Lego Boost Roboter steuern mit Python unter Windows oder Linux

Inhaltsverzeichnis

Verwendete Betriebssysteme und Softwareversionen:

PC mit Windows 7, 64 Bit und Python 3.6.5 64 Bit
PC mit Ubuntu 18.04 LTS und Python 3.6.9, pygatt 4.0.5, pyserial 3.4
Rapberry Pi 3 mit Rasbpian Stretch „with Desktop“ GNU/Linux 9 Version 4.9.59 und Python 3.5.3
Raspberry Pi Zero W mit Rasbpian Stretch „lite“ GNU/Linux 9 Version 4.14.34 und Python 3.5.3

Achtung:

Diese Anleitung gilt ausschließlich für Python 3.

Aktuell (Stand März 2020) wurde die Bibliothek *`pylegoboost`* ausschließlich mit einem Blue-Giga-Adapter BLED112 unter Ubuntu 18.04 LTS eingehend getestet.

Die Test wurden an einem Lego Boost MoveHub vom Herstelljahr 2018 durchgeführt. Es ist möglich, dass neuere MoveHub eine andere Firmware besitzen, für die die Bibliothek *`pylegoboost`* angepasst werden muss.

0 Einleitung

„Lego Boost“ ist ein Robotikset von Lego®, welches für Kinder von unter 10 Jahren als Einstieg in die Robotik gedacht ist. Die Hardware ist sehr ähnlich und **teils kompatibel mit dem WeDo** Robotiksystem von Lego Education®, welches aber ausschließlich für den Schulbereich gedacht ist.

Das **Konzept vom Lego Boost sah ursprünglich vor, diesen Roboter ausschließlich zusammen mit einem Tablet zu nutzen**, welches als Master-Rechner für die Steuerung, für die Programmierumgebung als auch für die Darstellung der Bauanleitungen dient. Zwischenzeitlich gab es diese Programmierumgebung auch für Windows 10 PCs, sie wurde jedoch in 2019 wieder eingestellt. Im PC-Bereich kann man (Stand März 2020) den Lego Boost leider nur mit einem Apple-PC programmieren - von Linux ganz zu schweigen. Generell steht weniger das eigenständige Konstruieren und Programmieren im Vordergrund, sondern das Nachbauen von vorgegebenen Modellen sowie ein anschließendes vorgegebenes schrittweises Programmieren nach Plan. **Kreatives Konstruieren und Programmieren steht nicht im Vordergrund.** Es ist aber möglich über die sogenannten „creative canvas".
Andere Programmierumgebungen werden von Lego wohl auch im Hinblick auf die Alterszielgruppe bewusst nicht angeboten und auch nicht unterstützt.

Das Programm wird nicht wie bei Lego Mindstorms auf einem Mikrocontroller im Roboter ausgeführt. Bei Lego Boost ist die Roboterhardware ein sogenannter Slave, der seine Befehle über die BLE-Schnittstelle von einem externen Computer (Master) erhält und dort hin auch seine Sensorwerte sendet. Das eigentliche Roboterprogramm läuft somit nicht auf dem Roboter selbst sondern auf dem externen Computer – bevorzugt ein Tablet – ab

Zum Glück gab es von Anfang an kreative Hacker, die die unverschlüsselte Bluetooth Low Energy (BLE) Kommunikation der Lego-App mit dem Boost Roboters belauschten. Mit dem dadurch gewonnen Wissen wurden von mehreren Personen quelloffene Softwarebibliotheken erstellt. Inzwischen (Stand 2020) hat Lego jaber sein Kommunikationsprotokoll offengelegt.

Dadurch wurde es möglich, den Lego Boost auch mit der **kostenlosen Open Source Programmiersprache Python zu betreiben bzw. programmieren.**
Anders als bei der Lego-App **ist hier die Rechnerplattform egal**, denn Python gibt es auf PCs mit Windows, Linux oder Mac OS, sowie auf den meisten Einplatinencomputern wie beispielsweise dem Raspberry Pi.

Python ist als Programmiersprache gerade stark im Kommen, da sie von großen Softwarefirmen z.B. für Anwendungen in der künstlichen Intelligenz verwendet wird, und weil es im Internet eine große Anzahl guter Bibliotheken („Pakete" genannt) gibt. Außerdem **ist Python leicht zu erlernen** und bietet optimale Voraussetzungen, um das objektorientierte Programmieren zu lernen.

Es gibt leider **zwei verschiedene Linien der Programmiersprache Python: „Python 2“ und „Python 3“**. Die hier vorgestellten Pythonbibliotheken gibt es jeweils für beide Pythonversionen. Nachfolgend wird ausschließlich Python 3 verwendet. Daher werden die Bibliotheken auch mit dem Programm `pip3` anstatt `pip` installiert.

Zum Programmieren des Lego Boost unter Python steht unter GitHub die exzellente Bibliothek „`pylgbst`" bereit. Die Bibliothek `pylgbst` sollte - wenn sie auf Anhieb einwandfrei funktioniert - der Bibliothek „`pylegoboost`" vorgezogen werden. Möchte man den Code nachvollziehen, dann eignet sich `pylegoboost` besser, da der Code überschaubarer und besser kommentiert ist. Die vorliegende PDF-Doku hilft zudem, die technischen Hintergründe beider Bibliotheken besser zu verstehen.

1 Roboterhardware des Lego Boost

Der Kernbaustein des Boost Roboters ist der sogenannte „Move Hub“. Dieser beinhaltet einen Steuerrechner, zwei Motoren mit Encoder, einen 3-Achsen-Beschleunigungssensor, eine RGB-

LED, einen Taster sowie ein Mikrocontroller mit Bluetooth Low Energy Schnittstelle. An den zwei Schnittstellen des Move Hubs kann ein externer Motor sowie ein Farb/Abstandssensor angeschlossen werden. Innerhalb des Move Hubs werden sowohl die Batteriespannung als auch der Batteriestrom als weitere interne Sensorwerte erfasst.

2 Bluetoothkommunikation

Der Move Hub (Slave) kommuniziert via Bluetooth Low Energy (BLE) über das GATT-Protokoll mit einem Computer (Master). Dieser Steuercomputer muss also eine BLE-Schnittstelle besitzen. Ein Raspberry Pi 3 besitzt diese Schnittstelle fest verbaut auf dessen Platine. Zudem unterstützt dessen aktuelles Linux-Betriebssystem Raspian (Linux BlueZ Service) die BLE-Kommunikation.

Verfügt der Steuercomputer über keine interne BLE-Schnittstelle, so hilft ein Bluetooth-Dongle, der BLE-fähig ist (z.B.TP-Link UB400). Viele solcher BLE-Dongles funktionieren meistens auf Ubuntu-Rechnern und oftmals ;-) auf Windows-Rechnern (meistens erst ab Windows 8).

Eine Besonderheit ist der BLE-Dongle der Fa. BlueGiga (Modell BLED112): Dieser Dongle wird in einen USB-Anschluss des Steuercomputers eingesteckt, meldet sich beim Betriebssystem - egal ob Windows oder Linux - als serielle Schnittstelle an und bearbeitet die BLE-Kommunikation autonom intern. Das funktioniert übrigens auch bei einem Raspberry Pi 3. Der Dongle BLED112 kostet ca. 12 €[1].

Bei BLE gibt es anders als beim früheren „normalen“ BT nur ein Profil, nämlich GATT (Generic Attribute). GATT ist ein Schlüssel-Wert-Speicher ähnlich zu einem „Dictionary“ in Python. **Die Kommunikation mit einem BLE-Gerät besteht daher rein aus dem Lesen und Schreiben von Werten in diesem Speicher.**

Eine permanente Verbindung wie beim „klassischen“ BT ist nicht mehr nötig. Bei drahtlosen Sensornetzen z.B. wird durch diese „Kommunikation nur bei Bedarf“ sehr viel Energie gespart, was die Standzeit von Batterien erheblich verlängert.
Andererseits kann mit BLE aber beispielsweise keine Musik als Datenstrom („Streaming“) gesendet werden. Nähere Informationen siehe [2].

Dies BLE-Kommunikation unterscheidet sich stark von der gewohnten seriellen Kommunikation, bei der der Master permanent zyklisch Steuerbefehle an den Slave sendet oder dort Sensorwerte anfordert:
Bei BLE werden die Aktoren des Move Hubs wie gewohnt mit Steuerbefehlen vom Master angetriggert.
Die **Sensoren des Move Hubs verhalten sich jedoch wie die Knoten eines BLE-Sensornetzwerks**: Sie senden eigenständig die Sensorwerte an den Master, die zu Beginn vom Master „abonniert“ wurden. Je nach Parametrierung dieses „Abonnements“ werden nur dann Messwerte vom Slave gesendet, wenn sich der Messwert hinreichend stark geändert hat.

Der Slave „Move Hub“ betreibt hierfür einen GATT-Server, welcher den Master „PC“ als Client „bedient“. Der Client abonniert also beim Server eine Sensor-Messgröße, dessen Messwerte ihm der Server sendet, sobald sich diese z.B. hinreichend geändert haben.

Bei dem Taster des Move Hubs, wird in diesem Sinne nur dann eine Nachricht gesendet, wenn der Taster gedrückt bzw. losgelassen wurde. Der Master/Client muss dann selbst in Erinnerung behalten, was der letzte gesendete Status des Tasters ist. **Diese Art der *eventbasierten* Kommunikation/Programmierung erspart viel Kommunikationsarbeit**, da in diesem Beispiel der Taster nur selten betätigt wird, und nur dann eine Kommunikation stattfindet. Dies macht das „Low Energy“ im BLE aus, denn Funkkommunikation ist immer mit erheblichem Energieverbrauch verbunden.

Bei einem drahtlosen Sensornetzwerk ist oft der Energieverbrauch der Kommunikation für das ganze System dominant und bestimmt die Batteriestandzeit. Beim Move Hub ist der Energieverbrauch sicher nicht der Grund für den Einsatz von BLE gewesen, zumal die Motoren weitaus mehr

1 Er ist in Deutschland nicht leicht zu bekommen. Daher am besten aus China via Aliexpress bestellen.
2 golem.de/news/golem-de-programmiert-bluetoothle-im-eigenbau-1404-105896.html

elektrische Leistung beanspruchen als die Sensoren und die Funkschnittstelle. Bei Lego hat man sich vermutlich für BLE entschieden, da dies ein sehr aktueller und zukunftsweisender Kommunikationsstandard ist.

Die **BLE Kommunikation des Lego Boost wurde ursprünglich von dem Portugiesen Jorge Pereira und Russen Andrey Pokhilko „gehackt“**. Beide erstellten aus diesem Informationen die Pythonbibliotheken *pyb00st*[3] bzw. *pylgbst*[4]. Inzwischen hat Lego jedoch das Kommunikationsprotokoll[5] offengelegt.

Auf einem Linux-PC (z.B. Ubuntu 18.04 LTS) kann mit diesen Infos und dem Linux Tool *gatttool* sehr gut im Terminalfenster mit dem Move Hub „interaktiv und explorativ“ kommuniziert werden, wie in den nachfolgenden Beispielen dargestellt ist.

2.1 Testen der Bluetoothkommunikation unter Linux mit gatttool[6]

Mit dem Befehl *hciconfig* im Terminalfenster wird der Status der Bluetoothschnittstelle abgefragt:

```
pi@raspberrypi:~ $ hciconfig
hci0:  Type: Primary  Bus: UART
       BD Address: B8:27:EB:75:1F:F0  ACL MTU: 1021:8  SCO MTU: 64:1
       UP RUNNING
       RX bytes:822 acl:0 sco:0 events:57 errors:0
       TX bytes:4231 acl:0 sco:0 commands:57 errors:0
```

Mit dem Befehl *sudo hcitool lescan* wird nach eingeschalteten BLE-Geräten gesucht. Wenn der Move Hub durch Drücken des grünen Tasters aktiviert war, denn erscheint er mit seiner MAC-Adresse (Beenden mit *strg + c*):

```
pi@raspberrypi:~ $ sudo hcitool lescan
LE Scan ...
00:16:53:AB:64:29 (unknown)
00:16:53:AB:64:29 LEGO Move Hub
```

Nun kann über das Linuxprogramm *gatttool* mit dem Move Hub über das GATT-Protokoll kommuniziert werden Durch die Option *-I* wird eine interaktive Session gestartet, welche mit dem Befehl *exit* beendet wird.

```
pi@raspberrypi:~ $ gatttool -b 00:16:53:AB:64:29 -I
[00:16:53:AB:64:29][LE]> connect
Attempting to connect to 00:16:53:AB:64:29
Connection successful
```

2.1.1 Auslesen der vom Move Hub angebotenen GATT „Primary Services“

Ein Service ist eine Funktionalität, die der Move Hub (mit GATT-Server) seinem Client anbietet. Es gibt prinzipiell auch noch „Secondary Services“ und „Referenced Services“, die beim Move Hub jedoch nicht verwendet werden.

```
[00:16:53:AB:64:29][LE]> primary
attr handle: 0x0001, end grp handle: 0x0004 uuid: 0000180 1-0000-1000-8000-00805f9b34fb
attr handle: 0x0005, end grp handle: 0x000b uuid: 0000180 0-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x000f uuid: 00001623-1212-efde-1623-785feabcd123
```

Jeder Service trägt als Kennzeichnung eine "UUID" (Universally Unique Identifier) wie in der Ausgabe oben zu sehen ist. Diese Ziffernfolge besteht aus 32 Hexadezimalzeichen. Ein Hexadezimalzeichen steht für 4 Bit. Somit sind diese UUID 128 Bit lang.

Die innerhalb der BLE-Spezifikation vordefinierten gelb unterlegten Sevices sind: „1800“ steht für „Gereric Access“ und „1801“ für „Gereric Attribute“. Diese beiden Services muss jedes BLE-Gerät anbieten. Um auf sie zuzugreifen muss man nicht die volle 128-Bit lange UUID angeben. Für den

3 github.com/undera/pylgbst
4 github.com/JorgePe/pyb00st
5 lego.github.io/lego-ble-wireless-protocol-docs/
6 Es gibt übrigens auch eine sehr interessante APP für Android, die ähnliche Funktionen wie gatttool aufweist: „nRF Connect“ von Nordic Semiconductor

Zugriff reichen die oben fettgedruckt dargestellten vier Hexadezimalziffern, die sogenannte „16-Bit UUID".

Darüber hinaus gibt es beim Move Hub nur den herstellerspezifischen grün unterlegten Service. Auf diesen Service kann nur über die volle 128-Bit UUID zugegriffen werden.

2.1.2 Auslesen der vom Move Hub angebotenen GATT „Characteristics“

Characteristics sind Untermengen der Services. Sie beinhalten einem dem jeweiligen Service zugeordneten Wert sowie weitere Informationen dazu.

```
[00:16:53:AB:64:29][LE]> characteristics
handle: 0x0002, char properties: 0x20, char value handle: 0x0003, uuid: 0000 2a05 -0000-
1000-8000-00805f9b34fb
```

... gehört zum Service 1800

```
handle: 0x0006, char properties: 0x4e, char value handle: 0x0007, uuid: 0000 2a00 -0000-
1000-8000-00805f9b34fb
handle: 0x0008, char properties: 0x4e, char value handle: 0x0009, uuid: 0000 2a01 -0000-
1000-8000-00805f9b34fb
handle: 0x000a, char properties: 0x02, char value handle: 0x000b, uuid: 0000 2a04 -0000-
1000-8000-00805f9b34fb
```

... gehören zum Service 1801

```
handle: 0x000d, char properties: 0x1e, char value handle: 0x000e, uuid: 00001624-1212-ef-
de-1623-785feabcd123
```

... gehört zum herstellerspezifischen Service

Characteristics sind also Behälter für Daten, auf die der Benutzer zugreift. Es gibt für jede Charakteristik jeweils eine **„**Declaration**“** (inklusive „Properties“, blau unterlegt) und eine **„**Value Declaration**“** (rosa unterlegt), welche die eigentlichen Datenwerte enthält.

Auf diese Characteristic Datenwerte kann über ein 8-Bit „Characteristic Value Handle“ (zwei Hexadezimalziffern) zugegriffen werden, welches oben unterstrichen dargestellt ist (die vorangestellen beiden Nullen können weggelassen werden).

Alternativ kann bei allgemein unter BLE vordefinierten Characteristics über eine 16-Bit UUID, bei gerätespezifischen Characteristics über 128-Bit UUID zugegriffen werden. Bei der BLE-Bibliothek *`gatt`* im Abschnitt 3.1 wird z.B. nur über die 128-Bit UUID zugegriffen.

„char(acteristic) properties“ gibt an wie auf den entsprechenden Wertes unter „char(acteristic) value“ zugegriffen werden kann.

Beispielsweise kann auf die Properies *`0x1e`* des herstellerspezifischen Service (untersten Zeile der Ausgabe oben) wie folgt zugegriffen werden:
`0x1e` ist gleich *`b11110`*: Die erste 1 steht für „Notifiy“ Zugriff, die zweite 1 für „Write“ Zugriff, die dritte 1 für „Write Without Response“ Zugriff, die vierte 1 für „Read“ Zugriff und die 0 für eine nicht vorhandene „Broadcast“ Option[7].

Beim Move Hub läuft abgesehen von der Device ID die gesamte Kommunikation über den Handle des Characteristic Value *`0x0e`* bzw. die *`UUID 00001624-1212-efde-1623-785feabcd123`*.

2.1.3 Auslesen der vom Move Hub angebotenen Descriptors der Characteristics

Für einige Geräte wie hier beim Move Hub gehört noch eine **„Descriptor Declaration“** zur Characteristic, die die Datenwerte beschreibt und deren Konfiguration erlaubt. Näheres, siehe [8] oder [9].

```
[00:16:53:AB:64:29][LE]> char-desc
handle: 0x0001, uuid: 00002800-0000-1000-8000-00805f9b34fb
```

7 bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.attribute.gatt.characteristic_declaration.xml
8 Robert Davidson et al.: Getting Started with Bleutooth Low Energy. O'Reilly, Sebatopol, 2015. safaribooksonline.com/library/view/getting-started-with/9781491900550/ch04.html
9 Naresh Gupta: Inside Bluetooth Low Energy. Artech Houxe, Boston, 2013.

```
handle: 0x0002, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0003, uuid: 00002a05-0000-1000-8000-00805f9b34fb
handle: 0x0004, uuid: 00002902-0000-1000-8000-00805f9b34fb
```

...Service 1800

```
handle: 0x0005, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x0006, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0007, uuid: 00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0008, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0009, uuid: 00002a01-0000-1000-8000-00805f9b34fb
handle: 0x000a, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x000b, uuid: 00002a04-0000-1000-8000-00805f9b34fb
```

...Service 1801

```
handle: 0x000c, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x000d, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x000e, uuid: 00001624-1212-efde-1623-785feabcd123
handle: 0x000f, uuid: 00002902-0000-1000-8000-00805f9b34fb
```

... herstellerspezifischer Service

Einige 16-Bit UUIDs sind für alle BLE-Geräte Characteristics mit einer festen Bedeutung. „*2a00*“ steht z.B. laut Bluetooth GATT-Spezifikation[10] für den „Device Name“. Bei den anderen drei 16-Bit UUIDs „*2a05*“, „*2a01*“, und „*2a04*“ ergibt dies aber keinen wirklichen Sinn für den Move Hub.

Folgende 16-Bit UUIDs sind für alle BLE-Geräte Deklarationen mit einer festen Bedeutung: „*2800*“ markiert einen „Primary Service“ (gelb unterlegt). „*2803*“ ist die „Characteristic Declaration“ (blau unterlegt)

Die rot unterlegten 16-Bit UUIDs sind für BLE-Geräte generische Descriptoren: „*2902*“ steht für die Konfiguration von Notifications/Idications[11] . Mit *0x0100* aktiviert man z.B. Notifications bzw. mit *0x1000* entsprechend Indications.

Die UUID „*00001624-...*“ ist eine herstellerspezifische UUID. Über den Chracteristic Value Handle *0x0e* des läuft später die gesamte Kommunikation mit dem Move Hub (außer der Abfrage der Device ID).

16-Bit UUIDs sind Abkürzungen der üblichen 128-Bit UUIDs. Diese Abkürzung ist aber nur möglich, wenn eine UUID in den Standard-UUIDs der Bluetoothspezifikation enthalten ist.

2.1.4 Konkrete Beispiel der BLE-Kommunikation mit gatttool

Bei den folgenden drei Beispielen 1)-3) wird jeweils die Characteristic mit dem Handle *0x0e* mit einer Bytefolge (in Pyhton „Byte Array“) beschrieben. Wenn statt *char-write-cmd* der Befehl *char-write-req* verwendet wird, dann wird zusätzlich eine Rückmeldung über die Ausführung des Schreibbefehls ausgegeben.

1) Beispiel: Auslesen der Device ID des Move Hubs (in utf-8 Kodierung bedeutet die Ausgabe „LEGO Move Hub“):

```
[00:16:53:AB:64:29][LE]> char-read-hnd 0x07
Characteristic value/descriptor: 4c 45 47 4f 20 4d 6f 76 65 20 48 75 62
```

2) Beispiel: Setzen der LED des Move Hubs auf Gelb:

```
[00:16:53:AB:64:29][LE]> char-write-cmd 0x0e 0800813211510008
```

3) Beispiel: Ansteuern interner Motor A des Move Hubs für 2560 ms (*0x000a*):

```
[00:16:53:AB:64:29][LE]> char-write-cmd 0x0e 0c0081371109000a64647f03
```

10 bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.gap.device_name.xml
11 bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.descriptor.gatt.client_characteristic_configuration.xml

Um Notifications generell zu aktivieren, muss auf die Characteristic mit Handle *0x0f* der Wert *01 00* geschrieben werden. (Dies erkennt man über die UUID „*2902*“ dieser Characteristic.)

4) Beispiel: Notifications vom Move Hub allgemein aktivieren (Notifications für die einzelnen Sensoren müssen anschließen getrennt noch zusätzlich abonniert werden):

```
[00:16:53:AB:64:29][LE]> char-write-req 0x0f 0100
Notification handle = 0x000e value: 0f 00 04 01 01 25 00 00 00 00 10 00 00 00 10
Notification handle = 0x000e value: 0f 00 04 02 01 26 00 00 00 00 10 00 00 00 10
Notification handle = 0x000e value: 0f 00 04 37 01 27 00 00 00 00 10 00 00 00 10
Notification handle = 0x000e value: 0f 00 04 38 01 27 00 00 00 00 10 00 00 00 10
Notification handle = 0x000e value: 09 00 04 39 02 27 00 37 38
Notification handle = 0x000e value: 0f 00 04 32 01 17 00 00 00 00 10 00 00 00 10
Notification handle = 0x000e value: 0f 00 04 3a 01 28 00 00 00 00 10 00 00 00 02
Notification handle = 0x000e value: 0f 00 04 3b 01 15 00 02 00 00 00 02 00 00 00
Notification handle = 0x000e value: 0f 00 04 3c 01 14 00 02 00 00 00 02 00 00 00
```

Diese neun Notifications melden zurück, welche interne und externe Hardware vom Move Hub erkannt wurde und wie diese konfiguriert ist.

Für die einzelnen Sensoren des Move Hubs kann man jetzt jeweils Notifications abonnieren. In den dafür gesendeten Bytefolgen an die Characteristic mit Handle *0x0e* ist auch die Information enthalten mit welchen Schwellwerten und wie oft der Sensor Notifications verschicken soll.

5) Beispiel: Notifications vom Move Hub für grünen Taster abonnieren (Taster nicht gedrückt, gedrückt, nicht gedrückt, um Sensorwerte zu erzeugen):

```
[00:16:53:AB:64:29][LE]> char-write-cmd 0x0e 0500010202
Notification handle = 0x000e value: 06 00 01 02 06 00
Notification handle = 0x000e value: 06 00 01 02 06 01
Notification handle = 0x000e value: 06 00 01 02 06 00
```

6) Beispiel: Notifications vom Move Hub für Encoder des externen Motors an Port C abonnieren (Motor wird zwei mal kurz gedreht, um Sensorwerte zu erzeugen):

```
[00:16:53:AB:64:29][LE]> char-write-cmd 0x0e 0a004102020100000001
Notification handle = 0x000e value: 0a 00 47 02 02 01 00 00 00 01
Notification handle = 0x000e value: 08 00 45 02 bb 00 00 00
Notification handle = 0x000e value: 08 00 45 02 bc 00 00 00
```

Das Protokoll dieser Kommunikation mit dem Move Hub über den Handle *0x0e* -also welche Bytes in der Bytefolge welche Bedeutung haben - ist im GitHub Projekt *BOOSTreveng*[12] von Jorge Pereira oder in der offiziellen Lego-Dokumentation erklärt.

Später im Pythonprogramm werden die Notifications an Callbackfunktionen übergeben. Ähnlich wie bei einem Interrupt eines Mikrocontrollers, wird eine Callbackfunktion dann ausgeführt, wenn eine Notification vom GATT-Server (hier Move Hub) eingetroffen ist und die Callbackfunktion den Inhalt der Notification erhalten hat.

2.2 Testen der Bluetoothkommunikation unter Linux mit Python und der Bibliothek bluepy

Unter den BLE-Pythonbibliotheken eignte sich *bluepy*[13] am besten, um ähnlich wie mit *gatttool* die vom Move Hub angebotenen BLE-Services zu durchstöbern und zu testen.

Leider konnte diese Bibliothek aufgrund eines Bugs (Stand 2018) nicht für die in Abschnitt 4 beschriebene Steuerung des Move Hubs verwendet werden.

Mit dem Linuxtool *blescan* und dem Befehl *sudo blescan* wird nach dem (eingeschalteten) Move Hub gesucht:

```
Scanning for devices...
    Device (new): 00:16:53:ab:64:29 (public), -54 dBm
       Flags: <06>
```

12 github.com/JorgePe/BOOSTreveng
13 github.com/getsenic/gatt-python

```
      0x12: <10002000>
      Complete 128b Services: <23d1bcea5f782316deef121223160000>
      Complete Local Name: 'LEGO Move Hub'
      Tx Power: <00>
      Manufacturer: <9703004006fe4100>
    Device (new): 78:bd:bc:09:6c:3d (public), -91 dBm (not connectable)
      Manufacturer: <75004204018000078bdbc096c3d7abdbc096c3c01000000000000>
```

2.2.1 Auslesen der vom Move Hub angebotenen GATT „Primary Services“

Nun kommt die Pythonbilbliothek *bluepy* zum Einsatz für die Kommunikation mit dem Move Hub. Die nachfolgenden Befehle werden jeweils in die Pyhtonshell oder in einem separatem Skript eingegeben.

```
from bluepy import btle
from time import sleep

dev=btle.Peripheral('00:16:53:ab:64:29')
sleep(1)
for svc in dev.services:
    print(str(svc))
```

Dies ergibt die drei primary BLE-Services des Move Hub zusammen mit den Bereichen der Handles (als Dezimalzahlen):

```
Service <uuid=00001623-1212-efde-1623-785feabcd123 handleStart=12 handleEnd=15>
Service <uuid=Generic Access handleStart=5 handleEnd=11>
Service <uuid=Generic Attribute handleStart=1 handleEnd=4>
```

2.2.2 Auslesen der vom Move Hub angebotenen GATT „Characteristics“

Dabei ist der erste Service (Handel *0x0c* bis *0x0f*) der einzige Move Hub spezifische, über den auch die gesamte Kommunikation (außer Abfrage Device ID) abläuft. Für diesen Service werden nachfolgend mit Python die Characteristics abgefragt:

```
moveHub=btle.UUID('00001623-1212-efde-1623-785feabcd123')
moveHubSvc=dev.getServiceByUUID(moveHub)
for ch in moveHubSvc.getCharacteristics():
    print(str(hex(ch.handle)), end=' ')
    print(str(hex(ch.valHandle)), end=' ')
    print(str(ch.uuid))
```

Das obige Ergebnis zeigt, dass es nur eine Characteristic gibt, nämlich die mit Handle *0x0e*, über die Daten gelesen oder geschrieben werden.

```
0xd 0xe 00001624-1212-efde-1623-785feabcd123
```

2.2.3 Auslesen der vom Move Hub angebotenen Descriptors der Characteristics:

Die Abfrage der Descriptors mit Python für diesen speziellen Service wird mit folgendem Code erreicht:

```
for ds in moveHubSvc.getDescriptors():
    print(str(hex(ds.handle)), end=' ')
    print(str(ds.uuid), end=' ')
    print(str(ds))
```

In der Python Shell ergibt sich damit:

```
0xe 00001624-1212-efde-1623-785feabcd123 Descriptor <00001624-1212-efde-1623-785feab-
cd123>
0xf 00002902-0000-1000-8000-00805f9b34fb Descriptor <Client Characteristic Configuration>
```

Die BLE-generische „Client Characteristic Configuration“ ist der Handle *0x0f*, über den, die Notifications/Inducations ein- und ausgeschaltet werden (16-Bit UUID 2902).

Die Abfrage aller Descriptors mit Python geschieht über den folgenden Befehl:

```
for ds in dev.getDescriptors():
    print(str(hex(ds.handle)), end=' ')
    print(str(ds.uuid), end=' ')
```

```
    print(str(ds))
```

In der Python Shell wird dadurch der Handle als Hexadzimalzahl, die UUID und die Art des Descriptors ausgegeben:

```
0x1 00002800-0000-1000-8000-00805f9b34fb Descriptor <Primary Service Declaration>
0x2 00002803-0000-1000-8000-00805f9b34fb Descriptor <Characteristic Declaration>
0x3 00002a05-0000-1000-8000-00805f9b34fb Descriptor <Service Changed>
0x4 00002902-0000-1000-8000-00805f9b34fb Descriptor <Client Characteristic Configuration>
0x5 00002800-0000-1000-8000-00805f9b34fb Descriptor <Primary Service Declaration>
0x6 00002803-0000-1000-8000-00805f9b34fb Descriptor <Characteristic Declaration>
0x7 00002a00-0000-1000-8000-00805f9b34fb Descriptor <Device Name>
0x8 00002803-0000-1000-8000-00805f9b34fb Descriptor <Characteristic Declaration>
0x9 00002a01-0000-1000-8000-00805f9b34fb Descriptor <Appearance>
0xa 00002803-0000-1000-8000-00805f9b34fb Descriptor <Characteristic Declaration>
0xb 00002a04-0000-1000-8000-00805f9b34fb Descriptor <Peripheral Preferred Connection Pa-
rameters>
0xc 00002800-0000-1000-8000-00805f9b34fb Descriptor <Primary Service Declaration>
0xd 00002803-0000-1000-8000-00805f9b34fb Descriptor <Characteristic Declaration>
0xe 00001624-1212-efde-1623-785feabcd123 Descriptor <00001624-1212-efde-1623-785feab-
cd123>
0xf 00002902-0000-1000-8000-00805f9b34fb Descriptor <Client Characteristic Configuration>
```

Für den Move Hub spezifischen Service (*0xc* bis *0xf*) werden oben die vier unteren Descriptors angezeigt, wobei die beiden mit Handle *0x0c* und *0x0d* nur formalen Charakter haben.

2.2.4 Konkrete Beispiel der BLE-Kommunikation mit bluepy

Zum Empfangen der Notifications muss ein Delegationsobjekt erstellt werden, dessen Klasse von *btle.DefaultDelegate* erbt. Die (überschriebene) Methode *handleNotification()* dieses Objekts ist die Callbackfunktion für die Notifications. Dem Device wird mit der Methode *dev.withDelegate(MyDelegate())* dieses Delegationsobjekt zugeteilt.

```
class MyDelegate(btle.DefaultDelegate):
    def __init__(self):
        btle.DefaultDelegate.__init__(self)
    def handleNotification(self, cHandle, data):
        print('Notification erhalten : {}'.format(data.hex()))
dev.withDelegate(MyDelegate())
```

Daten werden als Byte Arrays vom Handle *0x0e* gelesen oder dort hin geschrieben.

1) Beispiel: Notifications nach dem Einschalten ausgeben:

Nach dem Einschalten der Notifications mit *dev.WriteCharacteristic(0x0f, b'\x01\x00')* werden dann folgende Initialisierungsnotifications ausgegeben (seltsamerweise erst nachdem zusätzlich Daten (z.B. Dummybefehl LED auf Blau) an das Handle *0x0e* gesendet wurden):

```
Notification erhalten : 0f0004010125000000001000000010
Notification erhalten : 0f0004020126000000001000000010
Notification erhalten : 0f0004370127000000001000000010
Notification erhalten : 0f0004380127000000001000000010
Notification erhalten : 090004390227003738
Notification erhalten : 0f0004320117000000001000000010
Notification erhalten : 0f00043a012800000000001000000002
Notification erhalten : 0f00043b0115000200000002000000
Notification erhalten : 0f00043c0114000200000002000000
```

2) Beispiel: Ansteuern der LED:

Die LED des Move Hubs auf Grün schalten:

```
dev.writeCharacteristic(0x0e, b'\x08\x00\x81\x32\x11\x51\x00\x05')
```

...ergibt:

3) Beispiel: Ansteuern des internen Motors:

```
Notification erhalten : 050082320a
```

Den internen Motor A 2560 ms laufen lassen...

```
dev.writeCharacteristic(0x0e, b'\x0c\x00\x81\x37\x11\x09\x00\x0a\x64\x64\x7f\x03')
```

...ergibt:

Notification erhalten : 050082320a

4) Beispiel: Notifications für grünen Taster aktivieren:

dev.writeCharacteristic(0x0e, b'\x05\x00\x01\x02\x02')

...ergibt:

```
Notification erhalten : 0500823701
```

5) Beispiel: Notifications für Motor an Port D aktivieren:

```
dev.writeCharacteristic(0x0e, b'\x0a\x00\x41\x02\x02\x01\x00\x00\x00\x01')
```

...ergibt:

```
Notification erhalten : 060001020600
```

6) Beispiel: Notifications ständig abfragen:

Um Notifications triggern zu können, muss folgende „Event Loop“-Schleife erstellt werden, wobei jeder Aufruf der Methode *dev.waitForNotifications(1.0)* einen Einzeltrigger mit einem Timeout von 1.0 s generiert:

```
while True:
        # EinzelTrigger für Notifications setzen mit Timeout von einer Sekunde.
        if dev.waitForNotifications(1.0):
            continue
        print('Warte auf Notifications…')
```

Dies ergibt folgende Meldungen, wobei der zweite Notificationsblock durch Drücken des grünen Tasters und der dritte durch Drehen des Motors an Port D hervorgerufen wurden.

```
Notification erhalten : 0a004702020100000001
Notification erhalten : 0800450200000000
Notification erhalten : 050082370a
Warte auf Notifications...
Warte auf Notifications...
Notification erhalten : 060001020601
Notification erhalten : 060001020600
Notification erhalten : 060001020601
Notification erhalten : 060001020600
Warte auf Notifications...
Notification erhalten : 08004502ffffffff
Notification erhalten : 08004502feffffff
Notification erhalten : 08004502fbffffff
Notification erhalten : 08004502f2ffffff
Notification erhalten : 08004502e7ffffff
Notification erhalten : 08004502e0ffffff
Notification erhalten : 08004502ddffffff
Notification erhalten : 08004502dcffffff
Notification erhalten : 08004502ddffffff
Warte auf Notifications...
```

7) Beispiel: Notifications in einem Event Loop als eigenständiger Prozess ständig abfragen:

Im nachfolgendem Beispielcode für die Verwendung von *bluepy* wird der Event Loop als eigener Prozess (Thread) gestartet, der dann unabhängig vom Hauptprogramm läuft (hier die Whileschleife die alle Sekunde das Zeichen „+“ ausgibt). Die globale Variable *stop_flag* ist nötig, damit beim Programmabbruch mit *Strg+c* auch der Event Loop Thread sauber beendet wird. Ansonsten ergeben sich Exception-Fehlermeldungen.

```
from bluepy import btle
from time import sleep
import threading

stop_flag = False # Globale Variable, um dem Notif.Thread ein Stopp mitzuteilen
```

```
class MyDelegate(btle.DefaultDelegate):
    def __init__(self):
        btle.DefaultDelegate.__init__(self)
    def handleNotification(self, cHandle, data):  # Eigentliche Callbackfunktion
        print('Notification erhalten : {}'.format(data.hex()))


print('Ich verbinde...')
dev=btle.Peripheral('00:16:53:ab:64:29') # BLE Device (hier Lego Move Hub)
sleep(1)
dev.withDelegate(MyDelegate()) # Instanzierung Delegationsobjekt für dieses Device
sleep(1)

# Client Characteristic Configuration (0x0f) für das Einschalten der Notifications
dev.writeCharacteristic(0x0f, b'\x01\x00')
sleep(1)
# Dummybefehl (LED auf Grün) damit Notifications ausgegeben werden: Programmabsturz!!!!
dev.writeCharacteristic(0x0e, b'\x08\x00\x81\x32\x11\x51\x00\x05')
sleep(1)

# Notifications für grünen Taster aktivieren
dev.writeCharacteristic(0x0e, b'\x05\x00\x01\x02\x02')
sleep(1)

def event_loop():
    global stop_flag
    while not stop_flag: # Schleife für das Warten auf Notifications
        if dev.waitForNotifications(1.0):
            continue
        print('.',end='')
    print('Notification Thread Tschuess!')

notif_thr=threading.Thread(target=event_loop) # Event Loop als neuer Thread
notif_thr.start()
sleep(1)

try:
    while True:
        sleep(1)
        print('+',end='')
except KeyboardInterrupt:
    stop_flag=True # Notification Thread auslaufen lassen
    sleep(1)
    print('Main Thread Tschuess!')
finally:
    dev.disconnect()
```

3 Python-Bibliotheken für die Bluetoooth Low Energy Kommunikation mit dem Protokoll GATT

Es gibt vier verschiedene Linien von BLE-Bibliotheken für Python. Verwirrend daran ist, dass es für jede dieser Linien noch unterschiedliche „Forks“ gibt.

3.1 BLE Python-Bibliothek gatt (nur Linux)

Diese Pythonbibliothek ist in GitHub unter „*`gatt-python`*“ zu finden. Sie wird im c't-Artikel von T. Harbaum für die Steuerung von Lego Boost, Lego WeDo und Fischertechnik-Robotiksets verwendet. In dem dazu gehörigen Software Repository finden sich übrigens auch Bash-Skripte (also ohne Python) um die BLE-Kommunikation zu testen, den Move Hub via *`gatttool`* anzusteuern oder die für die Bibliothek *`gatt`* nötigen Linuxprogramme zu installieren.

Zentral ist hier ein sogenannter „*`gatt.DeviceManager`*“-Prozess, der als getrennter Thread im Hintergrund sich um die BLE-Notifications kümmert.

Die konkreten Steuerbefehle für den Lego Boost stammen aber aus der Arbeit von Jorge Pereira.

Leider ist diese Bibliothek nicht sonderlich gut dokumentiert und hat vermutlich nur eine kleine Community, die sich um ihre Weiterentwicklung kümmert. Leider kann man mit ihr nur anhand der 128-Bit UUID Characteristic auf Werte zugreifen und nicht über das kürzere Handle.

Die Bibliothek *gatt* wird mittels *pip3 install gatt* installiert. Zusätzlich muss aber im Linux *python3-dbus* installiert sein (prüfen/installieren mit *sudo apt-get install python3-dbus*).

Sie ist aber die einzige BLE-Bibliothek, die nach vielen Tests des Autors einwandfrei auf einem Raspberry Pi funktioniert.

3.2 BLE Python-Bibliothek gattlib (nur Linux, Stand 2018)

Ein Fork von *gattlib* wird auch *pygattlib* genannt. *gattlib* basiert auf einen C-Quellcode der unter GitHub unter *gattlib*[14] zu finden ist. Die Installation mit *pip3* funktioniert aufgrund von Bugs nicht automatisch sondern muss teils manuell über folgende Schritte in der Linuxkonsole ausgeführt werden:

```
pip3 download gattlib
tar xvzf ./gattlib-0.20150805.tar.gz
cd gattlib-0.20150805/
sed -ie 's/boost_python-py34/boost_python-py35/' setup.py
pip3 install .
```

Der Fork *pygattlib* wird entsprechend installiert.

Sowohl *gattlib* als auch der Fork *pygattlib* führen auf einem Raspberry Pi 3 zu Laufzeitfehler auch dann, wenn wie folgt die nötigen Linuxprogramme vorhanden und auf dem neuesten Stand sind:

```
sudo apt-get update
sudo apt-get install libboost-thread-device
sudo apt-get install libboost-python-dev
sudo apt-get install libbluetooth-dev
sudo apt-get install pkg-config
sudo apt-get install libglib2.0-dev
sudo apt-get install python-dev
```

Die Pythonbibliothek *pylgbst* verwendet *gattlib*. Auf dem Ubuntu Betriebssystem des Autors von *pylgbst* Andrey Pokhilko funktioniert *gattlib* unter Python 2.7 offensichtlich fehlerfrei - anders als auf dem Raspberry Pi 3 unter Python 3.5.3. Im Magazin MagPi (Ausgaben 4 und 5 2019) wurden Skripte veröffentlicht, die *pylgbst* verwenden und offensichtlich auf einem Raspberry Pi fehlerfrei funktionieren. Die Verwendung von *gattlib* ist also (Stand März 2020) vermutlich jetzt möglich.

```
HUB_MAC = '00:16:53:AB:64:29'
SET_LED_PINK = b'\x08\x00\x81\x32\x11\x51\x00\x01'
SET_LED_GREEN = b'\x08\x00\x81\x32\x11\x51\x00\x06'

class Requester(GATTRequester):
    def on_notification(self, handle, data):
        print('Notification...',end='')
        print('Notification erhalten Handle: {}  Daten: {}'.format(handle,bytes(data,
'utf-8').hex()))

device = Requester(HUB_MAC, False)
time.sleep(1)

print('MoveHub verbinden...')
try:
    device.connect(True)
except:
    print('BLE-Verbindung fehlgeschlagen')
```

14 github.com/labapart/gattlib

```
devId=device.read_by_handle(0x07)[0]
time.sleep(1)
print(devId)

device.write_by_handle(0x0f,b'\x01\x00')
time.sleep(1)

while True:
    try:
        device.write_by_handle(0x0e,SET_LED_PINK) # LED-Farbe auf Pink
        time.sleep(1)
        device.write_by_handle(0x0e,SET_LED_GREEN) # LED-Farbe auf Grün
        time.sleep(1)
    except KeyboardInterrupt:
        device.disconnect() # BLE-Verbindung beenden
        break
```

Der oben dargestellte Beispielcode funktioniert bis auf den Befehl, der die LED-Farbe setzt. Dies löst einen Neustart der Python-Shell aus jedoch ohne nähere Angebe der konkreten Exception. Auch die Verwendung von GATTResponse für das asynchrone Lesen führt zum selben Laufzeitfehler.

Daher kann (Stand 2018) *`gattlib`* nicht auf einem Raspberry Pi zur Steuerung des Move Hubs verwendet werden. Nach heutigem Stand (März 2020) ist die Bibliothek vermutlich verwendbar.

3.3 BLE Python-Bibliothek pygatt (Linux und Windows)

3.3.1 Normaler Bluetooth-BLE-Dongle

Diese Bibliothek ist für Python unter Linux ein sogenannter „Wrapper“ des in Abschnitt 2.1 vorgestellten Linuxtools *`gatttool`*. D.h. im Hintergrund werden *`gatttool`*-Befehle ausgeführt. Dadurch wird die BLE-Kommunikation recht langsam, weshalb immer wieder Notifications „verschluckt“ werden, also nicht erkannt und folglich nicht an die Callbackfunktion weiter gereicht werden. *`pygatt`* verwendet für diesen indirekten Aufruf von *`gatttool`* wiederum die Pythonbibliothek *`pexpect`*.

Folglich ist *`pygatt`* ohne BLED112-Dongle nicht zu empfehlen.

3.3.2 Bluetooth-Dongle BLED112

`pygatt` kann aber auch unter Windows verwendet werden, dann benötigt man BLE-Dongle BLED112 der Fa. BlueGiga. Dieser Dongle meldet sich beim Betriebssystem – egal ob Windows oder Linux – als serielle Schnittstelle an und bearbeitet die BLE-Kommunikation autonom intern. Pythonprogramme mit Verwendung der Bibliothek *`pygatt`* in Verbindung mit dem BLED112 Dongle laufen daher auf jeder Plattform fehlerfrei, die den Dongle einbinden kann, d.h. Treiber für ihn besitzt wie z.B. Windows 7, Ubuntu 18.04 LT oder Raspian.

Die Bibliothek *`pygatt`* wird unter Linux mittels *`pip3 install pygatt`* und unter Windows mittels *`py -m`*[15] *`pip install pygatt`* (Widows Command Promt) installiert[16].

Die Lego Boost Pythonbibliothek *`pyb00st`* verwendet *`pygatt`*. Ihr Autor von *`pyb00st`* empfiehlt entsprechend, den BLED112 Dongle zu verwenden.

Achtung:

15 Durch den Switch „-m“ muss nicht der Pfad des Moduls pip angegeben werden, wenn es in den Python-Standardverzeichnissen für Module zu finden ist.

16 ACHTUNG, falls auf dem Rechner sowohl Python 3 als auch Python 2 installiert sind: Dann wird bei der Eingabe des Befehls *pip* oft die Version für Python 2 verwendet (überprüfen mit *pip --version* bzw. py -m pip --version), siehe ehmatthes.github.io/pcc/chapter_12/installing_pip.html#pip-on-windows.
Hier unbedingt immer Python 3 und auch die Bibliotheken für diese Version verwenden.

Beim BLED112 Dongle wird in dieser Bibliothek beim Start ein Dongle-Reset durchgeführt. Auf dem Ubuntu 18.04-PC führt dies zu Laufzeitfehlern. Daher muss hier der Dongle mit der Option "reset=False" gestartet werden, was in der Bibliothek *pylegoboost* berücksichtigt ist. Es kann sein, dass auf einer anderen Plattform dieser Reset nötig ist, damit der Dongle fehlerfrei funktioniert.

Fazit: Zusammen mit dem BLED112 Dongle der Fa. BlueGiga kann diese Bibliothek auf einem Windows oder Linux PC als auch auf einem Raspberry Pi verwendet werden. Für die Verwendung unter Linux ohne BLED112 Dongle ist *pygatt* aber zu langsam.

3.4 BLE Python-Bibliothek bluepy (nur Linux, Stand 2018)

Für *bluepy* muss *libglib2.0-dev* installiert sein. Dieses Paket kann mit *sudo apt-get install libglib2.0-dev* nachinstalliert werden. Bzw. kann damit überprüft werden, ob es schon in der aktuellsten Version vorhanden ist.

Die *bluepy* Pythonbibliothek wird mit *pip3 install bluepy* installiert.

Mehr Quellcodebeispiele für die Bibliothek *bluepy* ist in Abschnitt 2.2.4 zu finden.

Leider war es nicht möglich, diese Bibliothek für die stabile Steuerung des Move Hubs einzubinden: Vor allem das Ansteuern von LED führte hier oft zu Laufzeitfehlern.

3.5 Fazit BLE-Python-Bibliotheken und BLE-Dongles

Auf einem Windows 7 PC funktionierte die BLE-Kommunikation nur über den BLED112-Dongle. Somit kommt hierfür nur die Bibliothek pygatt in Frage. Auf einem Ubuntu-PC bzw. einem Raspberry Pi ausgestattet mit diesem speziellen Dongle können dann die selben Skripte wie auf dem Windows-PC verwendet werden.

Für den Raspberry Pi ohne BlueGiga-Dongle bzw. für einen Ubuntu-PC mit einem "normalen" Buetooth-BLE-Dongle wird jedoch besser die Bibliothek gatt verwendet.

4 Bibliothek pylegoboost (Verwendung von pygatt für Windows/Linux und gatt für Windows)

Das Kommunikationsprotokoll sowie die meisten Algorithmen für die Steuerung des Move Hubs wurden hierfür der Bibliothek *pylgbst* von Andrey Pokhilko[17] entnommen.
Für die BLE-Kommunikation wurde der BlueGiga-Teil der Bibliothek *pygatt*[18] sowie die Linux-BLE-Bibliothek *gatt*[19] verwendet. Die Software der Bibliothek erkennt automatisch die Plattform und bevorzugt bei vorhandenem BLED112 Dongle diesen für die BLE-Kommunikation.

Dadurch ist es möglich, das selbe Pythonprogramm

- **auf einem Windows PC mit BLED112 Dongle**
- **auf einem Ubuntu-PC mit BLED112 Dongle**
- **auf einem Ubuntu-PC mit normalem BLE-Dongle**
- **auf einem Raspberry Pi mit / ohne BLED112 Dongle**

auszuführen.

Mittels *pylegoboost* kann das Move Hub Programm auf einem PC entwickelt werden und dann z.B. auf einem Raspberry Pi Zero ausgeführt werden, welcher auf dem Move Hub mitfährt.

In der Zip-Datei *pylegoboost_Vx* befinden sich Beispielprogramme sowie der Ordner

17 Siehe github.com/undera/pylgbst
18 pypi.org/project/pygatt/3.2.0/
19 pypi.org/project/gatt/0.2.7/

pylegoboost, der die Module *__init__*, *bleclient*, *constants*, *movehub*, *peripherals* und *utilities* enthält. **Das spätere Programm zur Steuerung des Move Hubs muss sich im selben Ordner wie die Beispielprogramme befinden, damit die Bibliothek *pylegoboost* gefunden wird.**

In den Modulen von *pylegoboost* befinden sich noch viele "Angst"-*sleep()*-Befehle. Ohne sie würden evtl. Laufzeitfehler auftreten, da bestimmte Kommunikationsschritte zu kurz aufeinander ausgeführt werden. Auch wurde mit solchen *sleep()*-Befehlen der Aufbau und das Beenden der BLE-Kommunikation künstlich verlangsamt, damit keine Timing-bedingten Laufzeitfehler auftreten. Einige dieser *sleep()*-Befehle können vielleicht weg gelassen oder verkürzt werden.

Die Module von *pylegoboost* wie auch die Beispielprogramme sind ausgiebig kommentiert. Darin befinden sich sämtliche Erklärungen der Klassen, Methoden und Attribute, die für die Verwendung dieser Bibliothek benötigt werden.

Ein Klassendiagramm von *pylegoboost* ist in Abschnitt 5.1 dargestellt.

4.1 Vorbereiten eines Raspberry Pi für die Verwendung der Bibliotheken pygatt (nur BlueGgia-Teil) und gatt

Die nachfolgenden Anweisungen gelten sowohl für einen Ubuntu-PC, einen Raspberry Pi 3 als auch für einen Raspberry Pi Zero (W).

Falls die Python 3 Paketverwaltung *pip3* nicht installiert ist:

```
sudo apt-get update
sudo apt-get install python3-pip
```

Ansonsten werden das vermutlich schon vorhandene Linux-Paket *python3-dbus* und die Python-Pakete *gatt* und *pygatt* benötigt.

```
sudo apt-get update
sudo apt-get install python3-dbus
pip3 install gatt
pip3 install pygatt
```

Die Bibliothek *pygatt* wird eventuell eine Warnung ausgeben, dass die dafür nötige Bibliothek *pexpect* fehlt. Da hier nicht der *gatttool*-Wrapper von *pygatt* (Linux) nicht verwendet wird, kann diese Warnung ignoriert werden.

Achtung:

Wenn man unter Linux **zum ersten Mal** (z.B. mit einem frischen Image) mit dem Move Hub via BLE unter Verwendung der Bibliothek *gatt* kommunizieren will, dann muss man **vorher eine „Device Discovery“ durchführen**.

Dafür gibt es das Pythonprogramm *deviceDiscMoveHub.py*, zu finden unter den Beispielprogrammen. Dieses Programm muss bei einem frischen Linux-Image einmal ausgeführt werden, und der angeschaltete Move Hub muss dabei erkannt werden.

Achtung, Achtung:

Auf einem Raspberry Pi empfiehlt sich die Verwendung der Python IDE *Thonny*.

Man kann auch einen einfachen Texteditor verwenden und das Skript dann direkt aus der Bash-Shell mit dem Befehl *python3 meinSkript.py* ausführen. In der Bash-Shell ist im Printbefehl noch ein *flush=True* nötig, damit Zeichen ohne *\n* auch ausgegeben werden. Außerdem wird mit t *strg + C* aufgrund eines Bugs in der Bibliothek *gatt* nur der Device Manager Thread gestoppt, was zu einer Exception führt[20].

20 Siehe hierzu github.com/getsenic/gatt-python/issues/5: In der Datei gatt_linux.py unter /usr/local/lib/python3.5/dist-packages/gatt muss Zeile 89 geändert werden in „self._main_loop = GObject.MainLoop.new(None, False)“. Dies beseitigt den Bug.

Die IDE *Idle* ist (Stand 2018) nicht zu empfehlen: Das Programm läuft unter *Idle* etwa einen Faktor 100 langsamer und *alle* Printbefehle - auch die ohne *\n* werden ausgegeben. In Idle wird wie zu erwarten mit *strg + C* der Hauptthread unterbrochen.
Startet man ein Pythonprogramm innerhalb *Idle*, so wird die Python-Shell gestartet Sie bleibt auch dann noch offen, wenn das Programm zu Ende ist bzw. mit *strg + C* abgebrochen wurde. **Bevor man über ein in der Bash-Shell gestartetes Pythonprogramm erneut mit BLE kommunizieren will, muss man die *Idle* Python-Shell beenden**, ansonsten werden nicht alle Notifications empfangen. Das liegt vermutlich daran, dass die *Idle* Python-Shell immer noch auf die BLE-Schnittstelle zugreift, auch wenn das Pythonprogramm gar nicht mehr läuft.

Auf einem Windows oder Ubuntu-PC kann auch die Pyhton-IDE *Spyder* verwendet werden. Allerdings ist dann darauf zu achten, welchen Pythoninterpreter *Spyder* verwendet und ob für diesen (und nicht für einen anderen vom Betriebssystem installierten Interpreter) auch wirklich die nötigen Bibliotheken in der gewünschten Version vorhanden sind.

Falls Laufzeitfehler infolge von Zugriffsrechten auf die Bluetoothschnittstelle auftreten, dann kann unter Linux mit dem Befehl *groups ${USER}* nachgeschaut werden, ob Zugriffsrechte auf *dialout* bzw. *tty* bestehen. Falls nicht, dann änder man dies mit dem Befehl *sudo usermod -a -G tty dialout ihrUsername*.

4.2 Spezielle Hinweise für die Verwendung des Raspberry Pi Zero W

Stand 2018

Auf den (normalen) Raspberry Pi 3 lassen sich unter Windows entwickelte Python Quellcodes am einfachsten via USB-Stick übertragen. Da der Raspberry Pi Zero (W) keine Benutzeroberfläche hat, kann man das hier nicht so einfach machen. Hier bietet sich das Programm *WinSCP*[21] an, mit dem man mittels Drag-And-Drop zwischen Windows PC und Raspberry Pi Dateien übertragen kann, falls beide Rechner sich im selben Netzwerk befinden.

Dazu muss beim Raspberry Pi Zero (W) jedoch das Protokoll SSH aktiviert sein: Bevor ein neues Image zum ersten Mal im Raspberry Pi bootet muss dafür eine leere Datei mit dem Namen *ssh* (ohne Endung) im Rootverzeichnis erstellt werden.

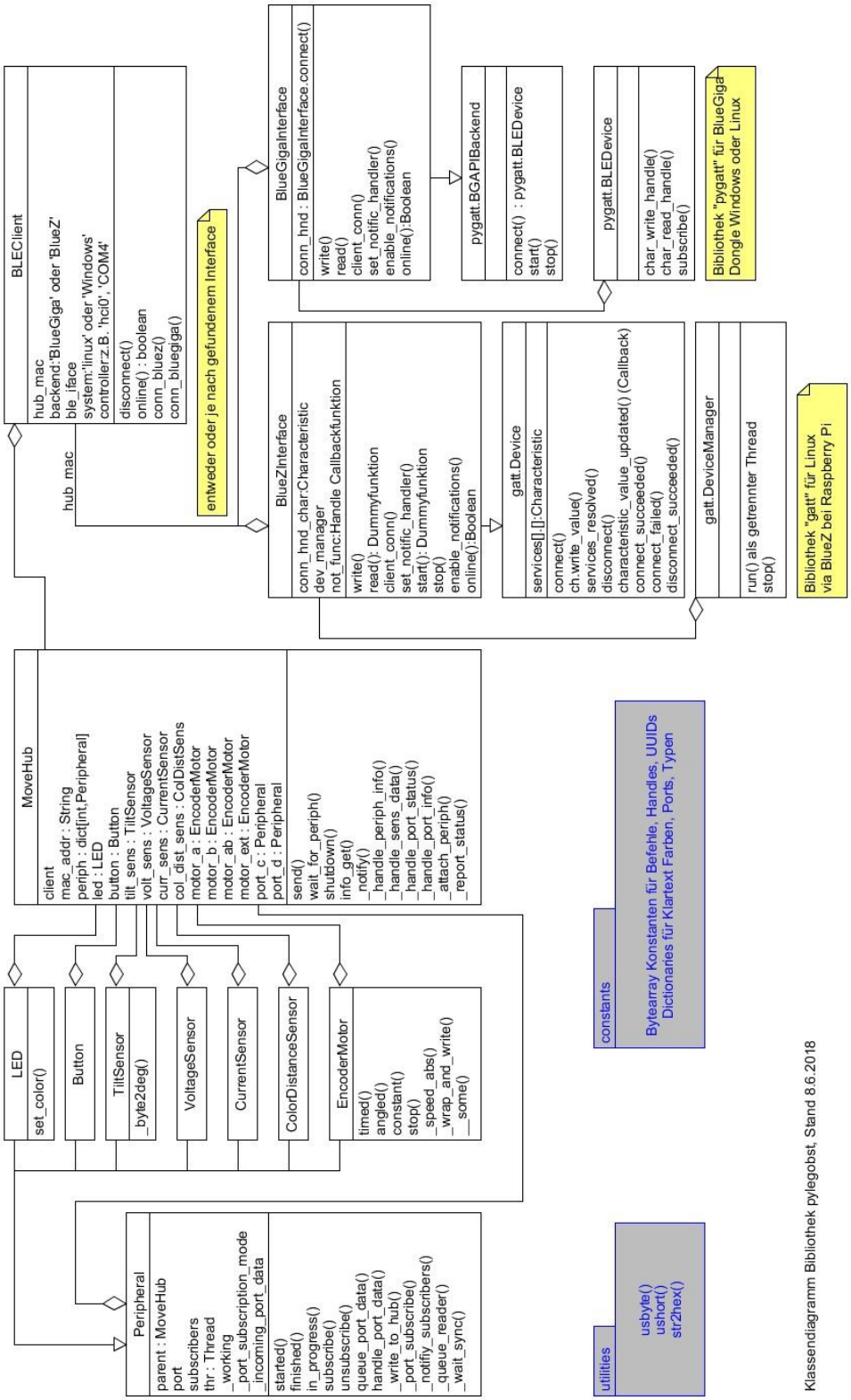
Damit sich der Raspberry Pi Zero W nach dem ersten Booten direkt in ein WLAN einbuchen kann muss vorher noch die Datei *wpa_supplicant.conf* erstellt und dort die WLAN-Zugangsdaten eingetragen werden. Siehe hierfür [22].

Die Pythonprogramme führt man am besten über das Windowsprogram *Putty* aus, indem man damit eine Linux-Bash-Konsole öffnet. Für das Ausführen von Pyhtonprogrammen muss immer *sudo* vorangestellt werden, also z.B. *sudo python3 deviceDiscMoveHub.py*. Denn der Zugriff auf den BlueZ D-Bus verlangt Administratorrechte.

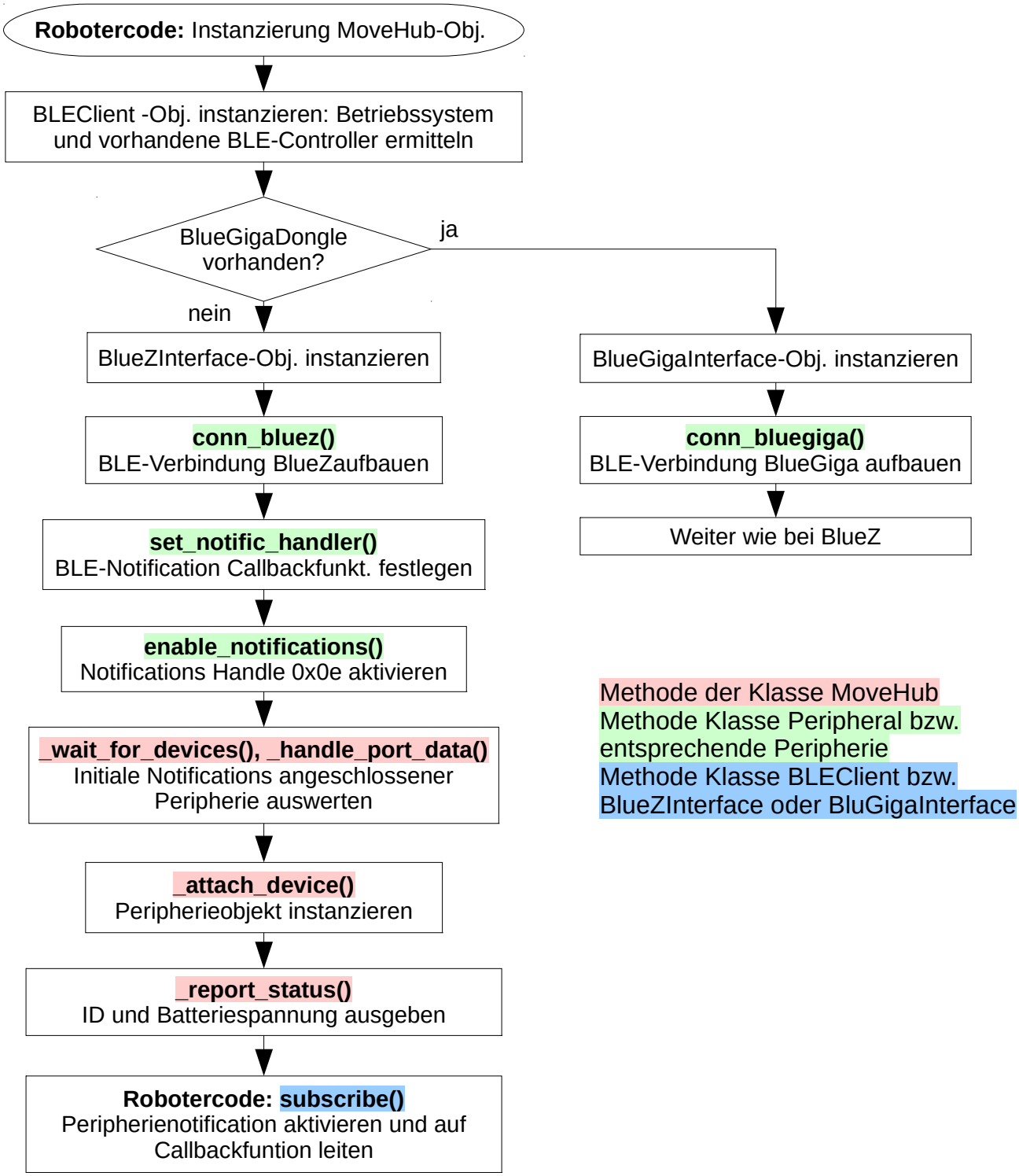
21 Siehe winscp.net
22 pi-buch.info/wlan-schon-vor-der-installation-konfigurieren/

5 Anhang

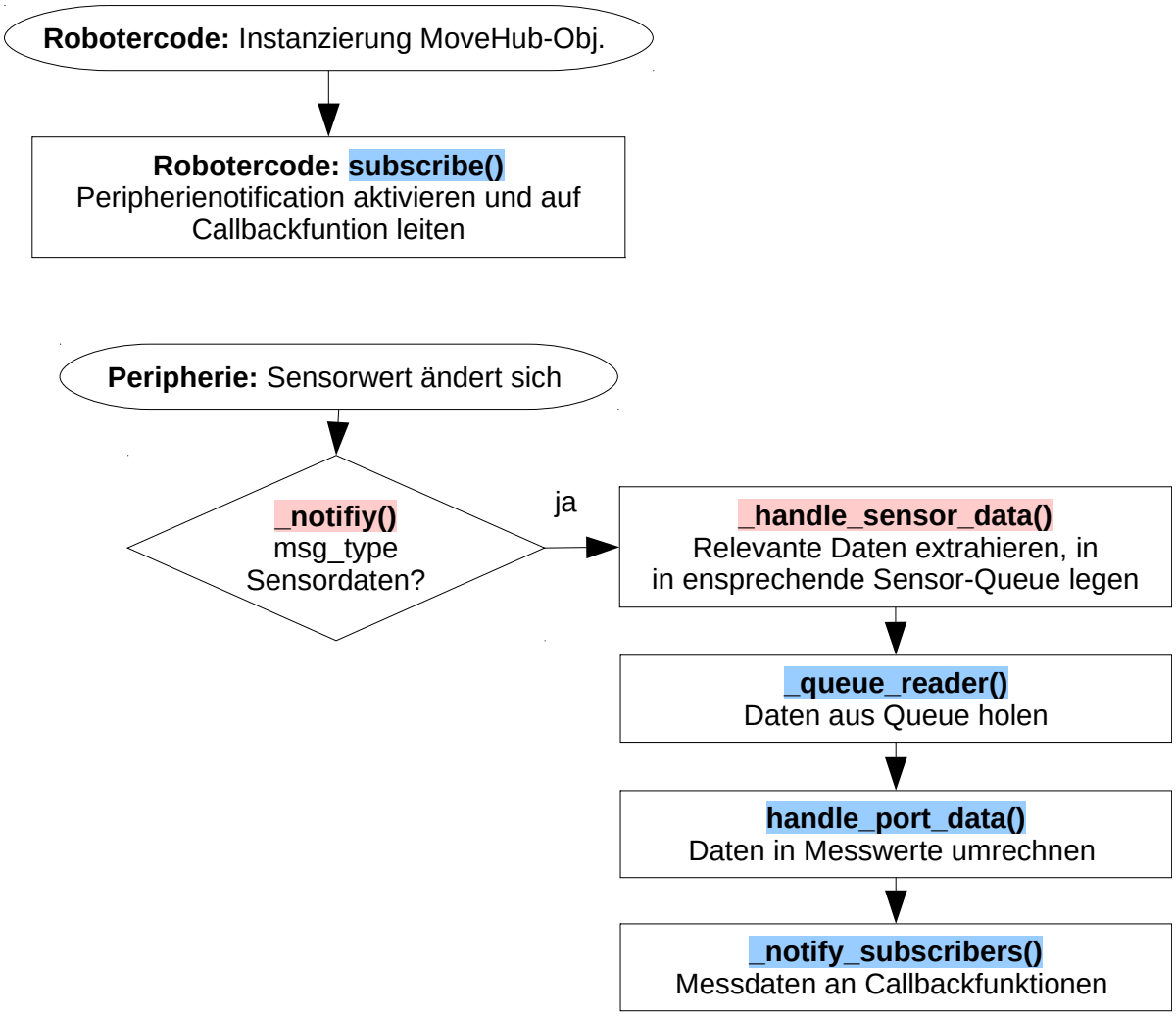
5.1 Klassendiagramm pylegoboost Bibliothek



5.2 Programmablaufplan Instanzierung MoveHub-Objekt



5.3 Programmablaufplan Sensordaten empfangen



Methode der Klasse MoveHub
Methode Klasse Peripheral bzw. entsprechende Peripherie
Methode Klasse BLEClient bzw. BlueZInterface oder BluGigaInterface

5.4 Tipps zum Umgang mit Python

Das größte Manko an Python ist die Tatsache, dass es zwei zueinander leicht inkompatible Pythonversionen 2 und 3 gibt.

Diese Anleitung bezieht sich ausschließlich auf die Pythonversion 3.

Besonders problematisch ist es, wenn beide Pythonversionen auf einem Computer präsent sind, da dann unklar ist, ob Befehle wie *python*, *py* oder *pip* sich auf Python 2 oder Python 3 beziehen. Bei der Verwendung von Tutorials, Fachbüchern und Bibliotheken muss immer vorher geprüft werden, auf welche Pythonversion sich diese beziehen.

Will man die Programmiersprache Python auf dem PC installieren (bei Linux-Computern wie dem Raspberry Pi oder Ubunu-PC meisten nicht nötig), dann stehen mehrere Distributionen zur Verfügung:
Die hier vorliegende Anleitung bezieht sich auf die „grundständigste“ bzw. „Minimal-“ Distribution, welche als Download über *python.org* angeboten wird.

Es gibt sehr umfangreiche Distributionen wie *WinPython* oder *Anaconda* die sehr viel umfangreicher sind (z.B. in Bezug auf vorinstallierte Bibliotheken) und gleichzeitig zusätzliche IDEs enthalten. Verwendet man jedoch einen alten PC oder einen Raspberry Pi, so sollte man diesen Distributionen aus dem Weg gehen, da sie sehr viel mehr Rechenleistung beanspruchen.

Weiter gibt es für jede Pythonversion eine 32 und eine 64 Bit Version. Wenn der PC ein 64 Bit Betriebssystem hat, dann sollte die 64 Bit Version verwendet werden.

Auf Computern mit wenig Rechenleistung bietet sich *Thonny*, mit viel Rechenleistung *Spyder* als optimale Python-IDE an. Es geht aber auch ganz ohne IDE mit einem einfachen Texteditor, der jedoch zumindest über Syntax Highlighting verfügen sollte.

Wird eine Pythonshell gestartet,so zeigt diese vor dem Promt (>>>) Informationen zur Pythonversion an. Im nachfolgendem Beispiel handelt es sich um die Version Python 3.6.5 für 64 Bit:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Den Speicherort der Bibliotheken (Module) erfährt man mit folgendem Code:

```
>>> import sys
>>> for dir in sys.path:
       print(dir)
C:\Users\MeinName\AppData\Local\Programs\Python\Python36\Lib\idlelib
C:\Users\MeinName\AppData\Local\Programs\Python\Python36\python36.zip
C:\Users\MeinName\AppData\Local\Programs\Python\Python36\DLLs
C:\Users\MeinName\AppData\Local\Programs\Python\Python36\lib
C:\Users\MeinName\AppData\Local\Programs\Python\Python36
C:\Users\MeinName\AppData\Local\Programs\Python\Python36\lib\site-packages
```

Die in der Pythondistribution enthaltenen Bibliotheken (Module) erfährt man mit:

```
>>> import sys
>>> print(sys.builtin_module_names)
('_ast', '_bisect', '_blake2', '_codecs', '_codecs_cn', '_codecs_hk', '_codecs_iso2022',
'_codecs_jp', '_codecs_kr', '_codecs_tw', '_collections', '_csv', '_datetime',
'_functools', '_heapq', '_imp', '_io', '_json', '_locale', '_lsprof', '_md5',
'_multibytecodec', '_opcode', '_operator', '_pickle', '_random', '_sha1', '_sha256',
'_sha3', '_sha512', '_signal', '_sre', '_stat', '_string', '_struct', '_symtable',
'_thread', '_tracemalloc', '_warnings', '_weakref', '_winapi', 'array', 'atexit',
'audioop', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'gc', 'itertools',
'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'parser', 'sys', 'time', 'winreg',
'xxsubtype', 'zipimport', 'zlib')
```

Alle verfügbaren (auch nachträglich installierten) Bibliotheken findet man mit dem Befehl:

```
>>> help('modules')
Please wait a moment while I gather a list of all available modules...
__future__          autoexpand          imghdr              runscript
__main__            base64              imp                 sched
_ast                bdb                 importlib           scrolledlist
_asyncio            binascii            inspect             search
_bisect             binhex              io                  searchbase
```

(Ausgabe wurde gekürzt)

Für nähere Informationen zu einer Bibliothek muss man diese zuerst importieren (z.B. *`import serial`*) und dann die Funktion *`help()`* aufrufen (z.B. *`help(serial)`*)

5.5 Fachbücher

Buch	**Bemerkung**
B. Klein: Einführung in Python 3.	Sehr umfassen, gut geeignet für „Programmierumsteiger“ von C
H.-B. Woyand: Python für Ingenieure und Naturwissenschaftler	Beschränkt sich auf das Wesentliche, dafür aber auch Infos zu den Bibliotheken *`Numpy`* und *`Matplotlib`*.
S. Kaminski: Python 3	Fast so umfassend wie das Buch von B. Klein. Recht wissenschaftlich geschrieben, so dass Informatik-Grundlagen von Python
P. Barry, J. W. Lang: Python von Kopf bis Fuß	Sehr unkonventionelles Buch mit einem (für den Autor) sehr guten Didaktikkonzept. Jedoch nicht als Nachschlagewerk geeignet.
J. Ernesti, P. Kaiser: Python 3: Das umfassende Handbuch: Sprachgrundlagen, Objektorientierung, Modularisierung	Absolut umfassend mit über 1000 Seiten. Optimales Nachschlagewerk auch zu etwas weniger gängigen Bibliotheken wie z.B. *`tkinter`*.
A. Allan: Make: Bluetooth. Bluetooth LE Projects for Arduino, Raspberry Pi, and Smartphones.	Irgendwie nicht wirklich hilfreich, da auch kein Bezug auf Python-BLE-Bibliotheken.
K. Townsend: Getting Started with Bluetooth Low Energy	Bestes Fachbuch zu BLE aber leider auf einem sehr hohem Level. Kein Bezug auf Python-BLE-Bibliotheken.