzeugen unseres Pin-Objekts muss der zweite Parameter deswegen jetzt nicht `Pin.OUT`, sondern `Pin.IN` lauten. Außerdem muss der interne Pullup-Widerstand an diesem Pin "aktiviert" werden; dies geschieht mit einem dritten Parameter `Pin.PULL_UP`. Das erzeugte Objekt nennen wir jetzt sinnvollerweise `taster`.

```
taster = Pin(0, Pin.IN, Pin.PULL_UP)
```

Ob der Zustand an dem Pin nun high oder low ist, kann mit der Methode `value` ermittelt werden:

```
zustand = taster.value()
```

Entscheidend ist hier, dass hier die Methode `value` kein Argument besitzt. In diesem Fall wird an dem Pin der Zustand High oder Low nicht gesetzt, sondern er wird gemessen und als Rückgabewert in der Variable zustand gespeichert. Liegt am Pin der Zustand High vor, ist dieser Rückgabewert 1, ansonsten 0.

Unser Programm muss nun immer wieder den Zustand am Taster bestimmen. Wenn dann `zustand` gleich 0 ist, dann muss die LED eingeschaltet werden, ansonsten muss sie ausgeschaltet werden. Das Micropython-Programm sieht dann so aus:

```
from machine import Pin

led = Pin(25, Pin.OUT)
taster = Pin(0, Pin.IN, Pin.PULL_UP)
while True:
      zustand = taster.value()
      if zustand == 0:
            led.value(1)
      else:
            led.value(0)
```

Die Grundstruktur der **Verzweigung** ist

```
if <Bedingung>:
      Block für den Wenn-Teil
else:
      Block für den Sonst-Teil
```

In vielen Programmiersprachen werden Anfang und Ende eines Blockes durch spezielle Klammern oder Schlüsselwörter wie `begin` und `end` markiert. In Micropython werden **Blöcke** lediglich durch **Einrücken** gekennzeichnet: Alle Befehle **desselben** Blockes werden **auf die dieselbe Weise eingerückt**. In unserem Fall befindet sich der Block für den Wenn-Teil innerhalb des Blockes für die Endlos-Schleife. Der untergeordnete Block (hier z. B. Wenn-Teil) muss dann weiter nach rechts eingerückt sein. Wie wir bereits bei der Endlos-Schleife im letzten Abschnitt festgestellt haben, unterstützt uns die Thonny-IDE beim Einrücken.

Die Bedingung, nach der das Programm seine Entscheidung fällen soll, besteht meist aus einem Vergleich, z. B.

```
if wert > 8:
if x == y:
```

Bitte beachten Sie: Beim **Überprüfen auf Gleichheit** benutzt man ein **doppeltes Gleichheitszeichen**; das einfache Gleichheitszeichen ist für die Wertzuweisung reserviert! Weitere Vergleichsoperatoren sind >=, <, <= und `!=` (ungleich).

Allgemein kann als Bedingung ein Wahrheitswert oder ein entsprechender Ausdruck angegeben werden. Solche Ausdrücke können z. B. auch Verknüpfungen von Wahrheitswerten sein:

```
if a>5 and b:
```

Micropython wertet dabei zunächst die einzelnen Ausdrücke aus und weist ihnen Wahrheitswerte zu, True oder False. Diese Wahrheitswerte werden dann zu einem einzigen Wahrheitswert verknüpft, z. B. `True and False` → `False`. (Für das Ergebnis bewertet Micropython dabei nicht unbedingt alle einzelnen Vergleiche, sondern verwendet eine so genannte "Kurzschluss-Auswertung".)

Ein Blick hinter die Kulissen: Zwar müssen wir bei Micropython Variablen nicht deklarieren, das bedeutet aber nicht, dass es bei Micropython keine unterschiedlichen Datentypen gibt. Bislang haben wir folgende (primitive) **Datentypen** kennen gelernt:

Variablentyp	Bedeutung
int	ganze Zahlen (integer)
float	Fließkommazahlen
string	Zeichenketten
bool	Wahrheitswerte (Boolesche Werte): True, False

An dieser Stelle sollte auch klar werden, wie unsere Endlos-Schleife funktioniert. Sie stellt einen Spezialfall dar von:

```
while <Bedingung>:
```
`Block` (welcher solange wiederholt wird, wie die Bedingung den Wert True hat)

Wenn nun die Bedingung nur aus dem Wert True besteht, wird der Block endlos wiederholt.

Überlegen Sie nun einmal, was die beiden folgenden Programme machen. Testen Sie es ggf. mit Ihrem ESP32-Board aus.

```
# Programm 1
from machine import Pin
from time import sleep

led = Pin(25, Pin.OUT)
taster = Pin(0, Pin.IN, Pin.PULL_UP)

while taster.value() == 1:
      led.value(1)
      sleep(0.1)
      led.value(0)
      sleep(0.1)


# Programm 2
from machine import Pin

led = Pin(25, Pin.OUT)
taster = Pin(0, Pin.IN, Pin.PULL_UP)
while True:
      led.value(1-taster.value())
```

E.5 Listen, Tupel, Strings und Iteratoren

Im nächsten Kapitel wollen wir das ESP32-Board nach Wlans scannen lassen. Ohne hier schon auf eine inhaltliche Analyse einzugehen, schauen wir uns das Ergebnis eines solchen Scans an (s. Abb. 10). Uns geht es in diesem Abschnitt darum, die einzelnen dort auftauchende Strukturen kennen zu lernen.

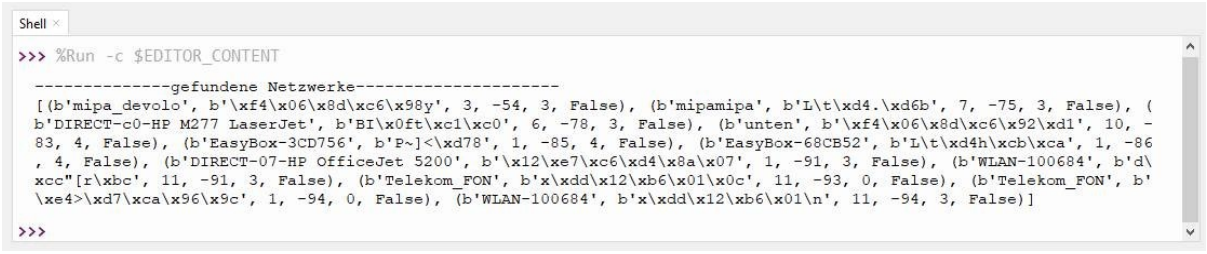


Abb. 10

Beginnen wir mit den so genannten Listen. Durch

```
quadratzahlen = [4, 9, 16, 25, 36, 49]
```

wird in der Variablen `quadratzahlen` eine Liste von 6 Quadratzahlen abgespeichert. **Listen**

bestehen aus einer (geordneten) Reihe von Daten; diese werden auch **Elemente** genannt. Die Elemente einer Liste stehen in **eckigen Klammern** und werden jeweils durch ein Komma getrennt; die Elemente einer Liste müssen nicht unbedingt denselben Datentyp besitzen.

Entscheidend ist, dass wir mit Hilfe eines Index auf die einzelnen Elemente zugreifen können; der Index nummeriert die einzelnen Elemente der Reihe nach durch; dabei beginnt die Zählung bei 0. Bei unserer Liste `quadratzahlen` erfolgt der Zugriff auf das Element mit dem Index 4 z. B. durch

```
quadratzahlen[4]
```

Das Ergebnis ist in diesem Fall 36. Nicht nur auf einzelnen Elemente der Liste kann man zugreifen, sondern auch auf **Teillisten**: Die Eingabe

```
quadratzahlen[2:5]
```

liefert z. B. die folgende Teilliste:

```
[16, 25, 36]
```

Beachten Sie: Es werden hier Elemente aus der Liste `quadratzahlen` "herausgeschnitten", und zwar vom Index 2 (**einschließlich**) bis zum Index 5 (**ausschließlich**).

Mit einer **Schleife** kann man die Elemente einer Liste zeilenweise ausgeben (oder auch anderweitig bearbeiten):

```
i = 0
while i<6:
    print(quadratzahlen[i])
    i += 1
```

Die letzte Zeile ist eine abkürzende Schreibweise für die Anweisung `i=i+1`; der Index `i` wird hier also um 1 erhöht. Mit den folgenden zwei Zeilen lässt sich die zeilenweise Ausgabe allerdings deutlich einfacher gestalten:

```
for quadratzahl in quadratzahlen:
    print(quadratzahl)
```

Ein wichtiger Vorteil hierbei ist auch, dass die Anzahl der Schleifendurchläufe nicht explizit bestimmt werden muss.

Wegen des Schlüsselwortes `for` spricht man hier auch von einer `for`-Schleife. Anders als bei manchen anderen Programmiersprachen taucht hier ein Index nicht (explizit) auf, vielmehr wird direkt mit den Elementen gearbeitet, die hier durch die **Schleifenvariable** `quadratzahl`

repräsentiert werden. Hinter dem Schlüsselwort `in` müssen sogenannte **iterierbare** Objekte stehen; dazu gehören z. B. Listen und Strings.

> *Ein Blick hinter die Kulissen:* Durch das Schlüsselwort `for` wird zu unserer Liste `quadratzahlen` ein neues Objekt erzeugt, **Iterator** genannt. Dieser Iterator besitzt einen Zeiger, über den auf die einzelnen Elemente unserer Liste zugegriffen werden kann. Bei jedem Durchlauf der `for`-Schleife wird dieser Zeiger automatisch auf das nächste Element der Liste gerichtet; dieses Element wird sodann der Schleifenvariablen `quadratzahl` zugewiesen. Das geschieht so lange, bis das Ende der Liste erreicht ist; dann wird die Schleife automatisch abgebrochen.

Ein ähnlicher Datentyp wie die Liste ist das **Tupel**: Äußerlich unterscheiden sich Tupel von Listen dadurch, dass sie nicht mit eckigen, sondern mit **runden Klammern** gekennzeichnet werden. Im Gegensatz zu den Listen sind die Elemente eines Tupels **nicht veränderbar**. Ansonsten lassen sie sich aber wie Listen behandeln. Insbesondere kann auch bei ihnen ein Iterator zur Anwendung kommen.

Dasselbe wie für Tupel gilt auch für **Zeichenketten (Strings)**: Versuchen Sie einmal selbst von der Zeichenkette 'Hallo Welt!' einen Teil auszuschneiden oder die einzelnen Zeichen zeilenweise auszugeben.

Werfen wir noch einmal einen Blick auf Abb. 10, so entdecken wir Objekte, die Zeichenketten sehr ähnlich sind; allerdings steht vor dem ersten Anführungszeichen jeweils ein `b`. Beispiele sind `b'mipa_devolo'` und `b'\xf4\x06\x8d\xc6\x98y'`. Hier handelt es sich um Objekte vom Datentyp **`bytes`**; manche bezeichnen sie auch als **Byte-Strings**. Ein solches Objekt besteht aus einer **Folge von Bytes**. Es besteht **nicht aus einer Folge von Zeichen**, auch wenn einzelne Bytes auf dem Terminal manchmal in Form von Zeichen dargestellt werden. Dies ist genau dann der Fall, wenn es sich bei dem Byte gerade um den Code eines *darstellbaren* Zeichens des Standard-ASCII-Zeichensatzes handelt: Der Code 65 steht im ASCII-Zeichensatz für das Zeichen "A". Taucht das Byte 65 in einem Byte-String auf, wird es also auf dem Terminal mit dem Zeichen "A" dargestellt. Der Code 6 steht für ein so genanntes Steuerzeichen, nämlich für das Signal ACK ("Acknowledge", das bedeutet: "Verstanden"). Solche Steuerzeichen sind nicht darstellbar. Die Standard-ASCII-Zeichensatz-Tabelle besitzt nur Codes zwischen 0 und 127. Bytes mit einem größeren Wert können somit auch nicht durch ein Standard-ASCII-Zeichen dargestellt werden.

Wenn nun ein Byte größer als 127 ist oder zu einem nicht darstellbaren Zeichen gehört, dann stellt Micropython es auf dem Terminal in **hexadezimaler Form** dar. Dabei wird die Hex-Zahl durch `\x` eingeleitet. Im Beispiel `b'\xf4\x06\x8d\xc6\x98y'` ist das erste Byte größer als 127; es kann also nicht durch ein Zeichen dargestellt werden. Das zweite Byte hat den Wert 6; wir wissen schon, dass es einem Steuerzeichen entspricht. Beide Bytes werden daher hexadezimal dargestellt. Lediglich das letzte Byte in diesem Beispiel gehört zu einem darstellbaren Zeichen (nämlich "y") und wird dann entsprechend auf dem Terminal ausgegeben.

Selbstverständlich kann man sämtliche Bytes hexadezimal eingeben, auch wenn es sich um darstellbare ASCII-Codes handelt: Gibt man am Terminal `print(b'\x41\x42\x43')` ein, wird jedoch `b'ABC'` ausgegeben. Dies ist für den Nutzer natürlich viel einfacher zu lesen als die Folge von Hexadezimalzahlen.

Die einzelnen Bytes, aus denen sich ein Bytes-Objekt zusammensetzt, kann man auch in eine Listenform bringen und umgekehrt; dazu benutzt man die Funktionen **`list()`** bzw. **`bytes()`**. Die Eingabe

```
>>> list(b'ABC')
```

hat z. B. das Ergebnis:

```
[65, 66, 67]
```

Umgekehrt kann man mit der `bytes`-Funktion eine Liste von einzelnen Bytes in ein bytes-Objekt umwandeln. Das Ergebnis von `bytes([72, 97, 108, 108, 111])` ist z. B. `b'Hallo'`.

Ganz lehrreich ist es, Bytes-Objekte mit deutschen Sonderzeichen wie z. B. `b = b'Käse'` zu betrachten. Wenn man die Länge dieses Objekts mit **`len(b)`** bestimmen lässt, erhält man den Wert 5. Auf den ersten Blick scheint das widersprüchlich zu sein, besteht das Wort 'Käse' doch nur aus 4 Buchstaben. Lassen wir uns das Objekt noch einmal anzeigen, erkennt man, dass das Objekt in Wirklichkeit so aussieht:

```
b'K\xc3\xa4se'
```

Die zugehörige Liste ist `[75, 195, 164, 115, 101]`. Offensichtlich wird das **Zeichen** "ä" hier **durch zwei Zahlen kodiert**, nämlich 195 = $C3 und 164 = $A4. Das ist die **UTF-8-Kodierung** des Zeichens "ä". Mehr Informationen dazu findet man z. B. unter https://www.utf8-zeichentabelle.de/ und https://realpython.com/python-encodings-guide/.

Den Datentyp `bytes` darf man nicht mit dem Datentyp `string` verwechseln. Einen ersten Unterschied erkennt man, wenn man von dem String `s = 'Käse'` mit `len(s)` die Länge bestimmen lässt. Hier ist der Wert 4; d. h. bei einem string-Objekt sind die **Zeichen** selbst relevant und nicht die **Codes**, mit denen sie gespeichert werden. Mit der `bytes`-Funktion lässt sich unser String `s` in den Typ `bytes` (unter Angabe der benutzten Zeichen-Kodierung) umwandeln:

```
b = bytes(s, 'UTF-8')
```

`b` hat jetzt wie erwartet den Inhalt `b'K\xc3\xa4se'`. Um eine Variable `b` vom Datentyp `bytes` in einen String umzuwandeln, wendet man die `str`-Funktion an:

```
s = str(b, 'UTF-8')
```

Schauen wir uns Abb. 10 ein letztes Mal an: Die beiden eckigen Klammern "[" und "]" tauchen jeweils nur einmal auf. Hier liegt demnach auch nur eine einzige Liste vor. Die Elemente dieser Liste bestehen aber ihrerseits aus Tupeln. Und die Elemente dieser Tupel besitzen verschiedene Datentypen. Insgesamt liegt also schon eine recht komplexe Struktur vor. Die Bedeutung der einzelnen Elemente werden wir im nächsten Kapitel klären.

E.6 Funktionen

Wenn eine Folge von bestimmten Befehlen im Verlauf eines Programms häufiger benutzt wird, bietet es sich an, diese zu einer Einheit zusammenzufassen. Eine solche Einheit wird in Micropython als **Funktion** bezeichnet. Die folgende Funktion `summe` soll als Beispiel für die Definition einer Funktion dienen:

```
def summe(liste):
  s = 0
  for l in liste:
    s += l
  return s
```

Sie bestimmt zu einer Liste, die als Parameter übergeben wird, deren Summe; diese Summe wird mit dem Schlüsselwort `return` zurückgegeben. Schreiben wir unter diese Funktion jetzt den Befehl `print(summe([12, 5, 18]))`, erhalten wir im Terminal als Ausgabe die Zahl 35.

Drei wichtige Bemerkungen noch zum Abschluss dieses kurzen Abschnitts über Funktionen:

- Wenn eine Funktion keine Parameter besitzt, werden die Klammern trotzdem hingeschrieben; sie bleiben dann leer.
- Funktionen können auch ohne Rückgabewert auskommen. In manchen Programmiersprachen würde man sie dann als Prozeduren bezeichnen. Mikropython macht hier keinen Unterschied: Wenn kein Rückgabewert erforderlich ist, lässt man einfach die `return`-Zeile weg.
- Alle Variablen, welche innerhalb der Definition einer Funktion auftauchen, sind (standardmäßig) **lokal**; von außerhalb kann man also nicht auf deren Werte zugreifen.

1 Grundlagen

Bluetooth stellt ein Funknetz für Kleingeräte wie z. B. Handys, Mäuse, Tastaturen oder Mikrophone dar, das speziell für Anwendungen mit kleinen Reichweiten und niedrigen Datenraten entwickelt worden ist. Bluetooth wurde in den 1990ger Jahren entwickelt. Die Bezeichnung Bluetooth geht zurück auf den dänischen König Harald Blauzahn; das Bluetooth-Logo (s. Abb. 1) ist ein Monogramm aus den beiden Buchstaben H und B, dargestellt mit den zugehörigen Runen ᚼ und ᛒ.

Abb. 1

In den folgenden Jahrzehnten hat sich der Bluetooth-Standard stetig weiter entwickelt. Seit dem Jahr 2010 gibt es neben der ursprünglichen Form (BR/EDR) eine neue Variante, nämlich **BLE (Bluetooth Low Energy)**. Diese zweite Variante soll Gegenstand dieses Skripts sein.

BLE benutzt eine andere Übertragungstechnik als BR/EDR. Diese ist ausgerichtet auf einen möglichst geringen Energiebedarf; zu diesem Zwecke sind die Datenraten auf 1 Mbps und die Sendeleistung auf maximal 10 mW beschränkt.

Der BLE-Standard beschreibt die Funktionsweise angefangen von den physikalischen Grundlagen der Funk-Übertagung bis zum Umgang mit den bereitgestellten Software-Einheiten. Die Darstellung folgt dabei dem in Abb. 2 dargestellten **Schichtenmodell**. Derartige Schichtenmodelle werden gerne zur Strukturierung von komplexen Systemen aus Hardware und/oder Software benutzt. In meinem Skript *ESP32-WLAN* habe ich im Kapitel *Client-Server-Modell* eine anschauliche Einführung für das beim Internet benutzte Schichtenmodell gegeben.

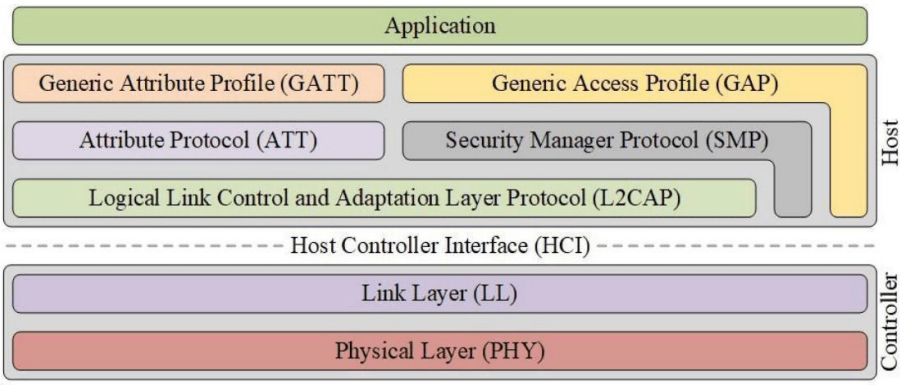


Abb. 2

In der Physikalischen Schicht (**Physical Layer**) werden die analogen Übertragungstechniken sowie die Umsetzung zwischen den analogen Signalen und den digitalen Daten beschrieben. BLE arbeitet im ISM-Band (**I**ndustrial, **S**cientifical and **M**edical) von 2,4 GHz, wobei es dieses Band in 40 Kanäle mit den Nummern 0 bis 39 aufteilt; sie besitzen jeweils eine Bandbreite von 1 MHz (s. Abb. 3). Dadurch wird die Datenübertragungsrate auf 1 Mb/s begrenzt; wir werden später sehen, dass die Übertragungsrate für die Nutzdaten wesentlich kleiner ist (vgl. Kap. 3).

Über die Kanäle mit den Nummern 37, 38 und 39 (in der Abb. 3 violett hervorgehoben), findet das Advertising, ein für die Kontaktaufnahme wichtiger Prozess (s. u.), statt. Diese Kanäle befinden sich außerhalb der WiFi-Kanäle, welche in Abb. 3 in gelber Farbe eingezeichnet sind.

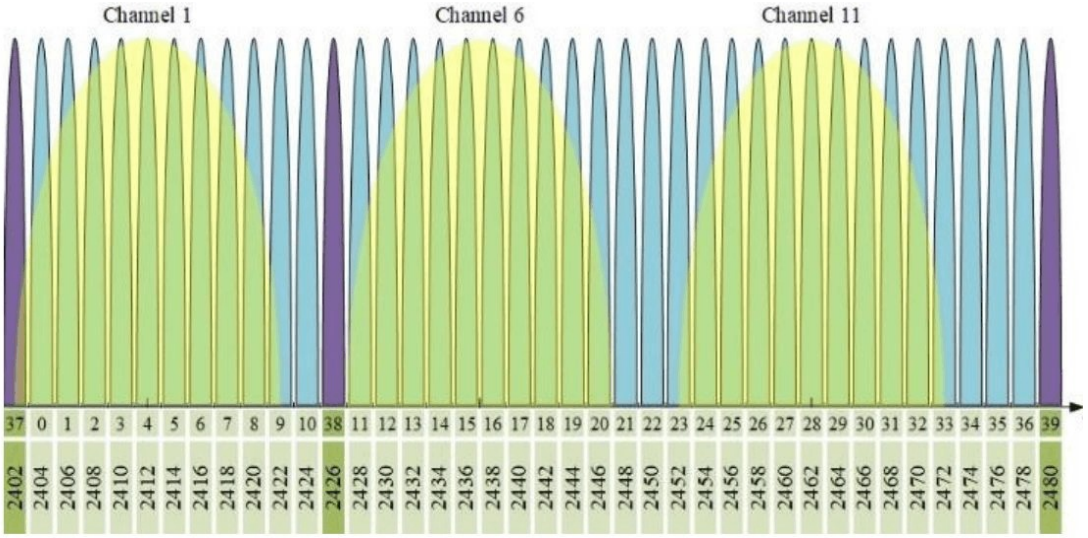


Abb. 3

In der nächsten Schicht (**Link Layer**) werden **Zustände (states)** und **Rollen (roles)** für BLE-Geräte definiert. Wichtige Zustände sind:

- **Advertising**: Ein Gerät sendet Advertising-Datenpakete (advertising packets, vgl. Kap. 3) über die Kanäle 37, 38 und 39 aus; diese Datenpakete beinhalten Informationen über das Gerät, z. B. seine BLE-Adresse.
- **Scanning**: Ein Gerät horcht nach Advertising-Datenpakete.
- **Connection**: Ein BLE-Gerät ist mit einem anderen verbunden und kann mit diesem Datenpakete austauschen.

Damit eine Verbindung zwischen zwei Geräten hergestellt werden kann, muss eines der Geräte im Advertising-Zustand sein: Es übt eine **Advertiser-Rolle** aus; das andere Geräte muss im Scanning-Zustand sein: Es übt eine **Scanner-Rolle** aus.

Es gibt Advertiser, die keine Verbindung zulassen; diese bezeichnet man als **Broadcaster**. Auf der anderen Seite gibt es auch Scanner, die keine Verbindung initiieren; diese nennt man **Observer**. Folgende Anwendung ist denkbar: Ein Bluetooth-Thermometer (Broadcaster) könnte in bestimmten Zeitabständen die gerade aktuelle Temperatur mit seinem Advertiser-Datenpaket aussenden. Diese Datenpakete könnten die Bluetooth-Handys in der Umgebung (Observer) empfangen, daraus den Temperaturwert herausfiltern und auf dem Display anzeigen. In Kap. 3 werden wir sehen, dass sich auf diese Weise allerdings nur wenige Bytes übertragen lassen.

Ein Advertiser, welcher durch seine Advertising-Datenpakete anzeigt, dass er für eine Verbindung zur Verfügung steht, wird **Peripheral** genannt. Ein Scanner, welcher Verbindungen initiieren ***kann***, wird als **Central** bezeichnet. Um eine Verbindung herzustellen, sendet er ein spezielles Datenpaket, eine sogenannte Verbindungsnachfrage (**Connection Request**), an den Peripheral. Akzeptiert der Peripheral diese Anfrage, so erhalten beide Geräte den **Connection-Zustand**; jetzt nehmen beide Geräte neue Rollen an: der Central wird zum **Master**, der Peripheral zum **Slave** (vgl. Abb. 4). Dabei ist es der Master, welcher die Kommunikation steuert.

Im nächsten Kapitel werden wir uns einige der gerade vorgestellten Zustände und Rollen an einem praktischen Beispiel anschauen.

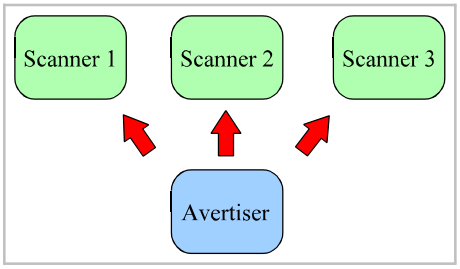


Abb. 4a

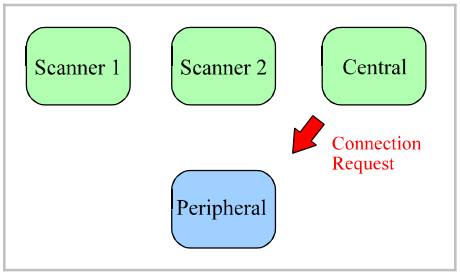


Abb. 4b

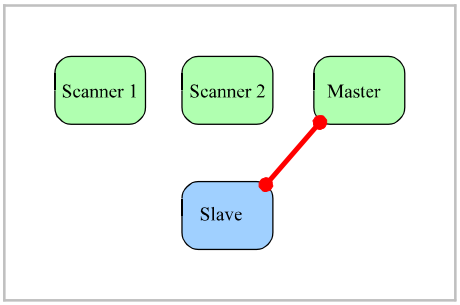


Abb. 4c

2 Experimente: Scan & Connect

In diesem Kapitel sammeln wir erste experimentelle Erfahrungen mit BLE. Als Scanner benutzen wir ein Android-Handy mit der App **nRF Connect** (aus dem Play-Store). Als Advertiser benutzen wir das TTGO-T-Display-Modul. Zunächst schließen wir eine LED über einen Vorwiderstand an den Pin 27 des ESP32 an (vgl. Einführungskapitel). Dann öffnen wir die Thonny-IDE, laden die Datei `experiment_1.py` und starten das Programm. Auf dessen Funktionsweise werden wir erst in einem späteren Kapitel zu sprechen kommen. Die LED fängt an zu blinken - ein Zeichen dafür, dass der TTGO jetzt im Advertising-Zustand ist. Dies wird auch in der Shell dokumentiert:

```
Advertising gestartet und läuft (Interrupt-gesteuert) weiter,
auch wenn das Prompt-Zeichen >>> erscheint...

Dieser Vorgang lässt sich mit Hilfe des Reset-Buttons beenden.

ESP32:
mac_type: 0 ; address:  ['0x84', '0xcc', '0xa8', '0x61', '0x7', '0x72']

>>>
```

In der letzten Zeile finden wir außerdem die **Bluetooth-Adresse** (genauer: Public Device Address, s. u.) des TTGO-Moduls als Liste von sechs Hexadezimalzahlen.

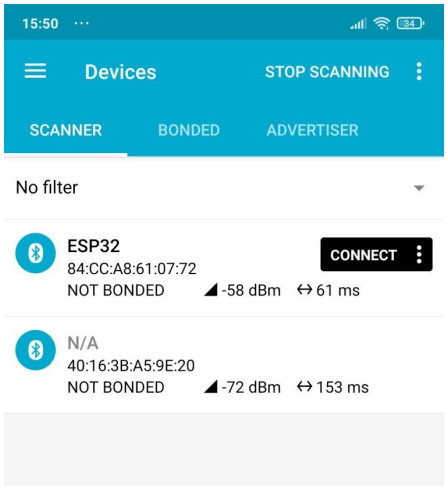


Abb. 1

Nun kontrollieren wir, ob bei unserem Handy Bluetooth aktiviert ist und öffnen dann die App **nRF Connect**. Die App startet sofort einen Scan-Vorgang. Dieser wird automatisch nach einiger Zeit beendet. Man kann ihn aber über die Einträge STOP SCANNING bzw. SCAN in der oberen Menüzeile jederzeit abbrechen oder erneut starten.

In Abb. 1 erkennt man unser TTGO-Modul an dem Namen "ESP32". Unterhalb des Namens finden wir außerdem die schon bekannte Bluetooth-Adresse des TTGO-Moduls.

Durch die Angabe -58 dBm wird hier die **Feldstärke** des empfangenen Signals angegeben: -58 dBm bedeutet eine relativ hohe Signalstärke; das Handy lag hier in unmittelbarer Nähe zum TTGO-Modul. In einigen Meter Entfernung wird nur noch ein Wert von ungefähr -75 dBm angezeigt. Neben dem Feldstärkewert finden wir auf dem Display noch die Angabe 61 ms. Dies gibt das **Advertising-Intervall** an; grob gesagt ist dies die Zeitspanne zwischen zwei aufeinander folgenden Advertising-Paketen. (Mehr dazu in Kap. 6)

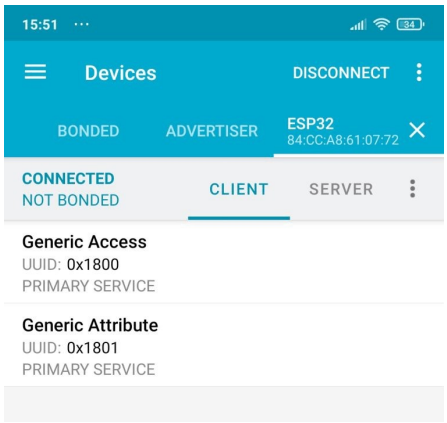


Abb. 2

Nun betätigen wir den schwarzen Button CONNECT. Die LED leuchtet jetzt dauerhaft und die App zeigt an, dass das Handy und das TTGO-Modul nun verbunden sind (vgl. Abb. 2), m. a. W.: Sie befinden sich im Connect-Zustand.

Im Terminal wird dies so dokumentiert:

```
connected
address_type: 1 ; address: ['0x72', '0x32', '0xb1', '0x75', '0xf', '0xd7']
```

In den eckigen Klammern finden wir 6 Hexadezimalzahlen, welche die Bluetooth-Adresse des Handys angeben. Diese wurde von dem Handy (Central) an das TTGO-Modul (Peripheral) gesendet.

Über den Eintrag DISCONNECT in der oberen Menüzeile können wir die Verbindung wieder auflösen; die LED blinkt dann wieder - ein Zeichen dafür, dass das TTGO-Modul erneut im Advertising-Zustand ist. In der Shell wird außerdem angezeigt:

```
disconnected
address_type: 1 ; address: ['0x72', '0x32', '0xb1', '0x75', '0xf', '0xd7']
```

Wartet man etwas längere Zeit, so kann man irgendwann auch eine andere Bluetooth-Adresse erhalten:

```
connected
address_type: 1 ; address: ['0x59', '0x4b', '0xf2', '0xf7', '0xf1', '0xeb']
```

Bei der am Terminal ausgegebenen Adresse handelt es sich also nicht um die Bluetooth-Hardware-Adresse (**Public Device Address**) des Handys; diese ist nämlich fest und unveränderlich. Vielmehr liegt hier eine so genannte **Random Device Address** vor, also eine vom Handy mit Zufallszahlen erzeugte Bluetooth-Adresse. Um zu erfahren, um welchen dieser beiden **Adress-Typen** es sich bei der ausgegebenen Adresse handelt, muss man sie nun nicht über einen längeren Zeitraum kontrollieren; vielmehr wird dies bereits durch den Wert des Parameters `address_type` angezeigt:

Wert von `address_type`	Bedeutung
0	Public Device Address
1	Random Device Address

Die Anzeige `address_type: 1` bestätigt, dass es sich bei der angezeigten Adresse des Handys tatsächlich um eine Random Device Address handelt.

Ein Blick hinter die Kulissen: Es gibt drei unterschiedliche Kategorien vom Typ Random Device Address:

Static Address (SA): Sie wird oft als Ersatz für eine Public Device Address benutzt, wenn der Hersteller des Geräts eine IEEE-Registrierung auslassen möchte. Diese Adresse wird zufällig beim Einschalten generiert und kann bis zum Abschalten auch nicht mehr geändert werden.

Non-Resolvable Private Address (NRPA): Diese Adressen werden aus einer Zufallszahl erzeugt und dienen als temporäre Adresse.

Resolvable Private Address (RPA): Diese Adressen werden aus einem Sicherheitsschlüssel und einer Zufallszahl generiert. Sie können jederzeit geändert werden, selbst im CONNECT-Zustand.

Um welche Kategorie es sich bei einer Random Device Address handelt, kann man am höchstwertigsten Byte der Adresse erkennen; in unserem ersten Fall ist dies `0x72`. Nun schauen wir uns dieses Byte in binärer Darstellung an:

$0x72 = 01110010_2$

Die beiden höchstwertigsten Bits (hier rot markiert) bilden den Kennwert für die Kategorie:

Kennwert	Kategorie
00_2	NRPA
01_2	RPA
11_2	SA

In unserem Fall liegt also eine Resolvable Private Address vor. Wer mag, kann dies auch einmal für die zweite Adresse überprüfen.

Weitere Informationen zu diesem Thema findet man in dem Abschnitt 1.3.2.2 der *Bluetooth Core Specification 5.2*, s. [6].

3 Struktur der Advertising-Datenpakete

In diesem Kapitel stellen wir die Strukturelemente der Advertising-Datenpakete vor, wie sie für das Verständnis der nächsten Kapitel erforderlich sind. Alle **BLE-Datenpakete** haben die folgende Grundstruktur:

Preamble	Access Address	Protocol Data Unit (PDU)	CRC
1 Byte	4 Bytes	2-257 Bytes	3 Bytes

Abb. 1 (Quelle: [1])

Durch die Präambel werden die Taktgeber von Sender und Empfänger synchronisiert. Für das Advertising spielt die Access Address keine Rolle. Es folgt die **P**rotocol **D**ata **U**nit (kurz: **PDU**); diese enthält u. A. die eigentlichen Nutzdaten (s. u.). Ihre Länge kann zwischen 2 und 257 Bytes variieren. Das Ende wird von drei CRC-Bytes gebildet: Mit diesen Prüfwerten können ggf. Fehler entdeckt werden, die bei der Übertragung des Datenpakets aufgetreten sind (**C**yclic **R**edundancy **C**heck).

Die **Advertising-PDU** besteht aus einem **Header** (Kopf) und dem **Payload** (Nutzdaten). Der Header ist zwei Bytes groß und hat folgende Struktur:

PDU Type	RFU	TxAdd	RxAdd	Length	RFU
4 bits	2 bits	1 bit	1 bit	6 bits	2 bits

Abb. 2 (Quelle: [1])

In dem Feld **PDU Type** stehen 4-Bit-Konstanten, welche den **Verbindungstyp** (kurz auch: **ADV-Typ**) angeben, die der Advertiser anbietet. Hier einige Beispiele:

Wert	ADV-Typ (PDU Type)	Bedeutung
0	ADV_IND	Gerät kann gescannt und auch verbunden werden. Das Advertising richtet sich an alle erreichbaren BLE-Geräte.
2	ADV_SCAN_IND	Gerät kann gescannt, aber nicht verbunden werden. Das Advertising richtet sich an alle erreichbaren BLE-Geräte.
3	ADV_NONCONN_IND	Gerät kann weder gescannt noch verbunden werden. Das Advertising richtet sich an alle erreichbaren BLE-Geräte.
4	SCAN_RSP	Scan Response (s. Kap. 6)

Mit dem Bit **TxAdd** wird der Adress-Typ festgelegt: Hat das Bit den Wert 0, liegt eine Public Device Address vor, hat es den Wert 1, handelt es sich um eine Random Device Address (vgl. Kap. 2).

Das Feld *Length* gibt an, aus wie vielen Bytes der Payload (ausschließlich der Bluetooth-Adresse) besteht. Der **Payload** selbst kann (inkl. der Bluetooth-Adresse) maximal 37 Bytes lang sein; für die eigentlichen Advertising-Daten stehen demnach lediglich höchstens 31 Bytes zur Verfügung:

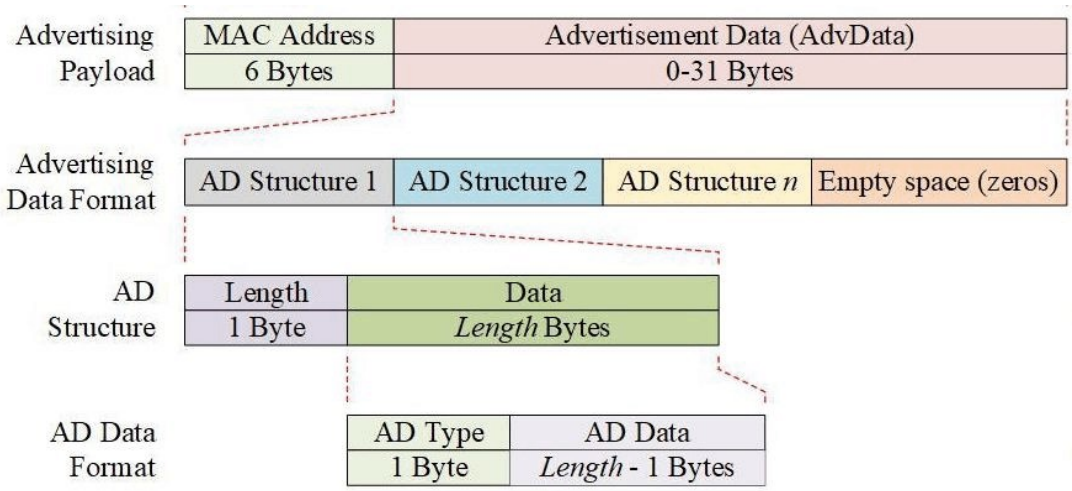


Abb. 3 (Quelle: [1])

Diese setzen sich aus einzelnen Datensätzen (**AD Structures**) zusammen, welche z. B. Zeichenketten oder Zahlen mit bestimmten Bedeutungen angeben; sie werden in einem einheitlichen Format dargestellt: Alle AD-Struktur werden jeweils angeführt durch ein Length-Feld; darauf folgt das eigentliche Datum, welches seinerseits aus einer Typ-Angabe und den eigentlichen Daten (z. B. Text oder Zahl) besteht.

Hier eine Auswahl von möglichen Typ-Angaben:

Typ-Wert	Bezeichnung der Konstanten	Bedeutung
0x01	ADV_TYP_FLAGS	Flags
0x09	ADV_TYP_NAME	Name des Advertisers
0x0A	TX_POWER_LEVEL	Tx-Power-Level

Für den Empfänger besteht der Payload zunächst lediglich aus einer Reihe von Bytes, z. B.:

```
0x84 0xcc 0xa8 0x61 0x07 0x72 0x02 0x01 0x02 0x06 0x09 0x52 0x65 0x64 0x6d
0x69 0x02 0x0a 0xf9
```

Diese Bytefolge kann der Empfänger nun folgendermaßen entschlüsseln: Zunächst trennt er die ersten 6 Bytes ab; diese bilden die Bluetooth-Adresse des Senders (blau). Mit dem nächsten Byte

beginnt die erste AD-Struktur (grün). Das erste Byte `0x02` gibt den Längen-Wert dieser ersten AD-Struktur an; diese besitzt demnach nur zwei Datenbytes: die Type-Angabe `0x01` und den Flag-wert `0x02` (für den LE General Discoverable Mode). Mit dem nächsten Byte beginnt eine zweite AD-Struktur (braun): Der Längenwert ist 6 und die Byte-Folge `0x52 0x65 0x64 0x6d 0x69` gibt den Namen des Advertisers an. Mit Hilfe einer ASCII-Tabelle ergibt sich aus den Hex-Codes die Zeichenkette "Redmi". Die Entschlüsselung der letzten 3 Bytes bleibt dem Leser überlassen.

Der Längenwert gestattet es uns, die Bytefolge in die passenden Struktur-Blöcke zu zerlegen; die Type-Angabe gibt dann an, welche Bedeutung die zugehörigen AD-Daten jeweils haben. Damit ist auch klar, dass die Reihenfolge der einzelnen AD-Strukturen keine Rolle spielt.

Mit diesen Informationen kann man nun z. B. eine Micropython-Funktion `find_adv_name(data)` schreiben, welche aus einer solchen Byte-Folge den Namen des Advertisers zurückgibt. Dabei soll es sich bei dem Argument `data` um den Payload ***ohne*** die Bluetooth-Adresse handeln.

```
def find_adv(adv_type, data):
    i = 0
    while i + 1 < len(data):
        ad_structure_len = data[i]
        ad_structure_type = data[i + 1]
        ad_structure_payload = data[i + 2: i + ad_structure_len + 1]
        if ad_structure_type == adv_type:
            return ad_structure_payload
        i += ad_structure_len + 1
    return None

def find_adv_name(data):
    n = find_adv(9, data)
    if n: # wenn n nicht None
        return str(n, 'UTF-8') # Text
    return None
```

Wie funktionieren nun die beiden Funktionen? Zunächst wird durch `find_adv_name(data)` die Funktion `find_adv(adv_type, data)` mit dem Wert 9 für `adv_type` aufgerufen. In einer `while`-Schleife wird die Bytefolge `data` nun AD-Struktur für AD-Struktur durchsucht, bis eine AD-Struktur mit dem Typ-Wert 9 gefunden wird. Die zugehörigen Bytefolge wird an die Funktion `find_adv_name` zurückgegeben. (Wird keine AD-Struktur vom Typ 9 gefunden, wird `None` zurück gegeben.) Diese Bytefolge wird nun in eine Zeichenkette umgewandelt, welche schließlich als Rückgabewert dient.

Im übernächsten Kapitel werden wir ein Scan-Programm für den TTGO vorstellen und austesten. Hierbei lassen sich unsere beiden Funktion einsetzen, um den Namen des Advertisers im Terminal auszugeben. Im nächsten Kapitel müssen wir uns aber erst noch mit den so genannten Interrupts beschäftigen; diese werden nämlich im Umgang mit dem Senden und Empfangen von Bluetooth-Datenpaketen eine wichtige Rolle spielen.

4 Interrupt-Intermezzo

Interrupts sind Unterbrechungen des normalen Programmablaufs, welche durch Ereignisse (**Events**) ausgelöst werden. Solche Events können z. B. darin bestehen, dass

- ein Taster gedrückt oder losgelassen wird
- ein Timer-Bausteine einen "Tick" abgibt
- ein BLE-Baustein ein Datenpaket empfängt

Natürlich könnte man in einem laufenden Hauptprogramm immer wieder den Zustand eines Tasters abfragen und dann dementsprechende Aktionen ausführen lassen. Dies würde das Haupt-Programm aber oft überfrachten. Interrupts bieten hier meist eine bessere Möglichkeit: Man sorgt dafür, dass durch das Taster-Event[1] der Mikrocontroller den Ablauf des Hauptprogramms unterbricht und zu einem Programm-Abschnitt springt, welches **Interrupt-Behandlungs-Routine** (**Interrupt Handler** oder **Event Handler**[2]) genannt wird. In diesem Programm-Abschnitt werden diejenigen Aktionen ausgeführt, welche direkt auf den Tastendruck folgen sollen. Danach kehrt der Mikrocontroller automatisch wieder an die Stelle im Hauptprogramm zurück, an der die Unterbrechung stattgefunden hatte.

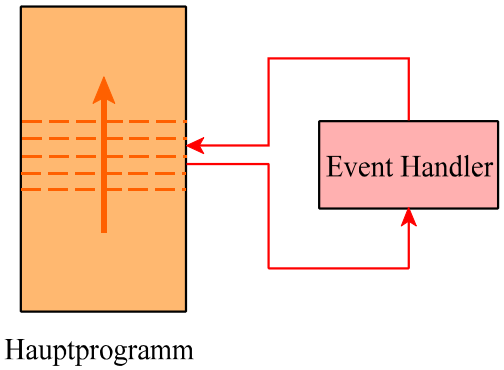


Abb. 1

Die Unterbrechung erfolgt unmittelbar nach dem Ereignis: Das bedeutet insbesondere, dass auch ein einzelner Micropython-Befehl wie z. B. print('Testen') mitten in seiner Ausführung, z. B. bei der Ausgabe des Buchstaben 't', unterbrochen werden kann.

An zwei Beispielen wollen wir verdeutlichen, wie man bei Micropython solche Interrupts programmieren kann.

Beispiel 1: Das Hauptprogramm besteht aus einer Schleife, welches eine LED blinken lässt (vgl. Einführungskapitel). Jedesmal, wenn einer der beiden Tasten des TTGO gedrückt wird, soll eine entsprechende Nachricht auf dem Terminal erscheinen.

---

[1] Genauer: durch ein entsprechendes Signal, welches auch als Interrupt Request, kurz IRQ, bezeichnet wird

[2] Genau genommen beziehen sich Interrupts auf Low-Level-Ereignisse und Events auf softwaremäßige Ereignisse; wir wollen hier diese Unterscheidung nicht machen!

Das Programm sieht so aus:

```
from machine import Pin
from time import sleep

def handleMyPinInterrupt(myPin):
    print(myPin, 'wurde betätigt.')

taster0 = Pin(0, Pin.IN, Pin.PULL_UP)
taster0.irq(trigger=Pin.IRQ_FALLING, handler=handleMyPinInterrupt)

taster1 = Pin(35, Pin.IN) # externer Pullup-Widerstand auf der TTGO-Platine
taster1.irq(trigger=Pin.IRQ_FALLING, handler=handleMyPinInterrupt)

led = Pin(27, Pin.OUT)
while True:
    led.value(1)
    sleep(0.2)
    led.value(0)
    sleep(0.2)
```

Schauen wir uns das Programm an: Im unteren Teil erkennen wir das "Hauptprogramm": unsere bereits bekannte Blink-Schleife. Darüber sehen wir, wie `taster0` und `taster1` als Instanzen der Klasse `Pin` erzeugt werden; beide sind als Eingänge konfiguriert. Diese beiden Instanzen besitzen eine Methode namens `irq`. Mit Hilfe dieser Methode werden zwei Dinge festgelegt:

1. Zunächst wird mit `trigger=Pin.IRQ_FALLING` festgelegt, **dass** ein Interrupt ausgelöst werden soll, wenn das Eingangssignal von High auf Low wechselt.

2. Dann wird mit `handler=handleMyPinInterrupt` festgelegt, **welche Funktion** bei dem Interrupt ausgeführt werden soll; in diesem Fall ist es die Funktion `handleMyPinInterrupt`.

Die Micropython-Dokumentation für die irq-Methode eine Pin-Objekts gibt an, dass als Interrupt Handler eine Funktion mit genau einem Parameter erwartet wird. Und diesem Parameter wird dann bei einem Interrupt dasjenige Pin-Objekt zugewiesen, welches den Interrupt ausgelöst hat. Auf diese Weise kann man dann auch innerhalb des Event Handlers auf dieses Pin-Objekt zugreifen. In unserem Fall geben wir mit dem print-Befehl einfach die Kennung unseres Pins an (Pin(0) bzw. Pin(35)) auf dem Terminal aus.

Beachten Sie dabei: Für beide Taster benutzen wir denselben Event Handler. Weil das Pin-Objekt aber als Parameter an diese Funktion übergeben wird, "weiß" der Event Handler bei der Ausführung des print-Befehls die richtige Kennung.

Sie sollten nun das Programm (`pin_irq.py`) einmal selbst ausprobieren. Zuvor muss allerdings noch eine LED über einen Vorwiderstand an Pin 27 angeschlossen werden.

Sicherlich werden auch Sie dabei bemerken, dass beim Loslassen(!) von `taster1` meist mehrere Interrupts ausgelöst werden. Offensichtlich ist dieser Taster nicht gut entprellt.

Beispiel 2: Als Hauptprogramm dient wieder eine Schleife, welche eine LED an Pin 27 langsam blinken lässt. Gleichzeitig soll eine weitere LED an Pin 25 rasch blinken; hierzu soll einer der Hardware-Timer zum Einsatz kommen, die in dem ESP32 integriert sind.

Das Programm (`timer_irq.py`) sieht so aus:

```
from machine import Pin, Timer
from time import sleep

def handleTimerInterrupt(myTimer):
  global led1
  led1.value(not led1.value()) # toggeln
  global counter
  counter = counter + 1
  print(counter)
  if counter == 50:
      myTimer.deinit()
      print('Timer deaktiviert')

counter = 0
led0 = Pin(27, Pin.OUT)
led1 = Pin(25, Pin.OUT)

timer = Timer(0) # 4 Hardware-Timer stehen zur Verfügung (Nummern: 0-3)
timer.init(freq=5, callback=handleTimerInterrupt)

while True:
  led0.value(1)
  sleep(3)
  led0.value(0)
  sleep(1)
```

Unten sehen wir wieder unsere Blink-Schleife. Darüber finden wir die Instanziierung von `timer`, mit welchem `led1` zum Blinken gebracht werden soll. Mit der `timer`-Methode `init` werden nun zwei Dinge festgelegt:

1. Zunächst wird mit `freq=5` festgelegt, **dass** jede 1/5 Sekunde ein Timer-Interrupt ausgelöst werden soll.

2. Dann wird mit `callback=handleTimerInterrupt` festgelegt, **welche Funktion** bei dem Interrupt ausgeführt werden soll; in diesem Fall ist es die Funktion `handleTimer-Interrupt`.

Dass hier der Event Handler den Attributnamen `callback` hat, soll uns hier nicht stören; dies ist einfach so vorgegeben und der Timer-Dokumentation zu entnehmen[3]. Dort ist ebenfalls festgelegt, dass als Event Handler eine Funktion mit genau einem Parameter erwartet wird. Und diesem Parameter wird dann bei einem Interrupt dasjenige Timer-Objekt zugewiesen, welches den Interrupt ausgelöst hat.

Der Parameter hat hier die Bezeichnung `myTimer`; und damit können wir innerhalb des Event Handlers dafür sorgen, dass nach dem 50-ten Blinken von led1 der Timer mit dem Befehl `myTimer.deinit()` deaktiviert wird.

Bei jedem Timer-Interrupt wird nun `led1` getoggelt. Damit innerhalb der Funktion auf das im Hauptprogramm definierte Objekt `led1` zugegriffen werden kann, muss es als `global` deklariert werden; gleiches gilt für die Zählvariable `counter`.

Ein Blick hinter die Kulissen: Event Handler sollten im Allgemeinen kurz gehalten werden. Dadurch wird der zeitliche Ablauf des Hauptprogramms kaum spürbar unterbrochen. Innerhalb eines Event Handlers ist nur bei wenigen Typen eine Definition als lokale Variable zuverlässig möglich. Bei einer Definition etwa von Zeichenketten oder Listen müsste nämlich Speicherplatz auf dem Heap (Stapel) reserviert werden, und das kann zu Problemen führen, wenn der Interrupt genau während einer Speicherzuweisung im Hauptprogramm erfolgt.

Die Micropython-Dokumentation sagt dazu: *Note that your callback functions must not allocate any memory (for example they cannot create a tuple or list). Callback functions should be relatively simple. If you need to make a list, make it beforehand and store it in a global variable (or make it local and close over it). If you need to do a long, complicated calculation, then use the callback to set a flag which some other code then responds to*.

Achten Sie bitte beim Austesten des Programms einmal darauf: Das Blinken von `led0` wird durch die Interrupts nicht spürbar beeinträchtigt.

[3] Event Handler werden manchmal auch Callback-Funktionen genannt.

5 Scan-Programm

Um mit der Bluetooth-Hardware des TTGO arbeiten zu können, laden wir als erstes aus dem Modul `ubluetooth` die Klasse `BLE`:

```
from ubluetooth import BLE
```

Damit erzeugen wir eine Instanz `ble`:

```
ble = BLE()
```

Wir aktivieren nun das Bluetooth-Modul und versetzen es in den Scanning-Zustand:

```
ble.active(True)
ble.gap_scan(5000, 100000, 25000)
```

Die Parameter der Methode `gap_scan` haben die folgende Bedeutung: Der erste Parameter (**Scan-Dauer**) gibt die Gesamtdauer des Scan-Vorgangs in Millisekunden an; in unserem Fall dauert der Scan-Vorgang 5 Sekunden. In dieser Zeit wird aber nicht ununterbrochen gescannt; das hilft, Energie zu sparen. Vielmehr wird in bestimmten Zeitabständen nur für eine kurze Zeit gescannt. In unserem Fall wird nur jede 100000 Mikrosekunden (zweiter Parameter: **Scan-Intervall**) für jeweils 25000 Mikrosekunden (dritter Parameter: **Scan-Fenster**) gescannt. Diese Vorgehensweise stellt einen wesentlichen Beitrag zur Energieersparnis dar.

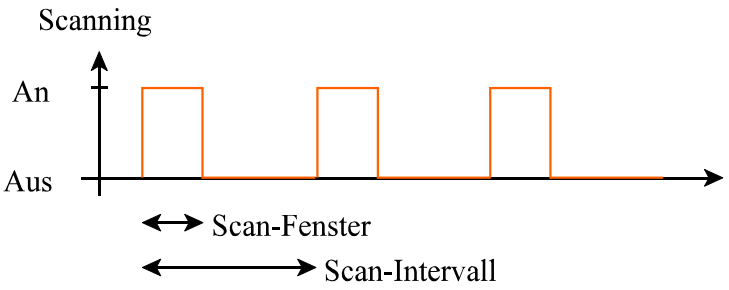


Abb. 1

Dabei wechselt der Empfänger fortlaufend zwischen den **Advertiser-Kanälen** 37, 38 und 39. Auch der Advertiser sendet seine Datenpakete nicht andauernd aus: Jedes Datenpaket wird direkt hintereinander auf den Kanälen 37, 38 und 39 ausgesendet (> Kap. 8). Es folgt eine Pause; danach wird das Datenpaket wieder auf den drei Kanälen gesendet u.s.w. Ein vom Advertiser ausgesandtes Datenpaket kann nur innerhalb des Scan-Fensters vom Scanner registriert werden; zusätzlich müssen beide gerade auf demselben Kanal arbeiten.

Entdeckt die Bluetooth-Hardware des ESP32 nun ein Datenpaket, löst sie einen Interrupt aus. Mit der Methode `ble.irq` können wir solchen Ereignissen einen Event Handler zuweisen; dazu übergeben wir den Event Handler als Parameter an die Methode `ble.irq` (Diese Zuweisung erfolgt sinnvollerweise, bevor der Scan gestartet wird):

```
ble.irq(bt_irq_handler)
```

In diesem Fall heißt unser Event Handler `bt_irq_handler`. Die Methode `irq` von `ble` erwartet, dass der Event Handler zwei Parameter besitzt. Dem ersten Parameter wird eine Zahl zugewiesen, welche den Event-Typ charakterisiert: Ist diese Zahl z. B. gleich 5, dann wurde ein Advertiser-Datenpaket empfangen, ist die Zahl gleich 6, wurde das Scannen beendet. Wir werden später noch weitere Ereignistypen mit ihren Kennzahlen kennen lernen. Der zweite Parameter erhält im Wesentlichen die Payload-Daten.

Der Event Handler sieht in unserer ersten Fassung so aus:

```
def bt_irq_handler(event, data):
  if event == 5: # Advertiser-Datenpaket erhalten
    print(' Data: ',data) # Ausgabe der Daten
  elif event == 6: # Scan beendet
    print('-------------------------------------------------------------')
    print('scan complete')
```

Wir laden das Programm (`ble_scan_v0a.py`) in die Thonny-IDE und starten es. Meist befinden sich in der Nähe einige Bluetooth-Geräte, die Datenpakete aussenden. Nach dem Start erhalten wir dann eine Terminal-Ausgabe ähnlich wie die folgende:

```
Data: (0, <memoryview>, 3, -78, <memoryview>)
Data: (0, <memoryview>, 3, -79, <memoryview>)
Data: (0, <memoryview>, 3, -90, <memoryview>)
Data: (0, <memoryview>, 3, -91, <memoryview>)
Data: (0, <memoryview>, 3, -91, <memoryview>)
Data: (0, <memoryview>, 3, -76, <memoryview>)
Data: (0, <memoryview>, 3, -76, <memoryview>)
Data: (0, <memoryview>, 3, -76, <memoryview>)
-------------------------------------------------------------
scan complete
```

Hier traten also 8 Events vom Typ 5 und am Ende eines vom Typ 6 auf.

Schauen wir uns nun die erhaltenen Daten an: Die runden Klammern weisen darauf hin, dass es sich hierbei jeweils um ein Tupel handelt. Die Bedeutung der einzelnen Elemente eines solchen Tupels entnimmt man der `ubluetooth`-Dokumentation:

Element-Nr.	Bedeutung	Quelle	Bemerkung
0	Adress-Typ	TxAdd-Bit im Header	0: Public (vgl. Kap.2 u. 3) 1: Random
1	Bluetooth-Adresse des Advertisers	Payload	`<memoryview>` s.u.
2	Verbindungstyp	Header	3: ADV_NONCONN_IND (vgl. Kap. 3)
3	RSSI: Ein Wert für die Stärke des empfangenen Signals	BT-Hardware des TTGO	Hier ist -76 dBm der Wert des stärksten Signals.
4	Payload (ohne Bluetooth-Adresse)	Payload	`<memoryview>` s.u.

Zwei der Elemente des Tupels können nicht direkt mit dem `print`-Befehl ausgegeben werden; hier handelt es sich um sogenannte `memoryview`-Instanzen, die auf einen internen Ringpuffer von `ubluetooth` verweisen. Um diese Instanzen müssen wir uns aber nicht genauer kümmern; wir werden gleich sehen, wie wir recht einfach die im Ringpuffer gespeicherten Daten auf dem Terminal ausgeben können.

Zunächst ordnen wir die Elemente des Tupels `data` mit Hilfe der Programmzeile

```
addr_type, addr, adv_type, rssi, adv_data = data
```

einzelnen Variablen mit aussagekräftigen Namen zu. Damit können wir durch

```
print('Address-Type: ', addr_type)
```

z. B. den Adress-Typ auf dem Terminal anzeigen lassen. Entsprechend verfährt man mit den Variablen `adv_type` und `rssi`.

Die Bluetooth-Adresse könnten wir mit

```
print('Address: ', bytes(addr))
```

ausgeben. Übersichtlicher ist es aber, wenn man sich die einzelnen Bytes als Liste von Hexadezimalzahlen anzeigen lässt:

```
print('Address: ', [hex(n) for n in addr])
```

Das Ergebnis sieht dann z. B. so aus:

```
Address-Type:  0
Address:  ['0x40', '0x16', '0x3b', '0xa5', '0x9e', '0x20']

Address-Type:  1
Address:  ['0x79', '0xd6', '0xcf', '0x73', '0xd8', '0xf6']

Address-Type:  0
Address:  ['0x40', '0x16', '0x3b', '0xa5', '0x9e', '0x20']

...
```

Es ist jetzt leicht, auch die restlichen Elemente des Tupels zur Anzeige zu bringen. Das entsprechende Programm findet man in der Datei `ble_scan_v0b.py`. Testen Sie das Programm bitte einmal selbst aus!

Nun wollen wir aus dem Payload `data` den Namen des Advertisers herausfiltern und als Zeichenkette im Terminal anzeigen lassen. Dazu benutzen wir die beiden Funktionen `find_adv_name` und `find_adv`, welche wir schon in Kapitel 3 vorgestellt haben. Wir fügen sie in unser Programm direkt unter den bereits bestehenden Import-Befehl ein. Außerdem ergänzen wir den Event Handler um die Zeile

```
print('adv_data (Name): ', find_adv_name(adv_data))
```

Das resultierende Programm finden Sie auch unter dem Namen `ble_scan_v0c.py`.

Um das Programm, insbesondere die korrekte Übertragung des Namens zu testen, benötigen wir einen Advertiser mit einem bekannten Namen. Hierzu können wir wieder unser Handy mit der App nRF Connect einsetzen. Wir starten diese App und wählen in der unteren Menü-Zeile die ADVERTISER-Option.

Auf dem Display erkennen wir dann ein vorbereitetes Advertiser-Profil mit dem Namen nRF Connect (Abb. 2). Wir wollen jetzt aber ein eigenes Advertiser-Profil herstellen. Dazu betätigen wir den Plus-Button in der rechten unteren Ecke. Es offnet sich nun ein Fenster, in welchem wir zunächst den von uns gewünschten Profilnamen-Namen eingeben. In Abb. 3 habe ich als Profilnamen *Adv Redmi* gewählt. Um nun weitere Advertiser-Daten einzugeben,

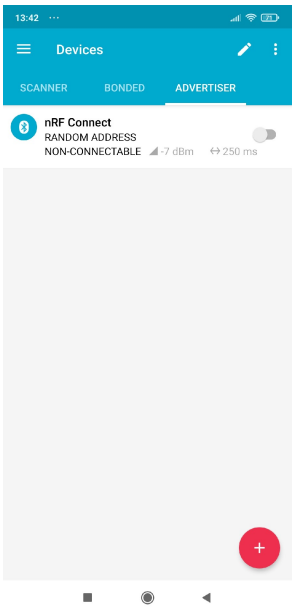


Abb. 2

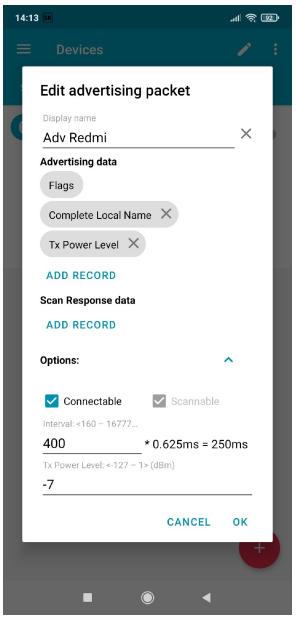


Abb. 3

benutzen wir die Option ADD RECORD. In dem jetzt angezeigte Pop-Up-Menü klicken wir zumächst *Complete Local Name* an; damit wird der Bluetooth-Name des Handys in den Payload des Advertisers aufgenommen. Auf die gleiche Weise sorgen wir dafür, dass auch die Sendeleistung *(Tx Power*) in den Payload eingefügt wird. Bei den Optionen aktivieren wir die Option *Connectable* (*Scannable* wird dann auch automatisch aktiviert.). Mit der Pfeil-Taste können wir noch weitere Parameter einstellen: das Advertising-Intervall und die Sendeleistung. Wir lassen sie zunächst unverändert. Die Eingaben schließen wir mit dem OK-Button ab. Auf dem Display finden Sie jetzt Ihr neues Advertiser-Profil.

Am rechten Rand dieses Profil-Eintrags finden Sie einen Schieber. Wenn Sie ihn berühren, öffnet sich ein Fenster, mit dessen Hilfe Sie den Advertiser für eine bestimmte Zeit aktivieren können. Nach der Aktivierung starten wir sofort unser Scan-Programm. Bei meinem Testlauf finde ich einige Einträge, die von meinem Handy (Redmi Note 6 Pro) stammen:

```
Address-Type:  1
Address:  ['0x40', '0xc2', '0x81', '0x71', '0xd0', '0x84']
ADV-Type:  0
RSSI in dBm:  -74
ADV-Data:  ['0x2', '0x1', '0x1', '0x11', '0x9', '0x52', '0x65', '0x64',
'0x6d', '0x69', '0x20', '0x4e', '0x6f', '0x74', '0x65', '0x20', '0x36',
'0x20', '0x50', '0x72', '0x6f', '0x2', '0xa', '0xf9']
ADV-Data (Name):  Redmi Note 6 Pro
```

Sollten Sie Ihr Handy auch nach mehreren Versuchen nicht auf dem Terminal finden, hilft es vielleicht, die Dauer des Advertisings und des Scannens zu vergrößern.

An dieser Stelle bietet es sich an, das Scan-Programm so zu erweitern, dass es auch den Tx-Power-Level ausgibt. Benutzen Sie dazu die Informationen aus dem Kapitel 3 und schreiben Sie damit eine entsprechende Funktion `find_adv_power`. Hinweis: Den Tx-Power-Level erhält man als erstes Element des Rückgabewertes von `find_adv`.

Vielleicht werden Ihnen die erhaltenen Tx-Power-Level-Werte merkwürdig vorkommen; sie stimmen nicht mit dem im Advertiser festgelegten Wert überein. Der Grund ist: Diese Werte können auch negativ sein. Solche Zahlen werden durch so genannte **signed bytes** dargestellt (vgl. die folgende Box *Hinter den Kulissen*...). Unsere Funktion `find_adv_power` erhält von der Funktion `find_adv` eine Liste `n` mit einem einzigen (Byte-)Element als Rückgabewert. Wenn wir dieses Byte einer Variablen zuordnen, dann wird dieses Byte als Zahl ohne Vorzeichen (**unsigned byte**) gedeutet. Handelt es sich z. B. bei `dBm = n[0]` um das Byte 11111001_2, wird `dBm` beim `print`-Befehl `print(dBm)` entsprechend als `249` ausgegeben. Um diesen Wert als signed byte zu deuten, subtrahieren wir einfach die Zahl 256 von dBm, falls dBm größer als 127 ist:

```
if dBm > 127:
    dBm = dBm - 256
```

und erhalten damit in unserem Fall den (korrekten) Wert -7. Ein Programm mit einer Ausgabe des Tx-Power-Levels finden Sie unter dem Namen `ble_scan_v0d.py`.

Ein Blick hinter die Kulissen: Negative Zahlen werden durch signed bytes dargestellt. Diese werden in der binärer Schreibweise durch eine `1` beim höchstwertigen Bit gekennzeichnet; positive Zahlen haben eine `0` als höchstwertiges Bit. So steht 11111110_2 z. B. für die Zahl -2. Um den Zusammenhang zwischen der Darstellung von negativen Zahlen im Binärsystem und im Zehnersystem zu verstehen, ist folgende Vorstellung nützlich: Der Kilometer-Zähler eines Autos stehe auf 4 km. Nun fährt das Auto einige Kilometer rückwärts. Beim Stand von 0 km kommt es schließlich beim Zähler zu einem "Unterlauf". Er springt auf 99999 km, dann auf 99998 km u.s.w. Genauso verhält es sich beim Binärsystem: Beim Unterlauf wird nach der Dualzahl 00000000_2 zunächst die Zahl 11111111_2, dann die Zahl 11111110_2 angezeigt. Die folgende Tabelle zeigt die Zuordnung von Zahlen im Zehner- und im Binärsystem:

dez.	127	...	2	1	0	-1	-2	...	-128
bin.	01111111_2	...	00000010_2	00000001_2	00000000_2	11111111_2	11111110_2	...	10000000_2

Allgemein erhält man die binäre Darstellung einer negativen Zahl aus dem Bitmuster der zugehörigen positiven Zahl durch Bildung des so genannten Zweierkomplements. Für die Zahl $6 = 00000110_2$ verfährt man dazu folgendermaßen: Zuerst bildet man das so genannte **Einerkomplement**, indem man in der Binärdarstellung Einsen und Nullen austauscht: 11111001_2. Das **Zweierkomplement** erhält man nun, indem man zu dieser Zahl (im Binärsystem) 1 addiert. Das ergibt: 11111010_2.

6 BLE-Konstanten

In den letzten Kapiteln haben wir es immer wieder mit Zahlen zu tun gehabt, welche bestimmte Eigenschaften kennzeichneten: Im Kapitel 3 hatten wir Werte kennen gelernt, die den Verbindungstyp kennzeichnen und weiterhin auch solche, die den Typ einer AD-Struktur angeben. Auch in Kapitel 5 sind uns solche Kennzahlen begegnet; hier bezeichneten sie den Event-Typ.

Auch in den folgenden Kapiteln werden uns immer wieder weitere derartige Kennzahlen begegnen. Es ist nicht einfach, sich diese Kennzahlen mit ihren Eigenschaften zu merken. Deswegen definiert man am Anfang des Programms alle benutzten Kennzahlen mit aussagekräftigen Abkürzungen, z. B.

```
ADV_TYP_NAME = 9
```

Diese Bezeichnung hatten wir schon in Kapitel 3 kennen gelernt. Die Bezeichnungen sind im Prinzip frei wählbar, aber es ist sinnvoll, diejenigen Bezeichnungen zu benutzen, welche in den micropython-Dokumentationen angegeben werden (vgl. [7]). Dort findet man z. B.

```
_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
_IRQ_GATTS_WRITE = const(3)
_IRQ_GATTS_READ_REQUEST = const(4)
_IRQ_SCAN_RESULT = const(5)
_IRQ_SCAN_DONE = const(6)
_IRQ_PERIPHERAL_CONNECT = const(7)
_IRQ_PERIPHERAL_DISCONNECT = const(8)
...
_ADV_TYP_FLAGS = const(1)
_ADV_TYP_NAME = const(9)
_TX_POWER_LEVEL = const(10)
...
```

Offensichtlich handelt es sich hier im ersten Block um Kennwerte für den Typ eines BLE-Interrupts; die beiden Kennwerte mit den Bezeichnern _IRQ_SCAN_RESULT (für den Wert 5) und _IRQ_SCAN_DONE (für den Wert 6) waren uns schon im Kapitel 5 begegnet. Im zweiten Block sind die Kennzahlen einiger AD-Struktur-Typen aufgelistet.

Auffällig ist hier, das rechts vom Gleichheitszeichen nicht einfach eine Zahl steht. Was hat es damit auf sich? Nun, wird der Variablen `x` ein Wert durch die Anweisung

```
x = 5
```

zugewiesen, dann wird der Compiler an jeder Stelle im Programm, an der `x` auftaucht, erst einmal nachschauen müssen, welcher Wert der Variablen x zugeordnet ist. Anders ist es bei der

Anweisung

```
x = const(5)
```

Hier wird direkt beim Erzeugen des Bytecodes (s. u.) jedes `x`, das im Quelltext auftaucht, durch die Zahl 5 ersetzt. Auf diese Weise wird der Bytecode kürzer und RAM eingespart. Das bedeutet aber auch, dass der Wert von `x` nicht veränderlich ist; man spricht hier deswegen von einer **Konstanten**.

Stellt man dem Bezeichner zusätzlich einen Unterstrich voran (z. B. `_x`), dann kann diese Konstante nur innerhalb dieses Programm-Moduls benutzt werden. Dafür wird der Bytecode allerdings nochmals kürzer.

Bevor die **const-Funktion** eingesetzt werden kann, muss sie aus dem `micropython`-Modul importiert werden:

```
from micropython import const
```

Bei den folgenden Kapiteln werden wir immer wieder von diesen Konstanten Gebrauch machen. Unser Programm `ble_scan_v0d.py` haben wir dementsprechend jetzt auch mit diesen Konstanten ausgestattet. Sie finden es unter dem Namen `ble_scan_v0e.py`. Zusätzlich haben wir einige kleinere Ergänzungen (mit entsprechenden Kommentaren) vorgenommen. Schauen Sie sich das Programm an und testen Sie es aus.

Ein Blick hinter die Kulissen: Micropython führt ein Programm in zwei Schritten aus: Zunächst wird der Quellcode in einen Zwischencode, den so genannten **Bytecode**, übersetzt. Dieser ist nun kompakter und schneller ausführbar als der Quellcode. Er ist von der realen Hardware unabhängig. Anschließend wird der Bytecode von der so genannten Micropython-Virtual-Machine interpretiert.

7 Passives und aktives Scannen

Bei dem bislang behandelten Scan-Vorgang erhält der Advertiser keine Rückmeldung darüber, ob sein Advertising-Datenpaket von einem Scanner empfangen worden ist. Diese Art des Scannens bezeichnet man als **passives Scannen** (vgl. Abb.1).

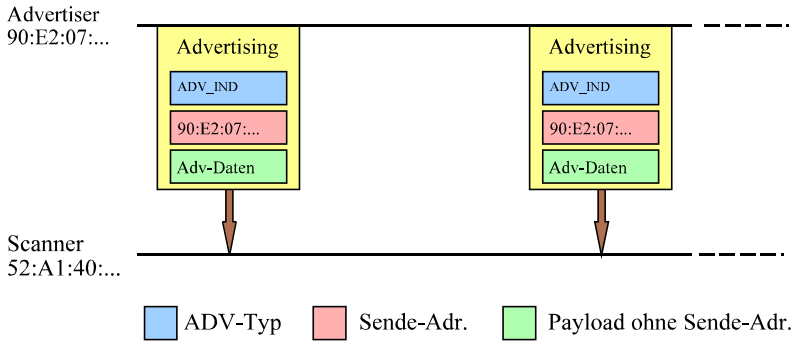


Abb. 1: Passives Scannen

Es besteht jedoch die Möglichkeit, dass der Scanner auf das Advertising-Datenpaket reagieren kann, indem er nun selbst ein sogenanntes **Request**-Datenpaket an den Advertiser sendet. Dies setzt voraus, dass der Advertiser solche Kontaktaufnahmen zulässt. In diesem Fall bezeichnet man den Advertiser als **scannbar** (**scannable**); er macht dies mit dem ADV-Typ **0** (**ADV_IND**) oder **6** (**ADV_SCAN_IND**) kenntlich.

Erhält nun ein Scanner ein Datenpaket mit einem solchen ADV-TYP, dann kann er einen Request senden. Dieser Request hat dieselbe Struktur wie ein Advertiser-Datenpaket: Sein ADV-Typ ist **SCAN_REQ** mit dem Wert 3; in seinem Payload gibt er die Adresse des Advertisers an, an den sich der Request richtet. Der Advertiser sendet nun als Response ein weiteres Datenpaket mit zusätzlichen Informationen; dieser hat den ADV-Typ **SCAN_RSP** mit dem Wert 4. Einen solchen Vorgang bezeichnet man als **aktives Scannen** (Abb. 2).

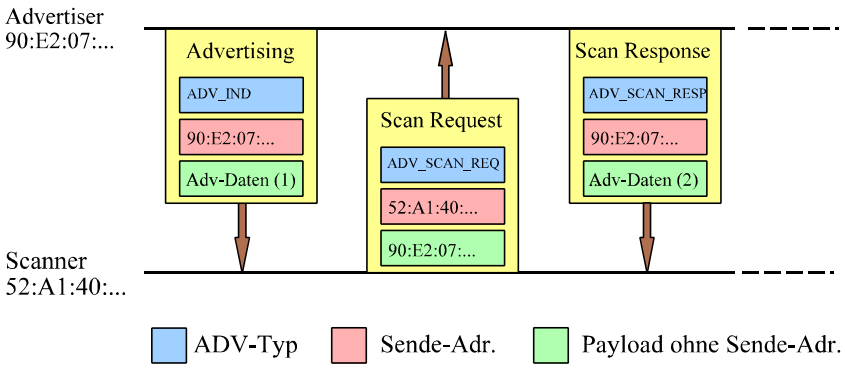


Abb. 2: Aktives Scannen

Einerseits hat damit der Advertiser erfahren, dass sein Datenpaket empfangen worden ist; und er kennt jetzt auch die Adresse des Gerätes, welches sein Datenpaket empfangen hat. Andererseits erhält der Scanner weitere Informationen vom Advertiser.

Standardmäßig wird durch die Methode `ble.gap_scan(5000, 100000, 25000)` aus Kapitel 5 nur ein passives Scannen durchgeführt. Diese Methode besitzt aber optional einen vierten Parameter ***active***. Geben wir ihm den Wert `True`, wird der Aktiv-Modus eingestellt; erhält er den Wert `False` (oder lassen wir ihn weg, was wir bislang gemacht haben) dann wird der Passiv-Modus benutzt. Ein Programm mit aktivem Scannen finden Sie in der Datei `ble_scan_v1a.py`.

Bislang habe ich in meinem Umfeld nur einen einzigen Advertiser gefunden, von welchem mein TTGO einen Scan Response erhalten hat. Hierbei handelt es sich um den ***technic hub*** eines Lego-Fahrzeuges, welches sich per BLE von einem Handy aus steuern lässt. Interessant ist, dass der Name dieses Advertisers erst im Response zu finden ist. Offensichtlich waren andere Daten für die Entwickler der Fernsteuerung von höherer Dringlichkeit.

```
------------------------------------------------------------
packet: 3
Address-Type:  0
Address:  ['0x90', '0x84', '0x2b', '0x51', '0x25', '0xf9']
ADV-Type:  0
RSSI in dBm:  -58
ADV-Data:  ['0x2', '0x1', '0x6', '0x11', '0x7', '0x23', '0xd1', '0xbc',
'0xea', '0x5f', '0x78', '0x23', '0x16', '0xde', '0xef', '0x12', '0x12',
'0x23', '0x16', '0x0', '0x0', '0x9', '0xff', '0x97', '0x3', '0x0', '0x80',
'0x6', '0x0', '0x41', '0x0']
ADV-Data (Name):  None
ADV_Data (TX_Power in dBm):  None

------------------------------------------------------------
packet: 4
Address-Type:  0
Address:  ['0x90', '0x84', '0x2b', '0x51', '0x25', '0xf9']
ADV-Type:  4
RSSI in dBm:  -58
ADV-Data: ['0x5', '0x12', '0x10', '0x0', '0x20', '0x0', '0x2', '0xa', '0x0',
'0xc', '0x9', '0x54', '0x65', '0x63', '0x68', '0x6e', '0x69', '0x63', '0x20',
'0x48', '0x75', '0x62']
ADV-Data (Name):  Technic Hub
ADV_Data (TX_Power in dBm):  0
```

Die Bluetooth-Adressen zeigen, dass die beiden Datenpakete Nr. 3 und 4 tatsächlich von demselben Advertiser stammen. Bei dem Paket Nr. 4 handelt es sich um einen Response, weil der ADV-Typ den Wert 4 besitzt.

8 Broadcaster und Observer programmieren

In Kapitel 1 hatten wir schon die Rollen eines Broadcasters und eines Observers kennen gelernt: Der Broadcaster sendet über seine Advertising-Datenpakete mit seinem Payload eine Message (Botschaft) aus. Diese bildet neben der Bluetooth-Adresse und den bereits bekannten AD-Strukturen von den Typen `ADV_TYP_FLAGS` und `ADV_TYP_NAME` noch eine weitere AD-Struktur. Wir geben ihr in diesem Kapitel die Typ-Bezeichnung `ADV_TYP_MSG` mit dem Wert 250.

Solche Advertising-Datenpakete können nun von sämtlichen Observern in der Nähe des Broadcasters empfangen werden. Ähnlich wie wir den Namen aus dem Payload gefiltert haben, können wir auch bei den Botschaften vorgehen: Wir brauchen dazu unser bekanntes Scan-Programm nur geringfügig abändern: Einige überflüssige Daten werden nicht ausgewertet bzw. ausgegeben, dafür fügen wir eine neue Funktion `find_adv_msg(data)` hinzu, mit deren Hilfe wir die Message aus dem Payload herausfiltern. Neue Befehle sind dafür nicht erforderlich. Ein fertiges Programm findet man unter dem Namen `ble_observe_0.py`.

Anders verhält es sich beim Programm für den Broadcaster. Bislang haben wir nämlich noch nicht dargelegt, wie Advertising-Datenpakete mit Micropython ausgesendet werden. Das holen wir jetzt nach.

Wie üblich definieren wir zunächst eine Instanz der BLE-Klasse und aktivieren sie:

```
ble = ubluetooth.BLE()
ble.active(True)
```

Um Advertising-Datenpakete für das Broadcasting auszusenden, benutzt man die Methode

```
ble.gap_advertise(interval_us, adv_data, connectable=False)
```

Dabei gibt der Parameter `intervall_us` das Advertising-Intervall in Mikrosekunden an (mehr dazu s. u.). Beim Broadcasting soll dieses Intervall größer als 100000 us sein. Der Parameter `adv_data` stellt den Payload (ohne die Bluetooth-Adresse) dar. Hat der letzte Parameter den Wert `False`, dann erhält `ADV-Type` den Wert 2, ansonsten den Wert 0 (vgl. Kap. 3).

Wir werden hier die folgenden AD-Strukturen benutzen:

1. Die Flags-AD-Struktur, bestehend aus den Bytes 0x02, 0x01 und 0x06. Dies kennzeichnet die Eigenschaften “LE General Discoverable Mode” und “BR/EDR Not Supported”, vgl. [8].

2. Die hinlänglich bekannte Namens-AD-Struktur

3. Unsere neue Message-AD-Struktur

Schritt für Schritt setzen wir nun das Advertising-Datenpaket zusammen:

```
adv_data = b'\x02\x01\x06'
name = bytes(name, 'UTF-8')
adv_data = adv_data +  bytearray((len(name) + 1, _ADV_TYP_NAME)) + name
msg = bytes(msg, 'UTF-8')
adv_data = adv_data + bytearray((len(msg)+1, _ADV_TYP_MSG)) + msg
```

Die Variable `name` sollte zuvor den Namen des Broadcasters erhalten, z. B.:

```
name = 'ESP32'
```

und der Variablen `msg` sollte die Botschaft zugeordnet worden sein, z. B. durch eine Tastatureingabe im Terminal:

```
msg = input('Message: ')
```

Wenn dann als Message die Zeichenkette 'hallo' verwendet wird, ist der Variable adv_data der Bytestring

```
b'\x02\x01\x06\x06\tESP32\x06\xffhallo'
```

zugeordnet. Dabei steht `\t` für das Tabulator-Steuerzeichen mit dem Code 9.

Das gesamte Programm `ble_broadcast_0.py` sieht dann so aus:

```
from time import sleep
import ubluetooth
from micropython import const

_ADV_TYP_NAME = const(9)
_ADV_TYP_MSG = const(250) # Eigener ADV_Typ für Messages

def advertiser(name, msg):
  adv_data = b'\x02\x01\x06'
  name = bytes(name, 'UTF-8')
  adv_data = adv_data +  bytearray((len(name) + 1, _ADV_TYP_NAME)) + name
  msg = bytes(msg, 'UTF-8')
  adv_data = adv_data + bytearray((len(msg)+1, _ADV_TYP_MSG)) + msg
  ble.gap_advertise(200000, adv_data, connectable=False)

# main:
ble = ubluetooth.BLE()
name = 'ESP32'
ble.active(True)
```

```
while True:
    msg = input('Eingabe: ')
    advertiser(name, msg)
    sleep(0.1)
```

Wenn wir den Broadcaster und den Observer im Zusammenspiel austesten wollen, benötigen wir zwei ESP32-Module. Wir starten nun den Broadcaster und den Observer; letzterer ist so programmiert, dass er für 30 Sekunden einen Scan-Vorgang durchführt und nur diejenigen Ergebnisse anzeigt, die den Advertiser-Namen 'ESP32' besitzen.

Beim Broadcaster sieht unser Eingabe-Protokoll so aus:

```
>>> %Run -c $EDITOR_CONTENT
Eingabe: Test:
Eingabe: Hallo Welt!
```

Beim Observer erscheint auf dem Terminal dann:

```
...

------------------------------------------------------------
ADV-Data (Name):  ESP32
ADV_Data (Text):  Test:

------------------------------------------------------------
ADV-Data (Name):  ESP32
ADV_Data (Text):  Test:

------------------------------------------------------------
ADV-Data (Name):  ESP32
ADV_Data (Text):  Hallo Welt!

...
```

Mit dieser Vorgehensweise können wir auch Messdaten von einem TTGO-Modul auf ein anderes übertragen und dort die übertragenen Werte auf dem Display anzeigen lassen. Mit den Programmen `ble_broadcast_1.py` und `ble_observe_1.py` können z. B. Temperaturwerte von einem Modul außerhalb des Hauses aufgenommen und von dort an ein Modul innerhalb des Hauses gesendet werden. Zur Ermittlung des Temperaturwertes kommt hier ein BME280-Baustein zum Einsatz. Beachten Sie dabei, dass das zum Betreiben dieses Bausteins benutzte Modul `Bme280.py` in den Speicher des ESP32 geladen sein muss.

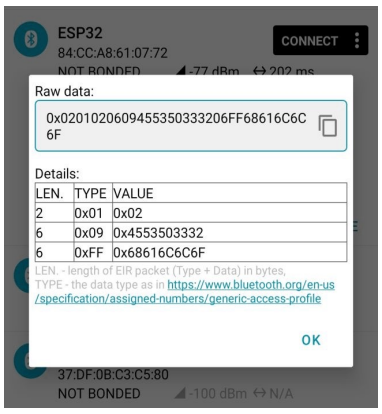
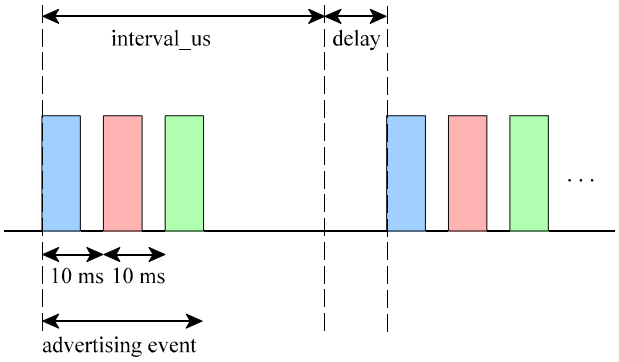


Abb. 1

Wer kein zweites TTGO-Modul zur Hand hat, der kann als Ersatz für den Observer auch den Scanner der Handy-App *nRF Connect* benutzen. Nach dem Start beginnt die App automatisch mit einem Scan-Vorgang. Berühren Sie nun den ESP32-Eintrag und wählen Sie dort die Option *RAW*. Sie erhalten dann eine Auflistung der mit dem Payload empfangenen AD-Strukturen (vgl. Abb. 1). In der letzten Zeile finden Sie die Message 'hallo' als Folge von 5 Bytes: 68 61 6c 6c 6f; der ADV-Typ hatte hier allerdings den Wert 255 = 0xff. Übrigens: Wer versucht, mit der App über den Button CONNECT eine Verbindung aufzubauen, wird feststellen, dass dies (erwartungsgemäß) nicht gelingt.

Ein Blick hinter die Kulissen: Im Advertising-Zustand sendet ein Bluetooth-Gerät in bestimmten Zeitabständen Advertising-Datenpakete auf den Kanälen 37, 38 und 39 aus. Dabei springt es innerhalb von 10 ms von einem Kanal zum nächsten (vgl. die folgende Abbildung). Eine Folge von drei solchen Datenpaket-Sendungen bezeichnet man als **Advertising Event**.



Diese Advertising Events werden in gewissen Zeitintervallen wiederholt. Im Wesentlichen entsprechen diese Zeitintervallen gerade der Zeitangabe, welche in dem Parameter `interval_us` der `advertise`-Methode angegeben wird (s. o.). Der Wert von `interval_us` sollte i. A. zwischen 20 ms und 10,24 s liegen und ein Vielfaches von 0,625 ms sein - andernfalls wird er automatisch entsprechend abgerundet. An dieses Zeitintervall schließt sich automatisch ein kurzes Zeitintervall `delay` an; die Länge dieser Verzögerung wird durch einen Zufallswert zwischen 0 und 10 ms festgelegt. Damit sollen Kollisionen zwischen den Datenpaketen benachbarter Geräte vermieden werden.

Den Abstand zwischen zwei Advertising Events können wir übrigens mit der Handy-App *nRF Connect* kontrollieren. Der gemessene Wert bestätigt die obigen Darlegungen.

Auch beim Herstellen einer Verbindung zwischen zwei Geräten kommen solche Advertising Events zum Tragen. Ist die Verbindung allerdings einmal hergestellt, dann findet der Daten-Austausch über die Kanäle von 0 bis 36 statt.

9 Connect & Disconnect

In Kapitel 2 hatten wir schon unser TTGO-Modul als Peripheral eingesetzt. Als Central diente uns die Handy-App nRF Connect. Mit der Scan-Funktion hatte diese App die Advertising-Pakete unseres TTGO-Moduls empfangen und einige zugehörige Daten angezeigt . Darüber hinaus hatten wir auch schon eine Verbindung (Connection) zwischen dem Handy und dem TTGO hergestellt. Dadurch war das Handy zum Master und der TTGO zum Slave geworden.

In diesem Kapitel wollen wir uns dies nun genauer anschauen. Dabei wollen wir insbesondere folgenden Fragen nachgehen:

- Welche Möglichkeiten bietet die Verbindung im Vergleich zum Broadcasting?
- Welche Prozesse finden beim Aufbau einer Verbindung und danach statt?
- Welche Befehle stellt die BLE-Klasse im Zusammenhang mit dem Verbindungsaufbau zur Verfügung und wie geht man damit um?

Dabei werden wir uns auf den oben genannten Fall beschränken, dass das Handy als Central bzw. Master und der TTGO als Peripheral eingesetzt wird.

Die folgende Tabelle vergleicht einige Aspekte der Kommunikation beim Broadcasting und im Connection-Modus:

Broadcasting	Connection-Modus
Nur wenige Bytes können übertragen werden.	Auch größere und komplexere Datenstrukturen können übertragen werden.
Die Übertragung der Daten ist unidirektional, Daten werden nur vom Broadcaster (TTGO) zum Observer (Handy) übertragen.	Die Übertragung der Daten ist bidirektional.
Die Übertragung der Daten geht vom Broadcaster (TTGO) zu *allen* Observern (Handys) in der Umgebung.	Die Übertragung der Daten findet *nur zwischen den beiden verbundenen* Geräten statt.
Die Übertragung geht vom Broadcaster (TTGO) aus; der Observer (Handy) hat darauf keinen Einfluss.	Der Master (Handy) steuert die Verbindung: er sendet dem Slave (TTGO) Daten oder fordert Daten vom Slave an.

Wie wird nun eine solche Verbindung aufgebaut? Abb. 1 zeigt uns den zeitlichen Ablauf (von oben nach unten): Der TTGO sendet als Peripheral zunächst Advertising-Datenpakte auf den Kanälen 37, 38 und 39 aus; hier ist beispielhaft nur ein einziges Advertising Event vollständig dargestellt (vgl. Kap. 8). Nun leiten wir mit der Handy-App mit dem CONNECT-Button eine Verbindung mit dem TTGO-Modul ein (vgl. Abb. 1 von Kap. 2). Dann sendet das Handy ein spezielles Datenpaket aus, einen so genannten **Connection Request**. Dabei muss dieser Request kurz nach dem Empfang eines Advertising-Datenpaket ausgesendet werden - und zwar auf dem Kanal, den das letzte Advertiser-Paket benutzt hat.

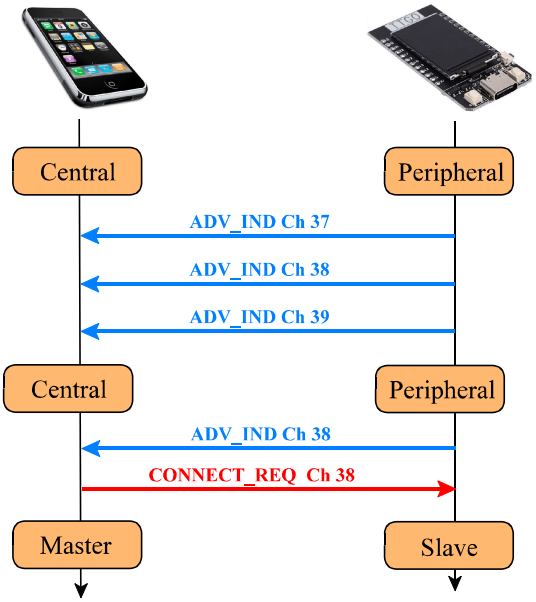


Abb. 1

Bei dem Connection-Request handelt es sich *nicht* um ein Advertiser-Datenpaket; dies wird durch einen entsprechenden Wert beim PDU-Type (CONNECT_IND) kenntlich gemacht. Im Connection Request befinden sich sowohl die Adresse des Handys (allgemein: des Centrals) als Initiator der Verbindung als auch die Adresse des TTGO (allgemein: des Peripherals). Nur an diesen Peripheral richtet sich dieses Datenpaket!

Der Peripheral horcht nach dem Aussenden eines jeden Advertising-Datenpakets für eine kurze Zeitspanne nach einem solchen Request. Empfängt er einen solchen Request, so ist die Verbindung zustande gekommen; der Peripheral ist nun ein Slave und sendet keine Advertising-Pakete mehr.

Nach dem Connection Request übermittelt unser Handy - nun zum Master geworden - auch Parameter, die für die weitere Kommunikation wichtig sind. Hier möchte ich nur einen einzigen davon erwähnen, das **Connection Interval**. Was hat es nun mit diesem Parameter auf sich?

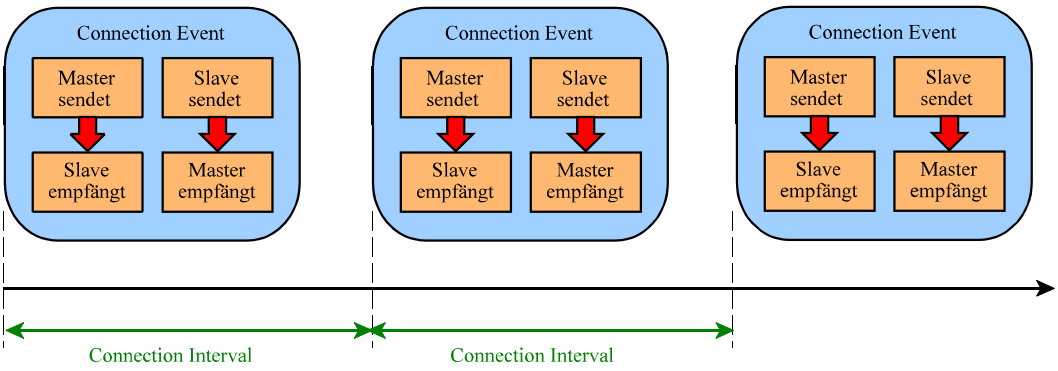


Abb. 2

Master und Slave bleiben nur so lange verbunden, wie sie Daten austauschen. Diesen Austausch bezeichnet man als **Connection Event**. Ein solches Ereignis beginnt damit, dass der Master ein Datenpaket sendet und der Slave dieses empfängt; anschließend sendet der Slave ein Datenpaket und der Master empfängt es (vgl. Abb.2); innerhalb eines solchen Ereignisses können auch mehrere Datenpakete ausgetauscht werden. Der nach dem Verbindungsaufbau übermittelte Parameter *Connection Interval* legt fest, in welchen Zeitabständen ein solches Connection Event stattfindet; gegebenenfalls werden auch leere Datenpakete ausgetauscht. In der Zwischenzeit können Sender bzw. Empfänger ausgeschaltet werden. Dies ist ein Beitrag zur Energieeffizienz: Höhere Werte für das Connection Interval bedeuten eine entsprechend größere Energieeinsparung; diese geschieht allerdings zu Lasten der Datenübertragungsrate. Für das Connection Interval sind Werte zwischen 7,5 ms und 4.0 Sekunden möglich. (Interessant ist: Der Slave kann einen Gegenvorschlag machen; den muss der Master aber nicht annehmen.)

Was geschieht, wenn ein Connection Event nicht rechtzeitig stattfindet? Dann wird die Verbindung automatisch unterbrochen. Dies ist z. B. der Fall, wenn Master oder Slave ausgeschaltet werden.

Programmieren

Nun wollen wir unser TTGO-Modul so programmieren, dass es eine Bluetooth-Verbindung mit der Handy-App nRF Connect aufnehmen kann. Zunächst muss der TTGO zum Advertiser werden. Wie dies geht, haben wir im Prinzip schon im Kapitel 8 über das Broadcasting gelernt. Hier sieht die Funktion so aus:

```
def advertiser(name):
  adv_data = b'\x02\x01\x06'
  name = bytes(name, 'UTF-8')
  adv_data = adv_data +  bytearray((len(name) + 1, _ADV_TYP_NAME)) + name
  ble.gap_advertise(200000, adv_data, connectable=True)
```

Ganz wesentlich ist hier: Im Gegensatz zum Broadcaster-Programm hat hier der 4. Parameter von `ble_gap_advertise` nun den Wert `True`. Damit erhält der PDU-Type im Header des Advertiser-Datenpakets den Wert `ADV_IND`, für den Central ein Zeichen dafür, dass eine Verbindung hergestellt werden kann.

Empfängt nun der TTGO einen Connection Request vom Handy, führt dies zu einem Interrupt bei der Instanz `ble`; der erste Parameter (`event`) des Event Handlers erhält dabei den Wert 1 (`IRQ_CENTRAL_CONNECT`).

Unser Event Handler sieht in einer ersten Version so aus:

```
def my_ble_irq(event, data):
  print('event: ', event)
  if event == _IRQ_CENTRAL_CONNECT:
```

```
    global conn
    conn = True
    print('Central connected\n')
```

Über die globale Variable `conn` können wir im Hauptprogramm den aktuellen Status mit Hilfe einer LED an Pin 27 jederzeit sichtbar machen; das gesamte Hauptprogramm sieht so aus:

```
from machine import Pin
from time import sleep
import ubluetooth
from micropython import const

_ADV_TYP_NAME = const(9)
_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)

led1 = Pin(27, Pin.OUT) # zeigt den Connect-Status an
conn = False

ble = ubluetooth.BLE()
name = 'ESP32'
ble.active(True)
ble.irq(my_ble_irq)
advertiser(name)

while True:
  led1.value(conn)
```

Erhält `ble` einen Interrupt mit dem `event`-Paramter `2 (IRQ_CENTRAL_DISCONNECT)`, so ist die Verbindung getrennt worden. Für diesen Fall erweitern wir unseren Event Handler (Version 2):

```
def my_ble_irq(event, data):
  global conn
  if event == _IRQ_CENTRAL_CONNECT:
    conn = True
    print('Central connected\n')
  elif event == _IRQ_CENTRAL_DISCONNECT:
    conn = False
    print('Central disconnected\n')
    global name
    advertiser(name)
```

Ganz wichtig ist die letzte Zeile: Nach dem Trennen der Verbindung geht der TTGO nicht automatisch wieder in den Advertising-Zustand. Wenn wir wollen, dass wir über das Handy nach einer Trennung erneut eine Verbindung mit dem TTGO herstellen können, müssen wir diesen Advertising-Zustand mit der `advertiser`-Methode wieder aktivieren.

Das vollständige Programm (ergänzt um einige Programm-Zeilen, welche den Inhalt vom `data`-Parameter des Event Handlers auf dem Terminal ausgeben) finden Sie in der Datei `ble_adv_con_0.py`. Testen Sie es einmal aus! (Übrigens: Das hier vorgestellte Programm stellt eine vereinfachte Fassung des Programms `experiment_1.py` aus Kap. 2 studiert dar.)

Bei Ihren Experimenten mit dem Programm `ble_adv_con_0.py` ist Ihnen sicherlich schon aufgefallen, dass unmittelbar nach dem Verbinden zwei weitere Ereignisse angezeigt werden. In beiden Fällen hat der Parameter `event` den Wert 27. Ein Blick in die `ubluetooth`-Dokumentation zeigt:

- Dieser Wert weist auf ein Connection-Update hinweist.
- Dabei teilt der Master dem Slave u. A. das Connection Interval mit.

Indem wir im Request Handler die Zeilen

```
elif event == _IRQ_CONNECTION_UPDATE:
  print('connection updated')
  conn_handle, conn_interval, conn_latency, supervision_timeout, status = data
  print('connection interval: ', conn_interval, '\n')
```

ergänzen, können wir uns nun im Terminal auch anschauen, welchen Wert der Master als *Connection Interval* an den Slave sendet. Natürlich dürfen wir nicht vergessen, auch die Konstante _IRQ_CONNECTION_UPDATE am Anfang des Programms zu definieren:

```
_IRQ_CONNECTION_UPDATE = const(27)
```

Wir starten das so erweiterte Programm (`ble_adv_con_0a.py`). Nach dem Verbinden erscheint nun auf dem Terminal:

```
Central connected

event:  27
connection updated
connection interval:  6

event:  27
connection updated
connection interval:  36
```

Auch der Slave kann eine Verbindung trennen. Dies geschieht mit der Methode `ble.gap_disconnect(handle)`; dabei muss hier als Parameter der beim Verbindungsaufbau erhaltene Connection Handler `conn_handle` eingesetzt werden. In der Datei `ble_adv_con_0b.py` finden Sie ein Programm, bei welchem Sie mit dem Taster S0 des TTGO eine Verbindung trennen können.

10 Eine eigene BLE-Klasse

Im letzten Kapitel hatten wir bereits mit der `BLE`-Klasse des `ubluetooth`-Moduls gearbeitet. Wenn wir in diesem Kapitel eine "eigene" `BLE`-Klasse vorstellen, dann wollen wir das Rad nicht neu erfinden. Vielmehr wollen wir bei der Definition unserer eigenen Klasse auf die bereits vorhandene Klasse zurückgreifen und diese neue Klasse mit zusätzlichen Eigenschaften und Methoden ausstatten. Unsere neue Klasse soll die Bezeichnung `MyBLEServer` erhalten. Sie wird unsere Programme in den nächsten Kapiteln einfacher und übersichtlicher machen. Darüber hinaus werden wir auf diesem Wege auch auf globale Variablen, wie sie noch im letzten Kapitel benutzt worden sind, verzichten können. Wie auch bei der `BLE`-KLasse des `ubluetooth`-Moduls kann aber auch von der `MyBLEServer`-Klasse jeweils nur eine einzige Instanz erzeugt werden.

Bevor wir mit der Definition unserer Klasse `MyBLEServer` beginnen, sollen die nötigen Importe vorgenommen und einige Konstanten definiert werden:

```
import ubluetooth
from micropython import const
_ADV_TYP_NAME = const(9)
_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
```

Unsere neue Klasse `MyBLEServer` definieren wir folgendermaßen (Einige Erläuterungen habe ich unterhalb angefügt; wem diese nicht ausreichen, sei auf [2] und [3] verwiesen.):

```
class MyBLEServer():                            # Erl. 1

    def __init__(self, adv_name):               # Erl. 2

        self.name = adv_name                    # Erl. 3
        self.ble = ubluetooth.BLE()             # Erl. 4
        self.ble.active(True)
        self.ble.irq(self.my_ble_irq)           # Erl. 5
        self.advertiser()                       # Erl. 6
        self.conn = False                       # Erl. 7


    def my_ble_irq(self, event, data):          # Erl. 8

        if event == _IRQ_CENTRAL_CONNECT:
            print('Central connected')
            self.conn = True                    # Erl. 9
```

```
        elif event == _IRQ_CENTRAL_DISCONNECT:
            print('Central disconnected')
            self.conn = False
            self.advertiser()


    def advertiser(self):

        adv_data = b'\x02\x01\x06'
        name = bytes(self.name, 'UTF-8')
        adv_data = adv_data+bytearray((len(name)+1, _ADV_TYP_NAME))+name
        self.ble.gap_advertise(200000, adv_data, connectable=True)
        print('advertising')
```

Erl. 1: Eine Klassendefinition wird mit dem Schlüsselwort `class` eingeleitet. Direkt dahinter steht der Klassenname. Darunter werden dann die Methoden dieser Klasse definiert. Diese Definitionen müssen eingerückt werden, um ihre Zugehörigkeit zur Klasse deutlich zu machen.

Erl. 2: `__init__` ist standardmäßig der Name derjenigen Methode, die automatisch bei der Instanziierung der Klasse ausgeführt wird. Allgemein werden **Methoden** wie Funktionen definiert. Beim Aufruf einer Methode einer Instanz wird diese Instanz automatisch als erster Parameter übergeben; deshalb wird dieser erste Parameter einer Methode konventionell `self` genannt. Damit kann dann die Methode auf die Eigenschaften *dieser* Instanz zugreifen. Die nun folgenden Parameter dienen dazu, bei der Instanziierung schon einzelne Eigenschaften des Objekts festzulegen. Mehr dazu in Erl. 3.

Erl. 3: Die mit dem Parameter `adv_name` übergebene Zeichenkette wird in der **Instanz-Variablen** `name` gespeichert. Innerhalb einer Klassendefinition versehen wir alle Instanz-Variablen und Methoden dieser Klasse mit dem Selbst-Verweis `self`, in diesem Fall:

```
self.name = adv_name
```

Dadurch beziehen sie sich auf das instanziierte Objekt. (Man beachte, dass im Prinzip von einer Klasse ja mehrere Instanzen mit unterschiedlichen Eigenschaften erzeugt werden können.) Für unseren Fall bedeutet das: Nach der Instanziierung `b = MyBLEServer('Test')` wird durch `print(b.name)` das Wort *Test* ausgegeben.

Erl. 4: Hier wird eine Instanz der Klasse `ubluetooth.BLE erzeugt` und der Instanz-Variablen `self.ble` unserer neuen Klasse `MyBLEServer` zugeordnet. `self.ble` hat damit alle Eigenschaften und Methoden eines `ubluetooth.BLE-Objekts`. Deswegen kann z. B. in der folgenden Programmzeile mit `self.ble.active(True)` das Bluetooth-Modul des TTGO aktiviert werden.

Erl. 5: In dieser Zeile wird `self.ble` ein Event-Handler zugewiesen. Bei dem Event Handler handelt es sich um eine Methode unserer neuen `MyBLEServer`-Klasse (vgl. Erl. 8). Der Event Handler kann deshalb auf alle Eigenschaften unserer Klasse zugreifen (vgl. Erl. 9).

Erl. 6: Schon bei der Instanzzierung wird automatisch der Advertiser gestartet.

Erl. 7: Nach der Instanziierung hat die Eigenschaft `conn` den Wert `False`.

Erl. 8: Hier beginnt die Definition unseres Event Handlers.

Erl. 9: Bei einem erfolgreichen Verbindungsaufbau wird die `Eigenschaft conn` auf `True` gesetzt.

Das Hauptprogramm ist jetzt sehr einfach; es besteht lediglich aus einer einzigen Zeile, der Instanziierung:

```
ble = MyBLEServer('ESP32')
```

Im Rahmen dieser Instanziierung werden alle in der `__init__`-Methode aufgeführten Programmzeilen ausgeführt. Insbesondere wird der Advertiser gestartet und der Interrupt aktiviert. Somit wird nach der Instanziierung auf dem Terminal `advertising` angezeigt und vom Scanner unserer App nRF Connect das Gerät *ESP32* angezeigt. Wenn wir dann noch auf dem Handy den Button CONNECT betätigen, wird eine Verbindung aufgebaut und dazu eine Meldung auf dem Terminal ausgegeben. Entsprechendes gilt, wenn die Verbindung gelöst wird. Da der Interrupt aktiviert bleibt, kann die Verbindung beliebig oft hinter einander hergestellt und gelöst werden.

Das gesamte Programm finden Sie in der Datei `my_BLE.py`.

Jetzt wollen wir noch zeigen wie man den aktuellen Connect-Zustand zusätzlich mit einer LED anzeigen lassen kann. Anders als in Kap. 9 greifen wir jetzt nicht mehr auf eine globale Variable Variable zurück, sondern auf die `conn`-Eigenschaft unserer MyBLEServer-Instanz `ble`. Das Hauptprogramm sieht dann so aus:

```
from machine import Pin
led = Pin(27, Pin.OUT)
ble = MyBLEServer('ESP32')
while True:
    led.value(ble.conn)
```

Das vollständige Programm finden Sie unter dem Dateinamen `my_BLE_LED.py`.

11 Der Heart Rate Service

Wie man unser TTGO-Modul (d. h. den Slave) programmiert, damit eine Verbindung aufgebaut werden kann, haben wir schon in den letzten beiden Kapiteln gelernt. Wie werden nun **im Rahmen dieser Verbindung** zwischen dem TTGO-Modul und dem Handy Daten ausgetauscht? Dieser Frage widmen wir uns jetzt.

Zunächst aber ein Hinweis zur Bezeichnung: Im Zusammenhang mit dem Speichern von Daten und dem Zugriff darauf spricht man nicht mehr von Slave und Master, sondern von **Server** und **Client**. Damit wird deutlich gemacht, dass die Geräte jetzt eine andere Rolle einnehmen.

Beispiel für Gerät	Herstellen oder Lösen einer Verbindung	Datenspeicherung und -Zugriff
Handy	Master	Client
TTGO	Slave	Server

In diesem Kapitel wollen wir mit einem einfachen Beispiel beginnen: Unser TTGO soll Herzschlag-Pulsraten an unser Handy übertragen. Dabei werden wir nicht mit echten Messwerten für die Pulsrate arbeiten. Vielmehr werden wir diese zunächst über das Terminal eingeben; später werden wir sie dann durch Zufallszahlen simulieren.

Die zu übertragenden Daten müssen nun so strukturiert sein, dass Client und Server jeweils nicht nur über die Daten selbst, sondern auch über ihre Bedeutung sowie über die Zugriffsrechte (mehr dazu s. u.) informiert sind. Diese Strukturierung geschieht über sogenannte **Services** (Dienste). Eine Reihe von Services sind bereits von der Bluetooth **Special Interest Group** (**SIG**) vordefiniert worden. Dazu gehört auch der **Heart Rate Service**. Dieser besitzt eine sehr einfache Struktur; deswegen wird er sehr gerne als Einstieg in dieses Thema benutzt. Natürlich kann man derartige Services auch selbst erstellen; im Rahmen dieser Einführung werden wir (fast) nur mit vordefinierten Services arbeiten. Wir werden diese Dienste eingehend erklären, sie an der einen oder anderen Stelle modifizieren und die Folgen dieser Änderungen anschauen.

Der Heart Rate Service der SIG stellt neben dem Pulsraten-Wert noch weitere Werte (mitsamt den zugehörigen Informationen) zur Verfügung. Jeden solchen Wert eines Service bezeichnet man als eine **Characteristic** dieses Service. Einen Service kann man sich als einen Schrank vorstellen, dessen Schubladen dann den Characteristics entsprechen. Diese Schubladen besitzen ein Schloss; je nach Zugriffsrechten können sie vom Client (Handy) geöffnet werden oder nicht.

Auf unserem TTGO und unserem Handy können gleichzeitig verschiedene Services vorliegen. Diese bilden dann das so genannte **GATT-Profil** (**G**eneric **Att**ribute Profile; Services und Characteristics bezeichnet man auch allgemein als **Attribute**; Genaueres dazu in Kap. 13). Im Rahmen unseres Modells entspricht diesem GATT-Profil dann der Raum, in welchem sich die Schränke befinden.

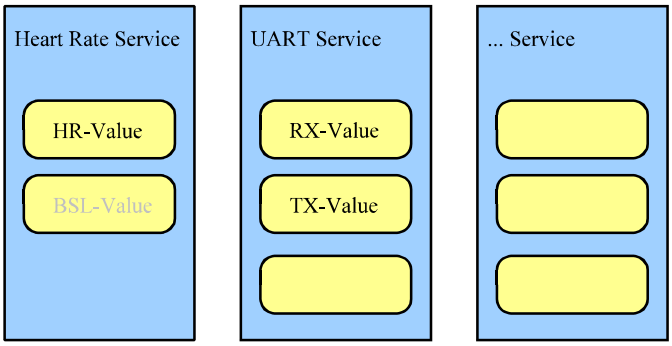


Abb. 1

Für alle Services und Characteristics gibt es einen **UUID** (Universally Unique Identifier), also eine Zahl, welche zur (eindeutigen) Identifikation eingesetzt wird. Im Allgemeinen sind dies 128-Bit-Zahlen; der in diesem Kapitel von uns benutzte vordefinierte Service und seine Characteristic besitzen UUIDs mit 16 Bit.

Zur Programmierung unseres (einfachen) GATT-Profils benötigen wir nur die folgenden Angaben:

Attribut	UUID	Zugriffsmöglichkeit (Permission)
Service (Heart Rate)	0x180D	
Characteristic (Value)	0x2A37	FLAG_READ (s. u.)

Für unser Programm benutzen wir die `MyBLEServer`-Klasse, so wie wir sie im Kap. 10 vorgestellt haben. Diese muss nun ergänzt werden. Zunächst fügen wir bei der Methode `__init__` die Zeile

```
self.register()
```

ein; diese wird bei der Instanziierung dann automatisch ausgeführt. Wir definieren diese Methode folgendermaßen:

```
def register(self):
  HR_UUID = ubluetooth.UUID(0x180D)
  HR_CHAR = (ubluetooth.UUID(0x2A37), _FLAG_READ)
  HR_SERVICE = (HR_UUID, (HR_CHAR,))
  SERVICES = (HR_SERVICE, )
  ((self.hr,),) = self.ble.gatts_register_services(SERVICES)
```

Die Bedeutung der einzelnen Zeilen erklären wir jetzt der Reihe nach:

In der ersten Zeile wird der Variablen `HR_UUID` lediglich eine Instanz der Klasse `UUID` zum Wert `0x180D` erzeugt und der Variablen `HR_UUID` zugewiesen.

In der nächsten Zeile wird der Variablen `HR_CHAR` ein Tupel zugewiesen, welches aus zwei Elementen besteht: dem UUID für die Value-Characteristic und dem zugehörigen Permission-Flag. (Für dieses Flag müssen wir die Liste der bereits eingeführten Konstanten noch um `_FLAG_READ = const(2)` erweitern!) Die Variable `HR_CHAR` entspricht einer Schublade.

In der dritten Zeile wird der Heart Rate Service beschrieben. Dieser wird durch ein Tupel gebildet, beim dem das erste Element den Service-UUID beinhaltet; damit ist quasi der Schrank gekennzeichnet. In dem darauf folgenden Tupel werden alle Schubladen (Caracteristics) aufgelistet; da wir lediglich ein einziges Characteristic haben, lautet dieses Tupel `(HR_CHAR, )`. An dieser Stelle ist das Komma sehr wichtig: Ohne dieses Komma würde dieser Ausdruck nämlich nicht als Liste, sondern als Term gedeutet (dessen Wert einfach `HR_CHAR` ist).

Wir hatten schon darauf hingewiesen, dass i. A. mehrere verschiedene Services vorliegen können; diese werden in der vierten Zeile in einem Tupel zusammengefasst und der Variablen `SERVICES` zugewiesen. Auch dieses Tupel besteht in unserem Fall lediglich aus einem einzigen Element, nämlich `HR_SERVICE`. In der Variablen `SERVICES` stehen jetzt alle relevanten Parameter unseres GATT-Profils.

Mit diesem GATT-Profil wird nun in der letzten Zeile der TTGO (Server) konfiguriert. Dabei liefert die dazu benutzte Methode `gatts_register_services` einen Rückgabewert. Dieser besteht aus einem Tupel, dessen Elemente selbst wieder aus Tupeln bestehen; das erste Tupel bezieht sich auf den ersten Service, das zweite auf den nächsten, usw. Da wir nur einen einzigen Service benutzen, bleibt es bei dem ersten Tupel. Und dieses Tupel besitzt auch nur ein einziges Element; dieses benennen wir mit der Variablen `self.hr` und machen es damit zu einer Eigenschaft unserer Klasse `MyBLEServer`.

Was hat es nun mit `self.hr` auf sich? Hierbei handelt es sich um einen Value Handler, also um einen Zeiger, mit dem wir vom Micropython-Programm aus einen Wert in die Schublade "HR-Value" legen (schreiben) können (vgl. Abb. 1). Dies geschieht mit der Methode `gatts_write`, welche wir durch

```
def write(self, data):
   self.ble.gatts_write(self.hr, data)
```

in eine Methode `write` unserer Klasse `MyBLEServer` kapseln.

Jetzt können wir uns um das Hauptprogramm kümmern:

```
ble = MyBLEServer("ESP32-HR")
```

```
while not ble.conn: # warten, bis Verbindung besteht
    pass

bpm = 63 # Testwert (kleiner als 256)
bpm = bytes([0, bmp])
ble.write(bpm)
```

Zunächst wird hier eine Instanz `ble` der Klasse `MyBLEServer` erzeugt; im Rahmen dieser Instanziierung wird auch das Advertising unter dem Peripheral-Namen `"ESP32-HR"` gestartet.

Danach wartet das Programm solange, bis eine Verbindung aufgebaut worden ist; in diesem Augenblick wird nämlich über den IRQ_CENTRAL_CONNECT-Interrupt `ble.conn` auf `True` gesetzt.

Anschließend legen wir (willkürlich) einen Wert für die Pulsrate (**bpm** steht für **b**eats **p**er **m**inute) fest, in diesem Fall haben wir dafür die Zahl 63 gewählt. Um das Protokoll für den HR-Service zu erfüllen, muss der HR-Wert ein Bytes-Objekt bestehend aus 2 Bytes sein. Dabei muss das erste Byte 0 sein, das zweite gibt dann unseren bmp-Wert an: `bpm = bytes([0, bmp])`. Mit dem letzten Befehl wird schließlich dieses Bytes-Objekt in der Schublade “HR-Value” abgelegt.

Das gesamte Programm finden Sie auch unter dem Dateinamen `ble_hr_0.py`.

Jetzt ist es an der Zeit, unser Programm auszutesten. Nach dem Start erscheint im Terminal der Hinweis `advertising`. Nun öffnen wir die App *nRF Connect* ; nach einem kurzen Scan finden wir unseren Advertiser unter der Bezeichnung `"ESP32-HR"`. Mit dem Button `CONNECT` bauen wir eine Verbindung auf; im Terminal wird dies mit dem Hinweis `Central connected` angezeigt.

Auf dem Handy-Display sehen wir jetzt 3 Blöcke; für uns ist nur der unterste relevant. Er hat den Titel “Heart Rate”; darunter steht auch der uns bekannte UUID für den Heart Rate Service. Nun drücken wir auf diesen Block; jetzt wird die Chacteristic “Heart Rate Measurement” angezeigt. Auch hier erkennen wir den zugehörigen UUID. Darunter wird durch die Eigenschaft (Property) READ angezeigt, dass wir vom Handy aus den Pulsraten-Wert lesen können. Dazu betätigen wir den Pfeil auf der rechten Seite; dadurch fordert das Handy (Client) den aktuellen Pulsraten-Wert an und es erhält den vom TTGO gesendeten Wert 63 (s. Abb. 2).

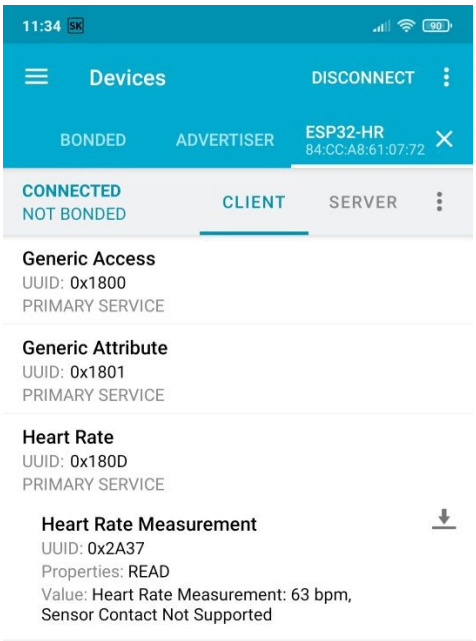


Abb. 2

Nun gehen wir einen Schritt weiter: Wir ändern das Hauptprogramm so ab, dass wir in einer Endlos-Schleife immer wieder neue Pulsraten-Werte eingeben können.

```
ble = MyBLEServer("ESP32-HR")

while not ble.conn:
  pass

while True:
  if taster0.value(): # warten, bis Taster0 betätigt wurde
    pass
  elif ble.conn:
    bpm = int(input('Input bpm: '))
    bpm = bpm & 0xff # bpm jetzt maximal 255
    bpm = bytes([0, bpm])
    ble.write(bpm)
    print()
```

Vor dem Hauptprogramm müssen wir erst noch `taster0` definieren:

```
# oberer Taster:
from machine import Pin
taster0 = Pin(0, Pin.IN, Pin.PULL_UP)
```

Unser neues Programm finden Sie in der Datei `ble_hr_0a.py`. Nach dem Verbindungsaufbau, kann man nun einen neuen Pulsraten-Wert am Terminal eingeben, sobald man den oberen Taster des TTGO-Module (S0) betätigt hat. Dabei können Sie sich auf dem Handy mit dem Pfeil-Symbol den jeweils aktuellen Pulsraten-Wert anzeigen lassen.

Bei diesem TTGO-Programm muss man jeden Werte jeweils einzeln anfordern. Es reicht nicht aus, das Pfeil-Symbol am Anfang ein einziges Mal zu betätigen. Diese Vorgehensweise bezeichnet man als **Polling**.

Bequemer wäre es, wenn nach einer Eingabe am Terminal der aktuelle Wert automatisch auf dem Handy angezeigt würde. Das lässt sich mit der `gatts_notify`-Methode erreichen, die wir in unsere eigene `BLE`-Klasse aufnehmen:

```
def notify(self, data):
  self.ble.gatts_notify(0, self.hr, data)
```

Diese Methode hat zwei Funktionen: Zum Einen sendet sie eine so genannte **Notification Request** an das Handy, zum Anderen sendet sie auch den Wert des Parameters `data` an das Handy

(Client). Allerdings finden diese beiden Aktionen nur dann statt, ***wenn*** dem Server eine entsprechende Erlaubnis vorliegt. Um diese Erlaubnis zu erhalten, muss zunächst die Zugriffsmöglichkeit von HR_CHAR geändert werden: `_FLAG_READ` müssen wir durch `_FLAG_NOTIFY` ersetzen; den Wert dafür tragen wir unsere Konstantenliste ein: `_FLAG_NOTIFY = const(16)`. Auf dem Handy sehen wir (nach dem Verbindungsaufbau) jetzt statt des einfachen Pfeils einen Dreifach-Pfeil. Über diesen Button kann der Client (Handy) dem Server (TTGO) diese Erlaubnis nun erteilen oder auch entziehen. Wie dieser Notification-Mechanismus funktioniert, werden wir in Kap. 13 näher erörtern.

Testen Sie das neue Programm aus! Sie finden es auch unter dem Dateinamen `ble_hr_0b.py`.

Zuletzt wollen wir noch den Herzschlag durch den TTGO simulieren lassen: Dazu ersetzen wir in dem letzten Programm die Endlos-Schleife `while True: ...` durch

```
while True:
  sleep(3)
  bpm = randint(50, 65) #simulierter bpm-Wert zwischen 50 und 65
  print(bpm)
  bpm = bytes([0, bpm])
  ble.notify(bpm)
```

Zusätzlich importieren wir vor dem Hauptprogramm die Funktion `randint` aus dem Modul `random`:

```
# Zufallszahlen:
from random import randint
```

und aus dem Modul `time` die Funktion `sleep`:

```
# Pausen:
from time import sleep
```

Die Definition des Tasters können wir außerdem löschen. Das fertige Programm finden Sie in der Datei `ble_hr_0c.py`.

Nachdem Sie Ihr Handy wie üblich mit dem TTGO verbunden und mit dem Dreifach-Pfeil auch die Erlaubnis für das automatische Versenden des Pulsraten-Wertes erteilt haben, wird jede drei Sekunden auf dem Handy-Display ein neuer Pulsraten-Wert angezeigt. Sie sollten zwischendurch auch einmal für einige Zeit die Erlaubnis entziehen und kontrollieren, ob dann noch weitere Werte auf dem Handy ankommen.

Bei der App BLE Scanner muss der Benutzer die Erlaubnis für die Notification explizit geben (Button R); bei der App nRF Connect ist dies nicht erforderlich: Vermutlich wird hier von der App diese

Erlaubnis direkt am Anfang erteilt, ohne dies kundzutun (vgl. auch Kap. 13).

Die gerade behandelte Notification Request dient nicht nur der Bequemlichkeit. Sie leistet auch einen weiteren Beitrag zu Einsparung von Energie: Zum Einen werden im Gegensatz zum Polling nur dann Datenpakete ausgetauscht, wenn wirklich neue Messwerte vorliegen. Zum Anderen kann der Client (bzw. der Benutzer des Handy) auch dem Server signalisieren, wenn keine Notification Requests und auch keine Messdaten mehr gesendet werden sollen, z. B. weil der Benutzer gerade keinen Bedarf dafür hat. Insgesamt müssen Sender und Empfänger somit seltener aktiviert werden.

Wer mag, kann den TTGO bei unserem letzten Programm natürlich auch ohne den PC betreiben - z. B. mit einer Powerbank. Für diesen Fall kann man die aktuellen `bpm`-Werte ggf. über das Display des TTGO kontrollieren (vgl. Kap. 8).

12 Der Nordic UART Service (NUS)

In diesem Kapitel wollen wir uns mit dem Nordic UART Service beschäftigen. Er dient dazu, Zeichenketten zwischen einem Client und einem Server auszutauschen. Das NUS-GATT-Profil stammt nicht von der SIG, sondern von der Firma *Nordic Semiconductors*, derselben Firma, von der auch die hier benutzte App nRF Connect stammt. Informationen zum NUS findet man auf der Homepage der Firma (vgl. [5]). Insbesondere sind dort auch die benötigten UUIDs angegeben.

Das zugehörige Programm für den TTGO unterscheidet sich im Wesentlichen nur in zwei Punkten von dem im letzten Kapitel behandelten Programm für den Heart Rate Service: Da wäre zunächst die `register`-Methode. Hier gibt es statt einer einzigen Charakteristic nunmehr zwei:

`BLE_TX` für die vom Server zu sendende Zeichenkette (Handler: `self.tx`)
`BLE_RX` für die vom Server zu empfangene Zeichenkette (Handler: `self.rx`)

Beachten Sie bei der folgenden Definition der `register`-Methode, dass sich das Flag für die Permission von `BLE_RX` auf den Client bezieht!

```
def register(self):

  # Nordic UART Service (NUS)

  NUS_UUID = '6E400001-B5A3-F393-E0A9-E50E24DCCA9E'
  RX_UUID = '6E400002-B5A3-F393-E0A9-E50E24DCCA9E'
  TX_UUID = '6E400003-B5A3-F393-E0A9-E50E24DCCA9E'

  BLE_NUS = ubluetooth.UUID(NUS_UUID)
  BLE_RX = (ubluetooth.UUID(RX_UUID), _FLAG_WRITE)
  BLE_TX = (ubluetooth.UUID(TX_UUID), _FLAG_NOTIFY)

  BLE_UART = (BLE_NUS, (BLE_TX, BLE_RX,))
  SERVICES = (BLE_UART, )
  ((self.tx, self.rx,), ) = self.ble.gatts_register_services(SERVICES)
```

Sodann müssen wir uns noch um den Empfang kümmern. Sobald der TTGO Daten vom Client empfängt, wird ein Interrupt mit der Event-Kennung 3 (_IRQ_GATTS_WRITE) ausgelöst. Bei einem solchen Ereignis muss die erhaltene Zeichenkette mit der gatts_read-Methode gelesen, anschließend gemäß UTF-8 dekodiert und auf dem Terminal ausgegeben werden. Dies geschieht mit dem folgenden `elif`-Block, welchen wir am Ende des Interrupt Handlers einfügen:

```
elif event == _IRQ_GATTS_WRITE:
  buffer = self.ble.gatts_read(self.rx)
  message = buffer.decode('UTF-8')
  print('Received: ', message)
```

Das Hauptprogramm ähnelt dem von der letzten Heart Rate -Version:

```
from machine import Pin
taster0 = Pin(0, Pin.IN, Pin.PULL_UP)

ble = MyBLEService('ESP32-UART')

while not ble.conn:           # warten, bis Verbindung besteht
  pass

# Zeichenkette eingeben und senden:
while True:
  if taster0.value():         # warten, bis Taster0 betätigt wurde
    pass
  elif ble.conn:              # nur falls Verbindung noch besteht...
    inp = input('Send: ')     # Zeichenkette
    ble.notify(inp)
```

Das gesamte Programm finden Sie in der Datei `ble_UART_0.py`.

Nun ist es an der Zeit, das Programm auszutesten: Wir starten das Programm auf dem TTGO und stellen mit der App nRF Connect eine Verbindung zu *ESP32-UART* her. Nun betätigen wir den oberen Taster und geben eine Zeichenkette ein, z. B. “Hallo Welt!”. Diese erscheint sofort auf dem Handy bei der TX Characteristic (vgl. Abb. 1). Anschließend betätigen wir auf dem Handy-Display den Pfeil-Button bei der RX Characteristic; es öffnet sich ein Fenster, in welchem wir einen Text eingeben können, z. B. “Hallo TTGO!”. Nachdem wir diese Eingabe mit dem SEND-Button abgeschlossen haben, wird der eingegebene Text auf dem Terminal angezeigt (vgl. Abb. 2).

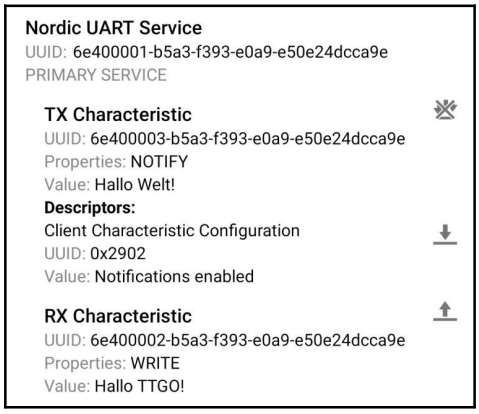


Abb. 1

```
>>> %Run -c $EDITOR_CONTENT
advertising
Central connected
Send: Hallo Welt!
Received: Hallo TTGO!
```

Abb. 2

Wer das Programm ausgiebig getestet hat, wird vielleicht schon festgestellt haben, dass Zeichenketten nur bis zu einer Länge von 20 Zeichen übertragen werden. Der Grund liegt darin, dass der von Micropython bereit gestellten Puffer standardmäßig maximal 20 Bytes speichern können. Diesen Puffer können wir aber vergrößern, z. B. auf eine Länge von 200. Dazu schreiben wir an das

Ende der `register`-Definition die beiden Zeilen

```
self.ble.gatts_set_buffer(self.tx, 200, True)
self.ble.gatts_set_buffer(self.rx, 200, True)
```

Diese Puffervergrößerung wirkt sich allerdings nur auf die Methoden `gatts_read` und `gatts_write`, nicht aber auf die `gatts_notify`-Methode aus. Deswegen ändern wir unser Programm noch an zwei Stellen ab: In der Definition der `register`-Methode wird `_FLAG_NOTIFY` durch `_FLAG_READ` ersetzt und im Hauptprogramm schreiben wir statt `ble.notify(inp)` nun `ble.write(inp)`.

Das entsprechende Programm finden Sie in der Datei `ble_UART_0a.py`.

Kann unser Programm auch mit **Sonderzeichen** umgehen? Die Antwort lautet: Im Prinzip Ja! Leider kann bei meiner aktuelle Firmware der `input`-Befehl Sonderzeichen nicht korrekt verarbeiten: verstümmelte Rückgabewerte führen zu Fehlermeldungen bei der Thonny-IDE. Aus diesem Grund testen wir unseren Server folgendermaßen aus: Vom Handy aus senden wir eine Zeichenkette mit Sonderzeichen an den TTGO, und dieser gibt die empfange Zeichenkette (quasi als Echo) an das Handy zurück.

Dazu müssen wir das Programm `ble_UART_0a.py` nur an wenigen Stellen abändern: Das Hauptprogramm beschränkt sich auf die Zeile

```
ble = MyBLEServer('ESP32-UART')
```

und der `elif-Block` zu `_IRQ_GATTS_WRITE` wird um folgende zwei Zeilen erweitert:

```
self.write(buffer)
print('Returned: ', message)
```

Das vollständige Programm dazu steht in der Datei `ble_UART_0b.py`.

Wer einen Temperatursensor besitzt, kann seinen TTGO nun auch als **Temperatur-Server** einsetzen. Dazu sind nur wenige Änderungen und Ergänzungen nötig. Im Gegensatz zu Kap. 8 können jetzt die Messwerte nicht mehr von *allen* Handys in der Umgebung des TTGO, sondern nur von *dem* Handy empfangen werden, welches mit dem TTGO *verbunden* wurde. Ein entsprechendes Programm für das Sensor-Modul BME/BMP280 finden Sie in der Datei `ble_UART_Temperatur.py`. Im Quelltext ist auch angegeben, wie dieser Sensor an den TTGO angeschlossen wird.

13 Das GATT-Profil unter der Lupe

In den Kapiteln 12 und 13 haben Sie das GATT-Profil bereits in groben Zügen kennen gelernt. Die Vorstellung vom Schrank-Modell hat uns gute Dienste geleistet bei der Programmierung des TTGO als BLE-Server. In diesem Kapitel wollen wir das GATT-Profil etwas genauer studieren.

Erinnern wir uns zunächst noch einmal an das Schichtenmodell, welches in Kap. 1 schon angesprochen worden war. Dort war allerdings nur auf die unteren beiden Schichten eingegangen worden.

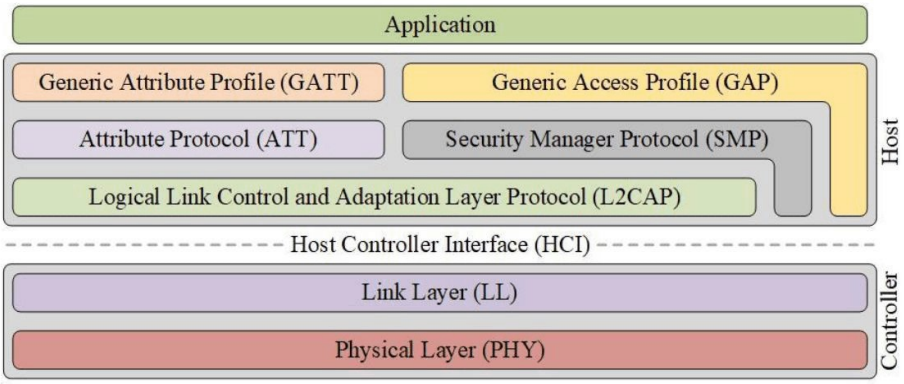


Abb. 1 (Quelle: [1])

Nun wollen wir uns die nächsten Schichten anschauen: Über dem Link Layer befindet sich das **Logical Link Control and Adaption Layer Protokoll** (kurz: **L2CAP**). Im Wesentlichen dient es als Verbindungsglied zwischen dem Controller- und dem Host-Bereich. Es kümmert sich z. B. darum, wie auch längere Datensätze mit Hilfe von BLE-Datenpaketen (vgl. Abb. 1 aus Kap. 3) übertragen werden können, obwohl diese nur kleine Datenmengen aufnehmen können; dazu zerlegt das L2CAP diese Datensätze in kleinere Segmente, die dann als Payload von BLE-Datenpaketen abgeschickt werden. Beim Empfänger müssen sie in passender Weise wieder zu dem ursprünglichen Datensatz zusammengefügt werden. Nur so können wir z. B. mit dem UART-Service auch längere Zeichenketten übertragen. Um weitere Details des L2CAP wollen wir uns nicht kümmern, weil wir für die Programmierung des TTGOs im Rahmen dieser Einführung solche Kenntnisse nicht benötigen.

Der **SMP**-Block in der nächsten Schicht widmet sich Sicherheitsaspekten. Auch auf diesen Block werden wir hier nicht weiter eingehen. Um so genauer wollen wir nun aber die restlichen drei Blöcke anschauen.

Attribute Protocol (ATT)

In diesem Protokoll werden die Datenpakte beschrieben, wie sie zur Übertragung der in Kap. 11 und 12 beschriebenen Attribute benutzt werden. In Abb. 2 ist der für uns wichtige Teil des Payloads dargestellt.

GATT Attribute	Handle	Type	Value	Permissions
	2 Bytes	2 or 16 Bytes	Variable length	Variable

Abb. 2 (Quelle: [1])

Der Handle ist ein Index, welchen der Server dem Attribut zuordnet, um auf dieses Attribut zugreifen zu können; ein Beispiel dafür war der `hr`-Handle in Kap. 11. In dem Type-Feld wird der UUID (vgl. Kap. 12) eingetragen. In dem Value-Feld befinden sich die Daten, welche durch den UUID beschrieben werden; dies können Zahlen, Zeichenketten, aber auch Handles sein. Im letzten Feld wird angegeben, welche Zugriffsrechte der ***Client*** auf dieses Attribut hat.

Solche Attribut-Felder sind uns schon in Kapitel 11 begegnet, als wir GATT-Profile mit Hilfe von Micropython konfiguriert hatten. Mehr dazu erfahren Sie im übernächsten Abschnitt.

Generic Access Profile (GAP)

Das GAP beschreibt z. B., wie ein BLE-Gerät detektiert werden kann oder wie eine Verbindung aufgebaut wird. Dazu gehören auf der `ubluetooth`-Ebene die schon behandelten Methoden `gap_advertise`, `gap_scan` und `gap_connect`.

Generic Attribute Profile (GATT-Profil)

Das GATT-Profil stellt ein hierarchische Struktur zur Verfügung, nach der Daten organisiert und übertragen werden. In der Abb. 3 ist ein GATT-Profil für das Heart Beat Beispiel aus Kap. 11 angegeben: In der ersten Zeile erkennen wir das Attribut für den Service; in den folgenden drei Zeilen befinden sich die Attribute, welche die Characteristic für die Pulsrate des Herzschlags beschreiben. (In Kap. 11 hatten wir schon darauf hingewiesen, dass hier weitere Characteristics möglich sind.)

Wir sehen sofort: Unsere bisherige Darstellung des GATT-Profils in Kap. 11 war etwas vereinfacht; offensichtlich hatte sie aber schon ausgereicht, die benutzen Micropython-Befehle so mit den erforderlichen Parametern zu versorgen, dass diese daraus die vier in Abb. 3 angegebenen Attribute konstruieren konnten.

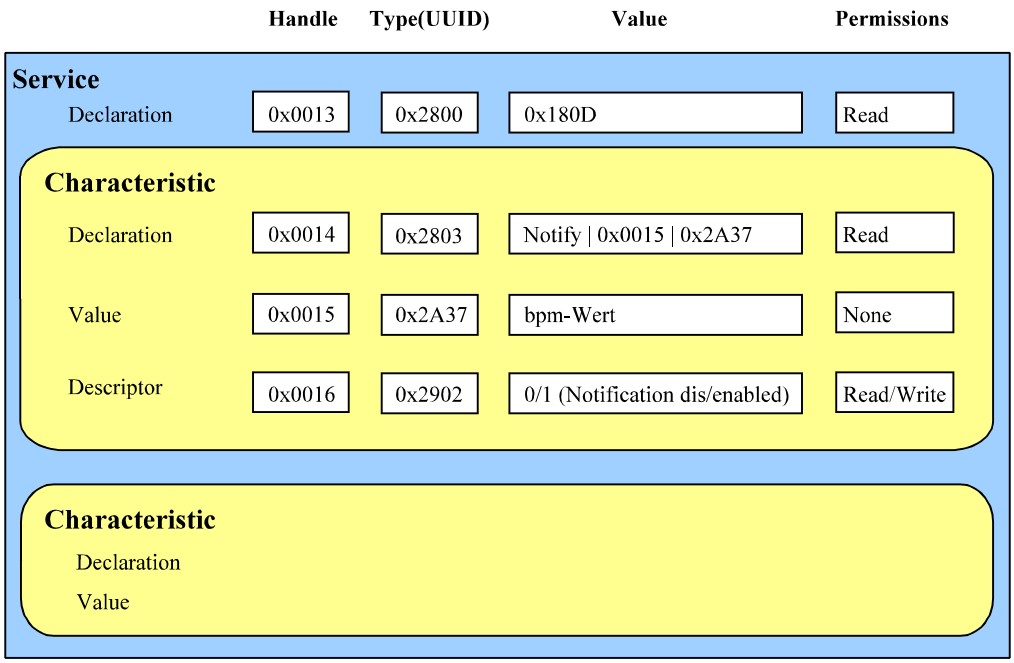


Abb. 3

Schauen wir uns nun die einzelnen Attribute der Reihe nach an:

1. Attribut: Dass es sich bei diesem Attribut um eine **Deklaration für einen Service** handelt, wird durch den UUID 0x2800 im Type-Feld erkennbar: Hierbei handelt es sich nämlich um den Standard-UUID für Service-Deklarationen. Im nächsten Feld (Value) steht ein weiterer UUID, nämlich 0x180D. Hieran wird erkennbar, dass es sich um den Heart Rate Service handelt. Das letzte Feld zeigt, dass der Client dieses Value-Feld lesen kann. Wenn unsere nRF Connect App nach dem Verbindungsaufbau dieses Attribut empfängt, kann sie diesen Wert lesen und uns anzeigen, dass dieser Service zur Verfügung steht.

2. Attribut: Die Typenbezeichnung 0x2803 zeigt an, dass es sich hier um die **Deklaration einer Characteristic** handelt. Das anschließende Value-Feld liefert einen Überblick über die entscheidenden Daten: das Handle, mit dem auf den bpm-Wert zugegriffen werden kann (0x0015), sodann den UUID 0x2A37, welcher die Messung für die Heart Rate kennzeichnet, und zusätzlich noch die Eigenschaft **Notify**. Diese Eigenschaft sorgt dafür, dass der Server den bpm-Wert mit einem **Notification Request** an den Client senden kann; für diesen Request gibt der Client kein Acknowledge. Wenn Notify gesetzt worden ist, dann muss es auch einen Deskriptor geben (vgl. 4. Attribut). Der Eintrag Read im letzten Feld sorgt dafür, dass der Client diese Deklaration lesen kann.

3. Attribut: In diesem Attribut steht endlich der aktuelle bpm-Wert (Value-Feld). Der Typ ist - wie bereits in der Deklaration angekündigt - der UUID 0x2A37. Auf der GATT-Ebene gibt es keine Zugriffsrechte. Über die Application-Schicht können wir allerdings darauf zugreifen, z. B. mit der gatts_write-Methode, die wir z. B. schon in Kap. 11 eingesetzt haben.

4. Attribut: Dieses Attribut repräsentiert einen **Descriptor**. Eine Characteristic *kann* einen oder auch mehrere Descriptors besitzen, muss aber nicht. Ein Descriptor enthält zusätzliche Informationen zur Characteristic. Hier handelt es sich um einen speziellen Descriptor, und zwar um den **Client Caracteristic Configuration Descriptor** (kurz **CCCD**). Ein solcher CCCD wird durch die UUID 0x2902 gekennzeichnet. In unserem Fall ist der CCCD unverzichtbar: In Kap. 11 hatten wir schon dargelegt, dass der Client darüber entscheidet, ob der Server Messwerte mit einem Notification Request senden kann oder nicht. In unserer Handy-App hatten wir dies mit dem Drei-Pfeile-Button festgelegt. Nun können wir deutlich machen, wie dies geschieht. Wenn die Notification damit aktiviert werden soll, dann sorgt der Client dafür, dass beim Server der Wert 1 in das Value-Feld des Descriptors eingetragen wird, ansonsten eine 0; das ist möglich, weil der Descriptor eine Write-Permission hat. An dem Wert des Value-Feldes vom CCCD kann der Server nun erkennen, ob er im Rahmen der `gatts_notify`-Methode einen Messwert inklusive Notification Request an den Client senden soll oder nicht.

Mancher fragt sich vielleicht an dieser Stelle, wieso in Kap. 11 dieses Attribut im Programm gar nicht (explizit) aufgetaucht ist. Offensichtlich hat Micropython beim Ausführen der `gatts_register_service`-Methode automatisch dieses Attribut generiert, sobald es in dem `CHAR_HR`-Tupel auf ein `FLAG_NOTIFY` gestoßen ist.

Nach dem Verbindungsaufbau wird der Client - dem hierarchischen Aufbau folgend - zunächst die Service Declaration vom Server anfordern. Eine spezielle Heart Rate App etwa wird sicherlich hier schon das Value-Feld dieses Attributs überprüfen: Hat es nicht den erwarteten Wert UUID(0x180D), dann wird sie wohl eine Fehlermeldung auf dem Display ausgeben; andernfalls wird sie weitere Attribute anfordern.

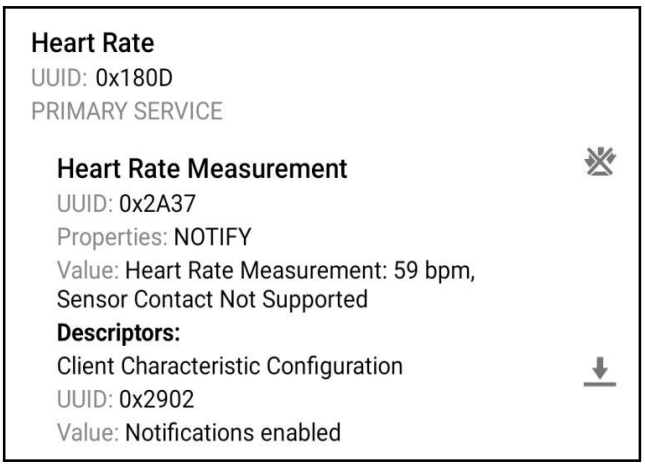


Abb. 4

Unsere App nRF Connect ist hingegen nicht auf ein bestimmtes Profil ausgerichtet. Deswegen wird sie das gesamte Profil so

weit wie erforderlich vom Server anfordern. In Abb. 4 sehen Sie einen Screenshot von der App nRF Connect: Das Handy hat bereits eine Bluetooth-Verbindung mit dem TTGO aufgebaut, auf dem unser Programm `ble_hr_0c.py` läuft; dieses Programm benutzt das GATT-Profil aus Abb. 3 mit aktivierter Notification (vgl. auch Kap. 12). Sehr rasch werden Sie selbst viele Teile aus diesem GATT-Profil in dem Screenshot identifizieren können.

Einige Fragen ergeben sich allerdings noch: Woher hat die App nRF Connect die Information, dass es sich um ein "Heart Rate Measurement" handelt? Wie gelangt es an die zugehörige Einheit "bpm"? Diese Informationen stehen ja nicht explizit in unserem GATT-Profil! Und wie ist der Hinweis zu dem Eintrag "Sensor Contact Not Supported" zu erklären?

Nun, offensichtlich ist es so, dass in der App nRF Connect bereits Informationen zu dem Heart Rate Service Standard implementiert worden sind; an den vom Server empfangenen UUIDs kann die App den Service und die Characteristic identifizieren und dementsprechend passende Überschriften bilden, sowohl für den Service als auch für die Characteristic; das Gleiche gilt für die Einheit "bpm".

Der Heart Rate Service bietet standardmäßig auch weitere (optionale) Characteristics an, z. B. die Body Sensor Location; hierin wird festgehalten, an welcher Stelle des Körpers der Sensor angeschlossen ist. Weil die App auch diese Option kennt, überprüft sie, ob vom TTGO Informationen bezüglich der Characteristic "Body Sensor Location" zur Verfügung gestellt werden. Da dies hier nicht der Fall ist, kommt es zu dem Eintrag "Sensor Contact Not Supported".

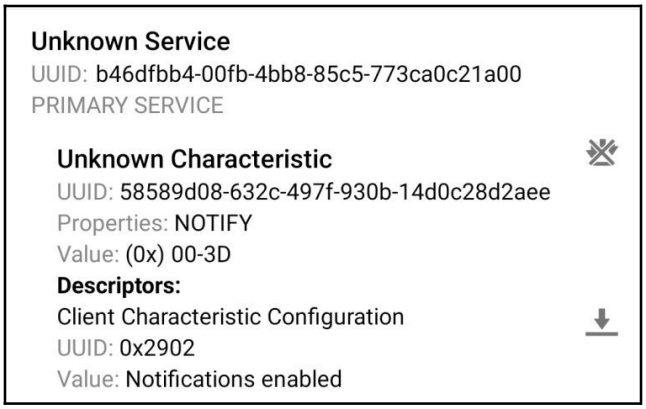


Abb. 5

Wie würde die App reagieren, wenn diese Vorinformationen nicht zur Verfügung stehen? Das können wir leicht in Erfahrung bringen. Dazu ersetzen wir in dem gerade benutzten Programm die Standard-UUIDs 0x180D und 0x2A37 durch eigene UUIDs, welche die App nicht kennen kann. Diese besorgen wir uns mit Hilfe eines UUID-Generators (z. B. über die Webseite https://www.uuidgenerator.net/). Das zugehörige Programm finden Sie in der Datei `ble_hr_0d.py`. In der Abb. 5 sehen Sie, dass jetzt weder passende Überschriften noch eine Einheit angegeben werden. Obendrein wird der bpm-Wert nicht entschlüsselt, sondern lediglich als Bytefolge ausgegeben.

Wenn der Hersteller eines Bluetooth-Pulsraten-Messgerätes eigene UUIDs benutzt, wird er sicherlich auch eine passende App für Handys anbieten, welche diese UUIDs kennt. Nach dem Connecting wird die App gezielt nach Attributen mit diesen UUIDs Ausschau halten und diese entsprechend interpretieren und darstellen.

Eine Frage bleibt vielleicht noch übrig: Warum ist das GATT-Profil deutlich komplexer, als als wir es zuvor in der einfacheren, aber für die Programmierung offensichtlich hinreichenden Weise von Kap. 11 kennen gelernt hatten. Es gibt mehrere Gründe. Zum Einen ist für einen Service schon durch den Bluetooth Standard eine bestimmte Struktur vorgeschrieben: Danach muss dieser immer eine Service Declaration und mindestens eine Characteristic besitzen; und letztere muss ihrerseits mindestens ein Declaration-Attribut und ein Value-Attribut beinhalten. Demnach muss jeder Service mindestens 3 Attribute besitzen. Wenn wir zusätzlich noch einen Notification Request - Mechanismus einsetzen wollen, der über den Client ein- und ausgeschaltet werden kann, dann benötigen wir ein Attribut mit der Permission `Write`. Keiner der bisherigen drei Attribute weist (sinnvollerweise!) diese Permission auf. Deswegen ist ein viertes Attribut erforderlich; das ist gerade unser CCCD!

Zu guter Letzt wollen wir das GATT-Profil unseres Servers aus dem Programm `ble_hr_0c.py` ausbauen: Es soll nun neben dem HR-Service einen weiteren Service erhalten. Dieser soll Informationen über das BLE-Gerät zur Verfügung stellen. Wir begnügen uns mit zwei Angaben:

- die BLE-Adresse
- den Modul-Namen

Diese Angaben werden durch je eine Characteristic dargestellt; diese bezeichnen wir im Programm mit `DI_CHAR_SYSTEM_ID` bzw. `DI_CHAR_MODEL`. Die `register`-Methode lautet damit:

```
def register(self):

  # Heart Rate (HR)
  HR_UUID = ubluetooth.UUID(0x180D)  # Service 1
  HR_CHAR = (ubluetooth.UUID(0x2A37), _FLAG_NOTIFY)
  HR_SERVICE = (HR_UUID, (HR_CHAR,))

  # Device Information (DI)
  DI_UUID = ubluetooth.UUID(0x180A) # Service 2
  DI_CHAR_SYSTEM_ID = (ubluetooth.UUID(0x2A23), _FLAG_READ)
  DI_CHAR_MODEL = (ubluetooth.UUID(0x2A24), _FLAG_READ)
  DI_SERVICE = (DI_UUID, (DI_CHAR_SYSTEM_ID, DI_CHAR_MODEL))

  SERVICES = (HR_SERVICE, DI_SERVICE)
  ((self.hr,),(self.di_id, self.di_mod)) = self.ble.gatts_register_services(SERVICES)
```

Hierbei wurde auf weitere 16-Bit-UUIDs der SIG zurückgegriffen:

```
0x180A      Device Information
0x2A23      System ID
0x2A24      Model Number String
```

Zusätzlich muss die Funktion `__init__` ergänzt werden: Unter den Funktionsaufruf `self.register` fügen wir ein:

```
self.ble.gatts_write(self.di_id, self.get_my_addr())
self.ble.gatts_write(self.di_mod,'TTGO T-DISPLAY')
```

Außerdem müssen wir noch die Methode `get_my_address` definieren:

```
def get_my_addr(self):
  return self.ble.config('mac')[1]
```

Schließlich geben wir der Instanz auch einen anderen Namen, um sie von der bisherigen HR-GATT zu unterscheiden:

```
ble = MyBLEServer("ESP32-HR-DI").
```

Auf dem Handy erhalten wir damit eine Anzeige wie in Abb. 6.

Das vollständige Programm finden Sie in der Datei `ble_GATT_2_Services_1.py`.

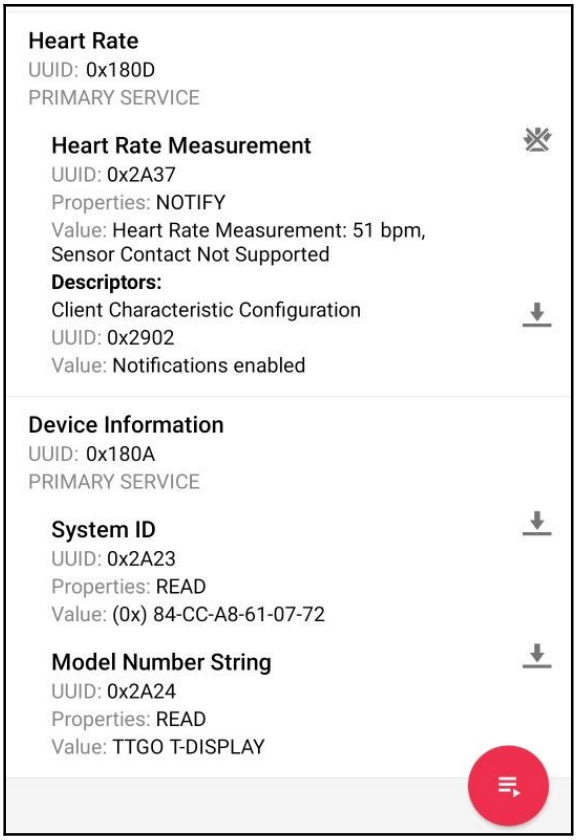


Abb. 6

Quellen und Literaturhinweise

[1]: www.dta.mil.nz/assets/Publications/A-Summary-of-Bluetooth-Low-Energy.pdf

[2]: Michael Bonacina: Python 3, Markt+Technik Verlag, Burgthann

[3]: https://docs.python.org/3/tutorial/classes.html

[4]: https://www.digitalocean.com/community/tutorials/understanding-class-and-instance-variables-in-python-3

[5]: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/bluetooth/services/nus.html

[6]: https://www.bluetooth.com/specifications/specs/core-specification/

[7]: https://docs.micropython.org/en/latest/library/ubluetooth.html

[8]: https://www.ti.com/lit/an/swra475a/swra475a.pdf (Abschnitt 2.4.1)

Stichwortverzeichnis